

C for BASIC programmers

Working with the cross-compiler z88dk

By

Jens Sommerfeld and Bodo Wenzel

Translated by Google Translate

Slightly edited by Tim Swenson

Introduction

This paper was taken from a series of articles posted to a German ZX81 bulletin board. The articles were written by Jens Sommerfield and Bodo Wenzel. The articles were posted by Thomas Lienhard. These articles are the only programming tutorial for using the z88dk C cross compiler for the ZX81. The original articles were in German, which did not help a lot of English users.

Tim Swenson used Google Translate to translate all of the articles into English (as best as Google could). Tim then did some minor editing, cleaning the document up and fixing any obvious errors. The English in this paper is not perfect and shows that it was translated, but it should be good enough for an English speaking audience to find it useful.

The structure of the articles follows the same chapters of the ZX81 Manual written by Stephen Vickers.

If any questions arise about the contents of this paper, please refer back to the original articles.

Part 1 – Setting up the ZX81

C is, if we may believe the "Bible" by Kernighan & Ritchie, a programming language for general applications. It is a relatively "low-level" language. We plan to deal with ANSI-C. This dialect is understood by z88dk.

On the whole, it should go here to the possibility of programs that are written in the C language, to translate for the ZX81. We use the z88dk cross compiler in version v2.57. About the compiler and its application already frequently been reported. Here it should also go to both the application and also to programming. We start all over again (the installation), and I (John) have also the possibility of a little more about C compiler. We will try to keep to the BASIC manual by Steven Vickers, that whoever ZX81 users can perform the steps in BASIC and C. So: START ...

All we need is a Windows system (works I work with XP, but under Win98 compiler), an editor (I work with "Notepad2" and "PS-PAD" - even "Notepad +" is very appropriate) and of course the compiler.

First, we invite us to download the file "z88dk-1.8.0-setup.exe". We then start the installation. After accepting the license agreement to install the program automatically. Personally, I have the installation path of c: \ \ program z88dk on c: changed \ z88dk because I have such quick access from a command prompt (DOS) - maybe I do not want just so much to tap :-).

Now we need an editor and then it can with the first program (the famous "Hello World") also begin. So we start our editor and enter the following program:

```
# Include <stdio.h>

int main (void)

{
    printf ("Hello world \ n");
    return 0;
}
```

This program is saved as "hello.c" in the z88dk \ bin directory.

Return to the DOS shell and Cd into the "z88dk\bin". Now we call the compiler at once (just enter zcc): The version number (v2.57) is easy to detect. If we are now to the "DIR" command to view the contents, we should find our file "hello.c": Now comes the first exciting moment. We ask the compiler to translate this small C program. Therefore we call the compiler with this command:

```
c:\z88dk\bin\zcc +zx81 hello.c
```

Here we see the first feature. First, call the compiler with zcc. (Do not forget space) then we enter + zx81. So we give the compiler the hint, the library to use for the ZX81. Then all we need is our hello.c file (again, a space). If we then hit "Newline" the compiler starts to work.

At first glance, it looks confusing. The compiler has now translated our file to a file, which is called a.bin. This format can not understand the ZX81 (yes, we need a. P-file). The a.bin is a binary file, a sort of machine language "raw" program, which has been translated for the ZX81.

Now we need to translate the a.bin a. P-file. That goes with this call:

```
c:\z88dk\bin\bin2p a.bin hello.p
```

The program bin2p makes our raw file (a.bin) a. P-file, we have to give it a name (in our case hello.p). Now we have a file that can be read the ZX81 (or emulator). In an emulator (we recommend the "Eightyone") our file is started running with RUN without problems.

Let's look at the program listing, in which we enter LIST. In line 1, the REM line, the machine code is hidden. When we call LIST 2, we see the following line:

```
2 RAND USR VAL "16514"
```

With this call, the machine language program starts. What is happening here in brief:

- We write a program in C and call it hello.c
- We let the C program translated by the compiler and get a bin file (a.bin)
- We convert the bin file with the program in a bin2p. p-file to (hallo.p)
- This program we can run in the emulator or a real ZX81

Part 2 – Telling the Computer What To Do

In the first part we have told you how, under Microsoft Windows, to get the z88dk running. Here's an addendum or the Quick Reference Guide for all Linux users.

Instead of the file "z88dk-1.8.0-setup.exe", you need the file "z88dk-src-1.8.tgz" from the project page <http://www.z88dk.org> and packs them off to a place agreeable to you. Then according to the instructions in your home directory "z88dk", run the the script "build.sh". There are a few warnings, but finally have all the necessary tools and libraries created. I personally did not want a system-wide installation, so I was already finished, except that I've tinkered a little script called prep.sh that expands the environment variables.

```
# / Bin / sh
export PATH = $ PATH :/ home/bodo/ZX81/ZX-C/z88dkv1.8/z88dk/bin
export Z80_OZFILES = / home/bodo/ZX81/ZX-C/z88dkv1.8/z88dk/lib /
export ZCCCFG = / home/bodo/ZX81/ZX-C/z88dkv1.8/z88dk/lib/config /
```

where is "/ home/bodo/ZX81/ZX-C/z88dkv1.8 /", of course, you set your own path.

Our recommendation to run the compiler is this way:

```
zcc zx81-vn +-Wall-o create-app beispiel.c beispiel.bin
```

Four new options have been added here:

- "Vn" switches the output of the internal program calls. This option can omit it if you would like to see the different stages of the translation, but with the option to drop the possible error messages much more.
- "Wall" turns on all warnings. This makes sense, because as long as even the compiler anmäkelt something, your program may not be correct.
- "Create-app" calls as the last tool appmake on which we automatically generate a P-file. This is actually a big 'P', but that does not do that.
- "-O beispiel.bin" ensures that the resulting binary file has a different name than "a.bin". This is also the condition for the P-file other than "aP" means. In the example this would be "beispiel.P".

That was it. Before you invoke the compiler now, run the above script once per session (in a bash with ". Prep.sh"), or it will not be found.

We have almost the Chapter 1 of the Vickers Manual ("Setting up the ZX81") behind us and get to chapter 2 with the wonderful title "Telling the computer what to do." There is language to use the command line, but you know how to use an editor, so let's leave that out of here.

Unlike BASIC there are no difference between a C program and a straight line running. In C, there is only one program, and we need no line numbers. This is the fact that a C program compiles and then runs as a whole only (At least that's normally so there is no technical reason why you could not also run single statements in an interpreter, except that the ANSI C standard that does not.

C is called a free format language, in which the arrangement of the individual language elements in the source code is released largely, therefore you can the example from Part 1 so write (try out the quiet, do it!)

```
# Include <stdio.h>
int main (void) {printf ("Hello world \n") return 0;}
```

Or also:

```
# Include <stdio.h>
int
Main
(
void
)
{
    printf
    (
    "Hello world \n"
    )
    ;
}
```

```

        return
        0
        ;
    }

```

Of course there are different styles, how many spaces and line breaks are placed where, and within the C-world, there are religious wars to the "right" style. Do not let them bother, choosing a style and stay with it. As in BASIC, there are also statements. Each statement is a semi-colon ';' finished. Likewise, there are some keywords that there are not too many:

auto break case char const continue default Thurs double else enum extern float for goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while

We are not safe in this course discuss all, but most will occur. What immediately strikes you, but: a few words from the first program are not there. OK, "dat we kriejen spähter".

But what happens when you have an error in the program code? Well, try it out! A few examples of faulty source code:

Example 1:

```

include <stdio.h>

int main (void) {
    printf ("Hello world \n");
    return 0;
}

```

The error here is the lack of character '#' in front of include. And the compiler complains about it too:

```

sccz80: "CfBASIC_2-2.c" L: 2 Warning: # 7: return type defaults to int
sccz80: "CfBASIC_2-2.c" L: 2 Error: # 27: Missing Open Parenthesis
Compilation aborted

```

Line 2 will be reported because it probably hoped to find there still some truth. First, the error messages you may seem strange, but it is actually right. Why, would take us too far. You can add that an error message every now and then a row referenced too deep, and that a mistake often takes more messages to yourself.

Example 2:

```

# Include <stdio.h>

main () {
    printf ("Hello world \n");
    return 0;
}

```

Here we have omitted, the keywords `int` and `void`, but the compiler has accepted it and only a little grumbling:

```
sccz80: "CfBASIC_2-3.c" L: 3 Warning: # 7: return type defaults to int
```

Example 3:

```
# Include <stdio.h>

int main (void) {
    printf ("Hello world \n");
    return 0;
}
```

Renaming `main` in general provoked the following error message:

```
1 errors occurred during assembly
Key to file names:
/ = Tmp/tmpXXtjuA1z.o CfBASIC_2-4.c
File '/ tmp / tmpXXSbQeOO.asm' modules 'ZX81_CRT0' symbol not defined
Error in expression _MAIN
```

The actual compiler accepts the source code, but in the later course (at left, but we're really in the near future), the absence of the symbol `_MAIN` discovered. Apparently, the word must mean so `main`!

Example 4:

```
# Include <stdio.h>

integer main (void) {
    printf ("Hello world \n");
    return 0;
}
```

If we tender the word `int` quite as `integer`, we get this message:

```
sccz80: "CfBASIC_2-5.c" L: 3 Warning: # 7: return type defaults to int
sccz80: "CfBASIC_2-5.c" L: 3 Error: # 27: Missing Open Parenthesis
Compilation aborted
```

The row number is correct, and in a short time, the text is also for you meaning.

Example 5:

```
# Include <stdio.h>
```

```
int main (void) {
    printf ("Hello world \ n")
    return 0;
}
```

This is an interesting case: we did not include the semicolon of the first statement. The compiler notes that although:

```
sccz80: "CfBASIC_2-6.c" L: 5 Warning: # 17: Expected ';'

```

But it thinks it is easy and move on. The result is a flawlessly produced program. You've probably already noticed that the line number is too low a row. The reason is that the compiler has only noticed in this line that the semicolon is missing, when it found the return keyword.

Example 6:

```
# Include <stdio.h>

int main (void) {
    print ("Hello world \ n")
    return 0;
}
```

This error could occur more frequently in BASIC programmers: the usual output function is printf with 'f' at the end. Its message shows that print is not known:

```
sccz80: "CfBASIC_2-7.c" L: 4 Warning: # 33: Call to function without prototype
1 errors occurred during assembly
Key to file names:
/ Tmp / = tmpXXmGnYzP.o CfBASIC_2-7.c
File '/ tmp / tmpXXcsKoEE.asm' modules 'CFBASIC_2' symbol not defined
Error in expression _print

```

Summary:

1. A Linux machine can generate C programs for the ZX81.
2. There is only "the program", C does not have direct command input.
3. C does not need line numbers and it does not matter, how to arrange the individual words in the source text.
4. C has statements, each of which is terminated with a semicolon.
5. Errors in the source code angemeckert (for beginners) as part krytische messages. With experience, you'll notice that the output message text is almost always useful.

Part 3 – A History Lesson

Chapter 3 of the BASIC manual called "A History Lesson". So let's look back to 1969.

In the AT & T Bell Laboratories Dennis Ritchie is working on a port of Unix from the PDP-7 on the PDP-11, while the assembler source code to actually be implemented in the B language of his colleague Ken Thomson. Unfortunately B is limited, therefore, the two begin to implement a new language without these restrictions. That takes a detour via the language BCPL until 1973. But now C is sufficiently powerful to implement Unix. That was the first time that an operating system written in a high level language, and the beginning of an unparalleled triumphant, which continues until today.

In 1978, Ritchie and Brian Kernighan's famous C-Bible "Programming in C" was published, which has since been known only as K & R. Unfortunately, as a consequence different dialects of C generated. We know with BASIC there was never a standard. To the language to give a firm foundation in 1989 published by ANSI (the American pedant to DIN) of the C89 standard. This is today respected by virtually all relevant compilers, the "update" of 1999 (C99) has however not yet fully enforced.

At least that is the official version of the story. In reality, the two wanted to write a parody of the Multics operating system and the Pascal programming language, and have the work completed on the compiler, as he was able to translate the following statement correctly (<http://gnu.org/fun/jokes/unix-hoax.html>):

```
for (; P ("\" n"), R -, P ("|")) for (e = C, e -, P ("_" + (* u + + / 8)% 2)) P ( "|" + (* u / 4)% 2);
```

Part 4 – The Sinclair ZX81 as a Pocket Calculator

We come to the chapter "The Sinclair ZX81 as a pocket calculator." This addition to the issue dealt with PRINT, especially the binary operators and the numerical representation. The issue is discussed in detail in Chapter 17, so we go here first addressed some operators of C. More are discussed in next parts.

The following examples illustrate only one statement. This, of course, you have to try to use in a program skeleton, because C yes no "command mode" knows. This program structure looks like an addition as an example:

```
# Include <stdio.h>

int main (void) {
    printf ("result =% d \ n", 1 + 2);
    return 0;
}
```

Here, C is not a distinction, from BASIC, as we know it: there are the four basic arithmetic operations that are evaluated before and after the point-associative rule. As usual, the precedence can be changed with parentheses:

```
printf ("result =% d \ n", 3 * 2 + 2);
```

is $6 + 2$, or 8, whereas

```
printf ("result =% d \ n", 3 * (2 + 2));
```

of course, $3 * 4$, or 12 results.

What stands in the C statement after the decimal point is called the "term". It has a value and a type. Wherever C expects a number, can be an expression.

And so we come already on the concept of data type, that is done in C relatively lax. The "lax" means that almost all of the predefined data types are converted into each other, without that you have to write down the explicit. The compiler inserts the appropriate conversion code automatically. We will come back later again. But what are the data types are there? The following table shows, the values apply to the z88dk (The forum has no tables ...):

Type		Minimum	Maximum
char	1 byte	-128	+127
unsigned char	1 byte	0	255
short	2 bytes	-32768	+32767
unsigned short	2 bytes	0	65535
int	2 bytes	-32768	+32767
unsigned int	2 bytes	0	65535
long	4 bytes	-2147483648	2147483647
unsigned long	4 bytes	0	4294967295
float	6 bytes	?	?
double	6 bytes	?	?

We come back again to the guys, if we treat variables. Unfortunately can edit the z88dk for our favorite no-point numbers (float), so we restrict ourselves (int and associates) on the integers. In a future version we are sure once the point numbers are available.

Besides the four basic arithmetic C knows a whole lot of other binary operators. Some of them we will learn during the course, but with the following, different from the well-known operators in BASIC, you can do here, you are already familiar:

- << The result of the value is shifted to the left to the left by the number of bits on the right side as shown.
- >> << just right.
- % Divide the value on the left by the value on the right side, the result is the remainder of the division (modulo-bill).
- & The result is the bitwise AND of the two values.
- | The result is the bitwise OR of the two values.
- ^ The result is the bitwise exclusive-OR of the two values, for assembler freaks: XOR.

Except the positive and negative signs, there are among others still following interesting unary operator:

~ The result is the bitwise inversion of the value, so the 1's complement.

The various operators are prioritized as usual, a complete presentation is available in C-hopefully every book. (If you want more programming in C, is worth the purchase of the Bible as "The C Programming Language" by Kernighan and Ritchie, but please, the 2nd edition with the subtitle "ANSI C". C Good references finds her but also by are looking up in Wikipedia or by Googling. A free e-book for example, under http://download.galileo-press.de/openbook/c_von_a_bis_z/galileocomputing_c_von_a_bis_z.zip.)

Integers, you can write out as usual decimal. But if you preferred hex numbers, the number is initiated, then you can all ten digits and letters (big or small)' with '0 x 'A' to 'F specify'. Example: 0x40AB.

To make clear that the value should be unsigned (eg unsigned int), it is a 'U' behind: 54321U. A "wide" value (long) is characterized by an 'L', both together go: 7654321UL. Otherwise numbers are always considered as int

Homework

1. Write a program that returns the sum of the two hex numbers ABC and 123.
2. Take a capable calculator and determine the binary notation of the operands in the following terms.

First calculates the result by hand and then compares it with the output of the C program:

```
printf ("%d \n", 85 & 240);  
printf ("%d \n", 85 | 240);  
printf ("%d \n", 85 ^ 240);  
printf ("%d \n", 170 & 15);  
printf ("%d \n", 170 | 15);  
printf ("%d \n", 170 ^ 15);
```

"Shifted to the left 1 each from 1 to 12 bits" you let the 12 results from spending. Tip: Of course these are 12 printf statements.

You let the 12 results of "12345 each to the right shifted by 1 to 12 bits" on startup.

Part 5 - Functions

We are just as beautiful on computing, here comes the Chapter 5, "Functions," just right. Functions are written in C, not only the method to perform mathematical figures, the concept goes much further. We'll come back, but only in part 14, until then you have to wait you ...

C as a language does not have its own key words for mathematical functions. If you review the list of keywords in part 2 again that is clear. Instead, functions are provided as subprograms in the so-called libraries. In standard C environments, and likewise when you need z88dk you responsible for these barely. The choice of words "no" for two reasons:

Before a function is called, it should be known to the compiler. This is accomplished by specifying the appropriate # include line, which we will explain in more detail later. The mathematical library (with sine, logarithm, etc.) for example is not part of standard automated, so it must also be specified explicitly on the left. This needs us but not to worry right now because we can not use anyway-point numbers.

Before we go into further details, once a practical example with the random number function, as pure as before printf line:

```
printf ("result =% d \ n", rand ());
```

However, we need to add another # include line, otherwise the compiler unhappy. This line you write directly under the existing:

```
# Include <stdio.h>
```

Let the program run several times, and you noticed that every time a different value is output. This is because rand () has an internal variable that is not cleared by RUN.

What is now a function in C? These are three things:

- The type of the return value, if a function does not return anything, this is the type of void. C makes no distinction between "functions" to return value and "procedures" with no return value.
- The name of the function, this must start with a letter (which includes the underscore '_', but no accents) start, then be allowed to follow other than letters and numbers. C distinguishes between uppercase and lowercase letters!
- The list of parameters, each parameter has a type and a name. A function can also be defined without parameters.

This function declaration is like the footprint, it is also referred to as a prototype. For the above position, he used random number is:

```
int rand (void);
```

The rand () function is to say "edge" (oh, what!) delivers a return value of type int, and the call gets no parameters. It is customary to make such an empty parameter list with void clearly.

The function to remove a sign called abs (). She gets an int as a parameter, the sign is removed by the statements of the function, and the value returned is then dated and unsigned int, with a corresponding prototype looks like this:

```
int abs (int value);
```

You can try this feature simply:

```
printf ("Result 1 =% d \ n", abs (-123));
```

```
printf ("Result 2 =% d \ n", abs (+123));
```

When calling a function, always brackets in the market! If you do weglasst, which is also a valid C expression. However, it has the value of the address of the function, and it is not called. The course you can try again:

```
printf ("result =% d \ n", rand);
```

A complete list of all possible functions can not be given at this point, because it just the size and the number of libraries depends on what features are available. However, there are some standards that are listed in all C-books. We are in this course learn some functions, and certainly you can also learn more at the forum.

Those who are very curious, can the various files with the extension "*. H" in the "include" See the z88dk installation. These are plain text files, which inter alia also a lot of function prototypes included. Otherwise, we must refer to the documentation of z88dk, which is in a Wiki (<http://www.z88dk.org/wiki/doku.php>) on the Internet. There are at least partly represented many functions.

Summary:

- Functions are subroutines in C.
- A function always has a name, a type which may be void, and a list of parameters that can be empty.
- Names must begin with a letter, upper and lower case letters are different.
- Before you call a function, the compiler must know the prototype. For standard functions, this is achieved by the error # include line.

Homework

The following program is wrong. Find the error, without entering it and compile. Only then use the compiler.

```
# Include <stdio.h>

int main (void);
{
    printf ("result =% d \ n", 1 + 2);
    return 0;
}
```

Also, main () has a "footprint", and we write also always go the return statement. So we can return a value to the BASIC. Write program which takes its calculation result returns (eg a random number). The returned value is then output through a BASIC statement.

Part 6 - Variables

This part is dedicated to the variables, because Vickers also says "Variables".

Create variables

First, a variable in C has a type and a name. We come to that later, but already we make you mouth water with the concept that there is not only the so-called "primitive" data types such as integers. You can even arrange the whole database as the data type of a program. But for now let's stick with the known data types ...

For the names of variables the same applies as for the names of functions, and in general these rules apply to all of you declared objects. Later we will find many more object types except variables and functions.

- Uppercase and lowercase letters are distinct.
- The underscore counts as a letter. Umlauts are, however, no letters. (Only modern languages like Java allow almost any character in the name.)
- A name that starts with a letter.

The ANSI standard requires that at least the first 31 characters can be distinguished. (This is all for the so-called "internal" binding. When we begin to share our program separately translated into modules, we also get "external" bonds. Here writes the ANSI standard provides for only 6 characters, and it must not more uppercase and lowercase letters are distinct. Most C systems are not so limited.)

Similar to the ZX81 BASIC variables must be declared before they are used. Firstly we present their types and their names. An example:

```
# Include <stdio.h>

int main (void) {
    int summand1;
    int summand2;
    int sum;

    summand1 = 123;
    summand2 = 468;
    sum += summand1 summand2;
    printf ("Sum of% d and% d =% d \n", summand1, summand2, sum);

    return 0;
}
```

Unlike BASIC be distinguished in C scopes and lifetimes of variables. In BASIC, all variables throughout the program known and exist from the moment in which they were first assigned a value. In C, it depends on the place where a variable is defined, and on two key words called static and auto.

The following example demonstrates this once:

```
# Include <stdio.h>

int main (void) {
    int known;

    printf ("known =% d \ n", available);

    known = 123;

    {
        auto int automatically;
        static int statisch1;

        printf ("automatically =% d", automatically);
        automatically = 987;
        printf ("% d \ n", automatically);

        statisch1 statisch1 = + 1;
        printf ("statisch1 =% d \ n", statisch1);

        Known known = + 1;
    }

    printf ("known =% d \ n", available);

    {
        auto int automatically;
        static int statisch2;

        printf ("automatically =% d", automatically);
        automatically = 987;
        printf ("% d \ n", automatically);

        statisch2 statisch2 = + 2;
        printf ("statisch2 =% d \ n", statisch2);

        known = known + 2;
    }

    printf ("known =% d \ n", available);

    return 0;
}
```

Let the program run three times and the issues you listed.

First, you see that in the program, other blocks are set off by curly brackets. As already said, is the nature and depth of the indentation, and the number and arrangement of the blank lines and spaces matter of taste, it should be clear just what is meant.

In the outer block is a variable "known" is defined, which apparently is also known within the inner blocks, and can be used. If you where a new value is assigned, will remain selected even when leaving an inner block.

In the inner blocks, two variables are defined. The first, "automatic", the auto storage class, and the second, "static", has defined static. The output of the program prove their properties:

- Variables can only be defined at the beginning of a block.
- static variables are directly related to their definition of a defined value, with all the bits are 0.
- auto variables are directly on their definitions (the beginning of their life) a random value.
- The lifetime of static variables begins with the existence of the program. They live to the end of the program. When z88dk there is a peculiarity: even if the program is terminated with the return from the main () function, the values of the static variables get.
- The lifetime of auto variables ends at the end of the block in which they are defined. If the block is executed again, these variables are "created" new.
- The scope of a variable extends from the beginning to the end of the block in which they are defined, including therein subordinate blocks. This means that the names (not the child) blocks used for other variables in other without conflicts will arise. (This applies to z88dk unfortunately only for auto variables! The compiler is not ANSI compliant, but for our purposes it is enough. Statisch1 and statisch2 The two variables were meant to both are called "static", but that resulted in an error.)
- Variables without storage class are within blocks auto variables.

Conversion between data types

As promised, we now return to the conversion between data types. There is a concept that is called cast. This English word in this context means "die", and that pretty well describes what it's about.

If we have an expression of a certain type, but need a different type, we use a cast:

```
short kurzer_wert;  
longer_wert long;  
  
kurzer_wert = (short) longer_wert;
```

Before the expression is thus simply the desired data type is written in parentheses. The compiler builds possibly a necessary conversions to the machine code. This can be created simply by machine code, that no statement to quite complicated, namely the call to a conversion function translated.

Conversions between integers, the compiler of unequal types before automatically, you need not actually write the same. This implicit casting is seen but bad if you read the source code. Writes so please always cast out explicitly, the machine code generated is the same - but the next reader of the

program will thank you!.

Homework

- Makes you even thought, why a name must begin with a letter.
- Write a program that defines eight variables with the possible different integer types (see Part 4). It is the variable of type long unsigned value "(200UL << 24) + (200U << 8) + 200" to assign, and then the other variables. The value of this variable Let you spend all eight variables. What do you observe?
- Modify the program so that the first variable of type signed char is assigned the value "-100", and thereafter. The other variables, the value of this variable What are you watching now?

Part 7 - Strings

Now that we've been able to deal with numbers, we come to the strings, commonly known as "strings".

By BASIC because we are pretty spoiled: Strings are a separate data type to manage the corresponding space we need not worry us. This is in C (unfortunately? (The positive thing is that the programmer in turn knows exactly when which memory area is used for what. At least he should know!)) Completely different, even if the existing programs have given a different impression.

In C there is a small integer type (on the property as a cross between a character and integer we come back in part 11.), Namely char. Of course, we can create and use a variable of this type, which you already know. In another example, you learn the same formatting option ("% c") for printf () (What printf () is still everything you learn in detail in 17th part Until then we will anticipate one or the other necessities.) Know:

```
# Include <stdio.h>

int main (void) {
    char character;

    sign = 'X';
    ("C content of signs =% \ n" character) printf;

    return 0;
}
```

Please note that the character constant is enclosed in single quote, that is the sign above the number '#' character to the usual German PC keyboard, to the left of the Enter key.

In order to store a string can create an array of char:

```
# Include <stdio.h>
# Include <string.h>
```

```

int main (void) {
    char string [30];

    strcpy (string, "This is a string.");
    printf ("string =% s \ n", string);

    return 0;
}

```

This example shows three facts:

- Arrays are defined very similar to normal variables. The difference is the dimension by specifying the number of elements in square brackets.
- C has no data type "string": we can not assign the variable string constant directly (Later we return to a method that allows something like that but ...). Therefore, we use the `strcpy ()` ("string copy") that copies a string into another, taking the target specified as the first parameter, the source, the second parameter. So the compiler knows this function, another include line can be specified.
- In our example, the source string is a constant. Nanu, but C does not have a data type "string"? Well, here is the exception to the rule! String constants C accepted, they will be recognized at the double quote (above the 2 on the usual German PC keyboard). This is a real concession ...

The standard library contains many functions that work with strings. Principle must string variables that are the target of an operation, to be sufficiently large. If that is not the case, we have one of the famous buffer overflows, which are responsible for many of today's vulnerabilities!

The following paragraphs provide some examples, imagine all functions would be beyond the scope considerably. If you want to know more, check out the corresponding header files (*. H) and read the documentation. There are some treasures to discover - and it does not pay to reinvent the wheel!

Concatenate strings

With the function `strcat ()` ("string concatenate") a string is attached to one other:

```

# Include <stdio.h>
# Include <string.h>

int main (void) {
    char string [30];

    strcpy (string, "This is");
    strcat (string, "a string.");
    printf ("string =% s \ n", string);

    return 0;
}

```

Determine length of a string

Of course, even the length of a string can be determined, if the argument of the function `strlen()` ("string length") is a string constant, some compilers can determine the length of itself and then immediately add the appropriate number in the code. For the duration of the program, in this case there will be no more call.

```
# Include <stdio.h>
# Include <string.h>

int main (void) {
    printf ("length =% d \ n", strlen ("This is a string.));

    return 0;
}
```

On this occasion, the question arises how C determines the actual length of a string, the character array is indeed usually larger. This is done by the special null character that comes after the last character. In this way, a string can (as part of the available space) be as large as desired, without anywhere space is held for a specified size.

You already know the special character `'\n'`, which marks the end of the line. In an analogous way, the null character is encoded with `'\0'`. There are more such special characters, which are discussed in the section 11.

So if I depositing the string "Hello \n" in a variable `char text [10]`, the array contains afterwards following content:

```
text [0] text [1] text [2] text [3] text [4] text [5] text [6] text [7] text [8] text [9]
'H' 'e' 'l' 'l' 'o' '!' '\N' '\0' (old) (old)
```

Note: (old) here means that this array element was not changed by the deposition of "Hello".

In virtually all C implementations of this null character is encoded by a byte with the value 0, so can not this character be part of a string (of course, I can write down a string "Hello \0Welt" which is also 12 bytes. But usual string functions then take the end after the 'o'.).

Compare strings

To compare two strings, the function `strcmp()` ("string compare") is used. The result is returned as a number:

- A value less than zero means that the first character string "smaller" than the second.
- A zero value means that the two strings are equal.
- A value greater than zero means ... you can imagine it. ;-)

An example:

```

#include <stdio.h>
#include <string.h>

int main (void) {
    char text1 [10];
    text2 char [10];

    strcpy (text1, "bigger");
    strcpy (text2, "SMALL");

    printf ("Comparison 1 =% d \ n", strcmp (text1, text2));
    printf ("Comparison 2 =% d \ n", strcmp (text1, text1));
    printf ("% compared 3 = d \ n", strcmp (text2, text1));

    return 0;
}

```

The decision on the "size" of the string is like based on the characters in the character set encoding; decides the first different character. If a string is shorter than the other, but until then they are equal, it is also considered "small".

String Search

Of the many functions which are used for searching strings, we mention just three examples. Unfortunately requires knowledge of their detailed explanation of pointers; (! Wrong) they are supposed to be the hardest part C, and only come in part 21

- strchr () ("character string") finds the first occurrence of a character in a string.
- strrchr () ("reverse character string") searches for the last occurrence of a character in a string.
- strstr () ("string string") searches a string in another string.

Convert strings to numbers

Because expressions in C compiler implemented in unchangeable machine code, there is no method that is similar to the BASIC function VAL. But there are other than the equivalent of printf () (it is the scanf () function, we treat them in Part 9) nor the simple function atoi () ("ascii to integer"):

```

#include <stdio.h>
#include <string.h>

int main (void) {
    int number;

    number = atoi ("12345");
    printf ("result =% d \ n", value);
}

```

```
        return 0;
    }
```

Please note the other header file!

Convert numbers to strings

The function itoa () ("integer to ascii") is the general partner to atoi ():

```
# Include <stdio.h>
# Include <string.h>

int main (void) {
    char text [10];

    itoa (text, 23456);
    printf ("result =% s \n", text);

    return 0;
}
```

Homework

- Writing a program that you the contents of all elements of the example output above (Table of char text [10]). Also makes certain that array indices always start with zero! For the output of the fourth member you can eg : Write

```
printf ("%4 characters =% c \n", text [3]);
```

- As the "empty" string is ("") encoded in bytes? How many characters or bytes are needed?
- In the example to strcmp () you will surprise the result safe. Apart from the absolute numerical value does not matter, you would surely have thought that "bigger" is less than "SMALL" is because the "G" so before the "K" comes. Declared the result ... Keywords are: upper and lower case letters, ZX81 character set and ASCII. Experimentation can run a lot of comparisons to her, as by a number of such statements:

```
printf ("result =% d \n", strcmp ("text1", "text2"));
```

Part 8 – Computer Programming

The chapter 8 of Vickers' BASIC book is called "computer programming" and he shows us how programs are entered and edited. Is what we have now in the past, because you do it with C since the Part 1 of this course. An editor and the compiler call their masters so already.

Therefore, we will focus in this part of the visible details of the translation, who do not care about that

can easily take part in the next attack.

Survey

A C program goes through when creating most of the steps (Some C systems summarize individual levels This limits the generality of the following is not a but..)

- The source code is typed into a text editor and saved to a file. This file is called source code, and it has the extension ". C" (with a small C!).
- The preprocessor filters the source and cleaned him. The comments Here, the pre-processor to run. If the result is stored in a file, it generally receives the extension ". I". This is certainly an acronym for intermediate file.
- The actual C-compiler translates the text in Assembler text. This text may end up in a file that is then usually have the extension ". S" (small S!) Wins. This is a source file.
- Now the assembler text is translated by the assembler into machine code. This is stored in a so-called object file, which therefore has the extension. "O".
- The machine code in the object file is not executable because he realized only the statements of the C source code. For the complete program still need to be added to the functions from the standard libraries. For most target systems like our Zeddy also need the addresses are calculated (step-and variable addresses) and used. This work is called "binding" and is performed by the linker. So that the program is basically done; z88dk receives when the resulting executable (the executable) the extension "bin.".
- As a specialty in z88dk still needs a minimal BASIC program tinkering around the finished machine program. Only then will we have reached our goal: a P-file!

How good that we all Aufruferei is the so-called front-end "zcc" removed. Therefore, limited calling for us on a command line. We can view each of the steps individually ...

Editor

This step, I will not treat large. Your use hopefully an easy to handle text editor that can hold multiple files open. It is also particularly helpful when it detects the C syntax and color (but not too colorful) represents. And if he still can perform with a mouse, such as the command-line compiler invocation and represent the issue, you are almost at the optimum.

As an example of this part will serve the following program, stored as source code in the file "CfBASIC_8-1.c":

```
/* We use functions of the standard input and output */
#include <stdio.h>

/* The constant EXIT_SUCCESS is defined here: */
#include <stdlib.h>

/* This program is just
 * Hello, world!
```

```

* From:
* /
int main (void) {
    puts ("Hello, world!");
    return;
}
// That's it!

```

Preprocessor

The preprocessor has two important functions: it removes all comments and performs all his own commands. In the next section we come back to that. In brief:

Comments begin with the character sequence `/*` and end, possibly in a later line, with the combination of characters `*/`.

The commands of the preprocessor begin with the character `#` as the first character of the line, even for tabs and spaces at the beginning. The next word is the actual statement, it does include already. Certain statements lead to substitutions in the code - but that we can see in the next part carefully.

With the following command line, the translation is terminated after the preprocessor; `zcc` is also kind enough to provide the same file name for the temporary file as the source, but with the extension, `.i`.

```
zcc + zx81-vn-Wall-E-1.c CfBASIC_8
```

If you are now in the file `"CfBASIC_8-1.i"` looks, you'll find still valid C code, but it is much greater than that. `"C"` file. The preprocessor has indeed removed the comments, but he has also run the include statements and pasted the contents of the specified header files. `"Our"` source code is shown at the end.

Interesting and important in some cases for debugging are two things:

- The lines beginning with `#` contain, next is always a number and sometimes a file name in double quotes. Thus it is stated from which file which line number with the following lines are, so that the compiler can generate useful error messages when needed.
- The preprocessor has replaced certain texts. In our example this is the constant `EXIT_SUCCESS`, which was replaced by the number `0`.

Compiler

If we want to see the generated assembler source code, we can run the translation with the following command line:

```
zcc +zx81 -vn -Wall -a-1.c CfBASIC_8
```

Because the generated assembly-optimized text, the `z88dk`-makers have decided that the file should have the extension `"opt."`. Look So the file `"CfBASIC_8-1.opt"` to; the interesting part is near the beginning: 6 (in words: six) lines of assembler ...

Assembler

If we are in part 25 for modular programming come, we will find the following command line. They finished the translation after the generation of the machine code of the C source code:

```
zcc zx81-vn +-Wall-c-1.c CfBASIC_8
```

This creates the file "CfBASIC_8-1.o". It now contains binary data that you can imagine with a corresponding program portfolio (eg a HexViewer). Accurate contemplation of revealed that both individual names such as "MAIN" and "PUTS" and the machine code and our text stand in it.

Linker

Now is an executable program can be created. For this we simply omit the "-create-app", but it is useful to specify a file name for the destination file:

```
zcc +zx81 -vn -Wall CfBASIC_8-1.c-o CfBASIC_8-1.bin
```

The resulting file "CfBASIC_8-1.bin" now includes the "pure" machine code, including the necessary subroutines for puts (). Also, you can file this example you with a view HexViewer: the names of the functions are no longer included because the processor they do not need to run. But we find our output text again ...

P file converter

This last step is only relevant for the ZX81. While the executable "CfBASIC_8-1.bin" already the finished machine program, but for convenience we need a BASIC "framework" that include unpacked the binary code in a REM line. This work makes appmake the utility, which is accessed by zcc if we add the option "create-app". And so we are back to the familiar command line:

```
zcc +zx81 -vn -Wall -create-app -CfBASIC_8 1.c -o CfBASIC_8-1.bin
```

The resulting P-file contains only two lines of BASIC. Line 1 contains the REM with the machine code, line 2, the RAND USR for the call. If you want to write a complete work of C and BASIC, so you have to write only complete the C-section, the load P-file to a ZX81 emulator or there and then write the BASIC part.

Views by zcc

If you the option "-vn" when calling zcc, you see all the command line that executes zcc. Here is a brief overview of some the lines are wrapped because our issue is too narrow. ;-) The highlighted words euer_pfad, temporaer1 temporaer2 and will vary depending on the installation and call. The actual line endings are marked with "↵".

A little program is necessary to start your main function. Such a piece is like "crt0" probably "C Runtime part 0". The first command line create a temporary copy:


```
cp euer_pfad/z88dk/lib/zx81_crt0.opt temporaer1.opt
cp temporaer1.opt temporaer1.asm
```

Now the preprocessor is invoked:

```
zcpp -I. -DZ80-DSMALL_C-DZX81-D__ZX81__-DSCCZ80 -Ieuer_pfad/z88dk/include
CfBASIC_8-1.c temporaer2.i
```

Now the compiler's turn:

```
sccz80 - / / Wall temporaer2.i
```

Whose assembler source code is not that great, so it is optimized in two stages:

```
copt euer_pfad/z88dk/lib/z80rules.2 <temporaer2.asm> temporaer2.op1
copt euer_pfad/z88dk/lib/z80rules.1 <temporaer2.op1> temporaer2.opt
```

The result can be translated into machine code:

```
z80asm-IXIY-Eopt-ns-Mo temporaer2.opt
```

Interestingly, the assembly acts as a linker:

```
z80asm-a-m-Mo oCfBASIC_8-1.bin -Ieuer_pfad/z88dk/lib/clibs/ndos
-Ieuer_pfad/z88dk/lib/clibs/zx81_clib -Ieuer_pfad/z88dk/lib/clibs/z80iy_crt0-IXIY temporaer2
temporaer1.opt. o
```

Finally, the translator for the P-file is called:

```
appmake +zx81 -b-c-1.bin CfBASIC_8 temporaer1
```

That's it! You've survived ...

Part 9 – More Computer Programming

In Chapter 9 Vickers tells us about "More computer programming." This includes such beautiful commands such as RUN, CONT, STOP and BREAK. This we can not apply in C because the current C program is a single BASIC statement for the Zeddy. We discuss in this part the C counterparts to REM and INPUT, and something more ...

Comments

There is nothing like a well-commented program. And unlike BASIC programs where comment lines actually affect the performance (at least when the program as the ZX81 in an interpreter is running),

you can get your C programs to comment in detail without feeling guilty. Because the comment will only not be passed to the actual compiler, as we have learned in the last part.

Comments are the combination of characters `/*` and end up going to the combination of characters `*/`, over several lines. For C++, there is also the one-line comments starting with the combination of `//` and stop automatically at the end of the line. In a pure C code, I expect only C comments of `/*` and `*/`:

```
/* This is a multiline comment.
 *
 * Especially nice it looks when the left margin asterisk
 * Below the other.
 *
 * Of course, these comments are not nested
 * Be!
 */
```

Professional sources have comment blocks by convention to look at a project is always the same:

- On header are inter alia Information on the purpose of the code is, who wrote it, when, and often that he has history behind it. Also, make notes for special circumstances (such as must be compiled, what are the operating limits it, ...) well.
- Before each function (BASIC programmers translate: "subroutines") is a comment block with a short description of the function, its parameters, if any, and their eventual return value.
- Comments in the current code should not repeat this in prose, but the reader to realize that what is important is not obvious, but for understanding.
- Dividing lines divide the source code pretty, and it becomes much more readable. I agree that my source code like in a kind of chapter.

The space in the ZX-Team Magazine will now not be wasted by an example. Seek out professional source code or just such programming guidelines and check them out. The whole Kommentiererei is pure matter of taste and thus always cause for discussion. Let's not stop you anyway! ;-)

Lines of code wrap

Although you have learned the same at the beginning that C is unformatted and therefore you can wrap the individual words of the source code and distribute unlimited, there are cases in which lines are to be wrapped in the code, although it really needs to be a single line.

The most important case is the preprocessor. You must be on one line, but it looks sometimes just not good. The junction of two lines is done exactly by a backslash at the end of the line. There must be no other characters more! Example:

```
#include \  
    <stdio.h>
```

Another case are long strings, they can share it by dragging them to a point where it separates an

double quotes:

```
printf (
    "Hello,"
    "World!"
    "\ N"
);
/* Is the same as: */
printf ("Hello," "World" "\ n");
/* The same as: */
printf ("Hello, world \ n");
```

Preprocessor

Besides the comments of processed preprocessor nor his own instructions. Interestingly, he has a different syntax than C, which is confusing for many beginners! But he can also be used for any other purpose, not only for C sources. I let him successful and meaningful also filter assembly language source code, these files usually have the extension (with a capital S) ".S". It's best to realize you that are compiler preprocessor and two pairs of shoes - they have really nothing in common except that they are often seen together ...

All preprocessor directives begin with a pound sign "#" at the beginning of the line. Indentation with spaces or tabulator are allowed, but rare. After the character is the actual statement, it can also be separated by white space. Some instructions have then arguments, but all the instructions just go to end of line. Unlike C statements they receive but no semicolon at the end!

Insert files: #include

This statement you already know. It means "joining at this point the source code of the specified file." Therefore, the instruction requires a filename as a parameter, which is equivalent, in two versions:

```
# Include <stdio.h>
# Include "modul.h"
```

Each input file is the preprocessor searched in the list of paths. Therefore, you also need to explicitly specify any paths, on the contrary, it is almost always harmful. If you yet again angebt a path, as a separator between the directory and file name forward slash "/" is used, even under Microsofts operating systems!

The spelling with angle brackets makes the preprocessor in the list of his known system search paths. Hereby files are mainly affecting the standard library included.

If the file name is enclosed in double quotation marks, on the other hand the first folder of the source code einfügenden searched. Such files have therefore to do with the program to be created.

The contents of the file to be naturally turn his C source. What is there in detail, is released: anything is possible. The normal application, the declaration of "objects" such as constants, variables, and

functions. Because such declarations at the head of a source code are necessary so that the compiler knows them in the later use, such files are called "headers".

Text Replacements: # define and # undef

With this statement, the preprocessor into a powerful tool for editing the source code, and it is also driven fast and loose so much. After # define a name is expected, and noted the rest of the line, the preprocessor under that name.

Let's start with the simple applications, the definition of symbolic constants (they are called "symbolic" because the source code by a symbol, that is a name, and can not be represented by their value.). How it looks:

```
# Include <stdio.h>
# Define TEST NUMBER 23

int main (void) {
    printf ("TEST NUMBER =% d \ n", TEST NUMBER);
    return 0;
}
```

Anywhere in the code, where now "TEST NUMBER", the preprocessor will replace the text of the name "23". This has two advantages:

- The symbol "TEST NUMBER" is clearer than the number itself (maybe not in this example, but in general!)
- If this value is used in its meaning more than once, it is prone to errors, perform a proper change. Too easy a job is forgotten. Using # define can not happen, because the concrete value is indeed only one place!

You can choose any combination of constants defined in this way with each other. The preprocessor replaces the symbols recursively:

```
# Define LENGTH 42
# Define WIDTH 23
# Define AREA (LENGTH * WIDTH)
```

Because these constants are assembled somewhere in the source code, you should hang them so that the right thing happens. If in the example, the symbol "AREA" in the source code occurs, the preprocessor replaces it by "(42 * 23)." The product in the sample can be calculated by the compiler. If it can not, it just happens to the duration of the program.

Each preprocessor / compiler ANSI standard has defined constants, each starting and ending with two underscores, the most important to you are the following sample program (more familiar z88dk apparently not):

```
# Include
```

```

int main (void) {
    printf ("filename =% s \ n", __ FILE__); / * text * /
    printf ("Line number =% d \ n", __ LINE__) / * number * /
    printf ("Date Created =% s \ n", __ DATE__) / * text * /
    printf ("Creation time =% s \ n", __ TIME__) / * text * /

    return 0;
}

```

As described above, successive strings are indeed directly connected to a single. Therefore, we have for example also can write:

```

printf ("Added: =" __ DATE__ "\ n");

```

With a # define once specified icon is valid until the end of the source code ... Or until you explicitly with # undef it again removes from the memory of the preprocessor.

As if the whole thing were not enough, we are still a top of it: you can create the symbol with parameters, and reference it in the replacement part. This corresponds to the Makroprogrammierung other occasions, so you will also often called "macro" to refer to defined with # define symbols. A small example? Here you go:

```

# Include <stdio.h>
# Define output (value) printf ("count =% d \ n", value)

int main (void) {
    int variable;

    variable = 81;
    output (23);
    output (42);
    output (variable);

    return 0;
}

```

Please take care of when you where semicolons sets. Here you have to know exactly what, when and where will be replaced as to avoid really hard to find bugs! As shown in the last part, then the output of the preprocessor is a great help in troubleshooting.

If you want, you can with # define C also completely switch to German, but that is one of the applications, which is really stupid (As mentioned before, the madness has method: The <http://www.ioccc.org>. Most of the submissions are working a lot with the preprocessor).:

```

# Define all main
# Define printf gib_formatiert_aus

```

```
# Define whole number int
# Define if if
# Define else else
# Define first {
# Define end}
/* ... and so on */
```

Conditional Compilation: # if, # else, # endif

While developing her notes now and then that you want to leave out a block of lines for testing purposes. Or do you want to let out all the debug statements from the "production code". Or you have to let compile dependent on certain circumstances different lines. Here, too, helps the preprocessor.

Begins with # if a source area that is passed only to the compiler if the condition after the # if is true. This section ends with # endif. # Else with an alternative range is introduced, and there is also a combined # elif else-if.

We have not had any comparison, but let's just say: as a condition for the # if everything counts as a "true", which does not have the value 0. Of course, you can only query conditions that are known at compile time. The preprocessor does not run the program itself!

Additionally, there is a kind of function that asks whether a macro is defined. Formerly preferred specific instructions # ifdef ("if defined") and # ifndef ("if not defined") were used, but there can be no further linkage occur with other conditions. With defined (name), the thing is more flexible.

An example will make things clearer sure:

```
# Include <stdio.h>

# Define GERMAN 1
# Define ENGLISH 0
/* # Define DEBUG uncomment for debugging */

int main (void) {
    # If GERMAN
        printf ("Hallo world \n.");
    # Elif ENGLISH
        printf ("Hello, world \n.");
    # Else
        printf ("\n?");
    # Endif

    # If defined (DEBUG)
        printf (__DATE__);
    # Endif

    return 0;
```

```
}
```

Other: # error, # pragma

There is much more to say about the preprocessor, but it would blow up this way too much of even more. Even veteran C-veterans often do not know all the possibilities ... Therefore now have only two simple instructions, which you can use meaningful.

Error with the translation is aborted. The rest of the line is displayed in the message:

```
# Include <stdio.h>

int main (void) {
    # If defined (GERMAN)
    printf ("Hello world \n.");
    # Else
    # There is no language defined error!
    # Endif

    return 0;
}
```

Single compiler can be controlled by specific instructions in their behavior. To the # pragma is used. If a compiler does not understand the rest of the line, he can safely ignore it, even if it was not the desire of the programmer. When z88dk can e.g. with a # pragma statement the address of the HRG-memory set:

```
# Pragma output hrgpage = 36096
```

That was already a huge bite, there is no homework! Surely you yourself ideas what you want to try it. Therefore, we now come back to more practical matters, namely the ...

Entering values

So far we can create simple variables, assigning values to them, link them together and output the result. But in complete communication with the user's missing is the opposite direction, the reading of values that the user enters.

The complementary function to printf () ("print formatted") is scanf () ("scan formatted"). She is at first glance very practical, but has quite a few quirks. Professionals they put not as a welcome, at least not their simple version (more on that below).

Analogous to printf () indicates the first argument, which is to be read. For integers, you use "% d", for single characters "% c", and for string "% s". The remaining arguments must be pointers to target variables - but we actually get pointer later. For now it is enough if it is going on the recipe ...

The implementation under z88dk also seems to be even less perfect, as we have read in the forum. So

you can see anything at all (read: the slow mode is active), it extends the appeal of the compiler to the "-startup = 2".

Let's go with an example in the solid:

```
# Include <stdio.h>

int main (void) {
    int number;

    printf ("input");
    scanf ("% d", & number);
    printf ("\ nEingegeben =% d \ n", value);

    return 0;
}
```

Please be sure that the call to scanf (), an ampersand "&" before the variable name "type" is on! Let it away, and - well, you'll see ...

If you let the program run, it detects that you see your entry can not. You have to type blindly, so to speak. Even after the entry with the Enter nothing appears except the output by printf (). This is clearly an error in the standard library of z88dk.

If you continue to experiment with it, you realize that even with the input characters outside "0" to be completed "9". But only the space or Enter to produce the expected result. Here, too, I assume error in z88dk. The usual scanf () to accept and use as many characters as it can. Therefore I expected after typing "1" - "2" - "3" - "x" and the reading of "123", but it seemed to "0". Too bad.

Let's talk about two other input and output functions. First, there puts () ("put string"):

```
# Include <stdio.h>

int main (void) {
    puts ("Hello");
    puts ("World.");

    return 0;
}
```

This is the super simple output function that returns a string. No other parameters, not numbers. Quite simple. And as you will notice when trying out, will even automatically output a newline, so the next issue begin on a new line.

The counterpart to this is gets () ("get string"). This function expects a string (or more precisely: a pointer to one) as a parameter; there must be enough space to receive the input from the user. Unfortunately, there is no control (Of course there is an extended version that offers a length limit. This

is fgets (), whose detailed description does not come in this part.), So we hereby can provoke the wildest crashes when we type too much! The implementation in z88dk succeeded better than scanf () so that we can see our entry and even edit:

```
# Include <stdio.h>

int main (void) {
    char line [10];
    int number;

    printf ("input");
    gets (line);
    sscanf (line, "% d", & number);
    printf ("\ nEingegeben =% d \ n", value);

    return 0;
}
```

This example calls instead of the normal scanf () on the modification of sscanf () ("string scan formatted"), look for the additional "s" and the additional argument at the beginning. This function also converts character to a number, but used the characters in the given string instead of direct input. This string we have read through just before gets ().

Usually, there are other input and output functions, such as the Use single character. They are called putchar () ("put character") and getchar () ("get character"). There are also more variable variants of them. The problem at this point is that I (Bodo) did not get z88dk with the running ...

From experience with the input and output I derive the recommendation to use the ZX81 C programs only support a BASIC program. We will undoubtedly get to know other uses, in particular I'm curious about the MRO capabilities that I have not tried at all. But for normal input and output are the BASIC commands compared to z88dk more reliable - but that is not symptomatic of C programs in general ...

Part 10 - If

So far we can only write programs that run straight ahead. Therefore it is high time for the chapter 10: "If ...", with its fabric, a program can make decisions.

Let's start with an example:

```
# Include <stdio.h>
# Include <string.h>

int main (void) {
    char input [20];
```

```

puts ("A JOKE SHOULD I tell?");
gets (input);
if (strcmp (input, "Gents AB") == 0) {
    puts ("\ nSCHON WELL, I RUN IT.");
Else {}
    int value;

    puts ("\ nWIEVIELE PAGES HAS A GUITAR?");
    gets (input);
    sscanf (input, "% d", & val);
    if (value == 6) {
        puts ("\ NJA, REALLY.");
    Else {}
        puts ("\ nNEE, 6: UP, DOWN, RIGHT,");
        puts ("LEFT FRONT, REAR.");
    }
}

return 0;
}

```

For programming decisions, there are only two key words: "if" and "else". The simple "if" you know already from BASIC: if the condition is true, run something, otherwise not.

In C, after the "if" is a term given in parentheses. The parentheses is clear what belongs to the term, therefore, is not a keyword such as "then" necessary. Within the parentheses, you can formulate an arbitrary (integral) expression, if it has a value other than 0 (zero) results, it is considered "true", a value of 0 is "false."

What is running in the real case is, after the closing parenthesis. This can be a single statement or a block of statements enclosed in braces. To avoid problems and mistakes, you should always use curly braces even if only one statement comes. Because it counts just not what the programmer thought, or was engaged as well, and it is one that the compiler recognizes!

If there are alternative instructions to be executed in the wrong case, this can be specified after the keyword "else". Also recommended the consistent use of braces! The specification of the "else" is optional of course.

For the formulation of logical operations is C provides some operators. Here are the binary operators that are between two operands:

&& The result is true if both values are true. Otherwise, the result is false. So this is the logical AND, not to be confused with the operator "&", which performs a bitwise AND operation.

|| The result is true if at least one of the two values is true. Otherwise, the result is false. So this is the logical OR, not to be confused with the operator "|", which performs a bitwise OR operation.

== The result is true if both values are equal. Otherwise, the result is false. In BASIC here handed a single '='.

! = The result is true if both values are equal. Otherwise, the result is false. BASIC was at this operator '<>'.

< The result is true if the first value is less than the second value. Otherwise, the result is false.

> The result is true if the first value is greater than the second value. Otherwise, the result is false.

<= The result is true if the first value is less than or equal to the second value. Otherwise, the result is false.

> = The result is true if the first value is greater than or equal to the second value. Otherwise, the result is false.

There is also a unary operator:

! The result is the opposite of the following value. BASIC was at this operator 'NOT'. Comparing the operator with the "not equal" to.

Incidentally, it is for the reader of a source text (and you are two weeks later myself!) Very helpful in the parentheses of the "if" always to write an expression directly in "true" or "false." Does not use the knowledge that the value 0 is "false" means! Compares itself:

```
/* Short, but misleading: */  
if (value) { /* ... */
```

```
/* Better and more clearly: */  
if (value != 0) { /* ... */
```

Within a statement block with braces by the way needs to be no statement. Likewise, a single semicolon is a valid statement, that the "empty". Thus, the following constructs are syntactically correct, I will not speak of logic here:

```
if (this == this) {  
  Else {}  
    puts ("This is the unequal.");  
}
```

```
if (zx == 81);  
else;
```

The expression of an "if" may of course also be assigned to a variable. Whose data type is usually an "int", but many real programs use the data type "int" to the two possible values "false" and "true" that you have to define the emergency itself. Such clever concepts we treat but only at the end ...

Very professional, it is finally, the names of these variables begin with "is" or "has", or in German "is" with or "has", eg "Ist_ok" or "hat_handy".

Homework

- What is the program?

```
# Include <stdio.h>
```

```
int main (void) {  
    int num1;  
    int number2;  
  
    number1 = 12;  
    number2 = 81;  
    if (num1 > 15)  
        puts ("class.");  
        if (num2 < 15)  
            puts ("Great.");  
    else  
        puts ("Why, yes.");  
  
    return 0;  
}
```

- Give the following program and let it run. Predicts ahead of the result!

```
# Include <stdio.h>
```

```
int main (void) {  
    int result;  
  
    result = 2 == -4;  
    printf ("2 == -4:%d \n", result);  
    result = result!;  
    printf (":, result "(2 == -4)%d \n");  
    result = result!;  
    printf (":, result "(2 == -4)%d \n");  
  
    return 0;  
}
```

- Write the expression for "if A is B or C is" out. Test it with a program.

- In ZX81 BASIC is an instruction "LET C = (A AND A > B) + (B AND A < B)" and he always returns the larger of the two values A and B. C is not the ... As you would have to write that?

- Results indicate that the following expressions? Please have considered that in mind before you try it with any one program!

1 + 2 * 3

10% 3 * 3 - (1 + 2)

(1 + 2) * 3

5 == 5

x = 5

- With x = 4, y = 6, z = 2, which? The expressions are true or false, and why Here, too, his own

reflection is called for, not simple trial and error.

```
x == 4
x = y - z
z = 1
y
```

- Writing a program which, when an input number of points to output the corresponding note.

The following rules apply:

More than 30 points give a score of 1;
21-30 points make the score 2;
11-20 points make the score 3;
Less than 11 points arise mark 4
Sample output:

```
ENTER THE SCORE: 25
FOR 25 POINTS IS THERE A second
```

Part 11 – The Character Set

Now that we have already seen in the last part of the branching structure as, for example, we could equally continue with the loop. But Mr. Vickers entitled his chapter 11 with "The character set". Therefore, it is this part of their character and data type "char" and its integral nature, the loops come next, I promise!

C also has a set of characters - and not just one, but two! The ISO standard defines a character set initially for the source code of the program, which must be understood by the compiler. The other character set is used to represent the input and output ... You look so quizzical ... Maybe I need to explain again what is a character set.

It's really quite simple: define a set of characters with which code symbol to be encoded. The code is an integer representing the Zeddy 6 bits wide (The seventh bit indicates bytes that are to be executed in the image output as a machine instruction, and covers "N / L" (newline), which in reality is a "STOP" .. . But the explanation would lead us too far. The eighth bit is the inversion of the sign.), the original ASCII 7 bits wide, 16 bits in Unicode, and UTF-8 variable width.

In C, characters are declared with the data type "char", which we have already had. This data type is guaranteed 8 bits wide and can therefore sign with the values -128 to +127 or accept unsigned values 0 to 255. Try as a try:

```
# Include <stdio.h>

int main (void) {
    char character;
    signed char mit_vorzeichen;
    unsigned char ohne_vorzeichen;
```

```

    sign = 0;
    mit_vorzeichen = character;
    ohne_vorzeichen = character;
    printf ("%d %d %d \n", sign, mit_vorzeichen, ohne_vorzeichen);

    sign = -1;
    mit_vorzeichen = character;
    ohne_vorzeichen = character;
    printf ("%d %d %d \n", sign, mit_vorzeichen, ohne_vorzeichen);

    sign = 81;
    mit_vorzeichen = character;
    ohne_vorzeichen = character;
    printf ("%d %d %d \n", sign, mit_vorzeichen, ohne_vorzeichen);

    sign = -81;
    mit_vorzeichen = character;
    ohne_vorzeichen = character;
    printf ("%d %d %d \n", sign, mit_vorzeichen, ohne_vorzeichen);

    sign = -128;
    mit_vorzeichen = character;
    ohne_vorzeichen = character;
    printf ("%d %d %d \n", sign, mit_vorzeichen, ohne_vorzeichen);

    sign = +127;
    mit_vorzeichen = character;
    ohne_vorzeichen = character;
    printf ("%d %d %d \n", sign, mit_vorzeichen, ohne_vorzeichen);

    return 0;
}

```

You see, that the omission of "signed" or "unsigned" the z88dk automatically "signed" results. This is often the case, but not always! The data type "char" can in some C compilers come by default as "unsigned".

Of course you can expect with these variables, all with an integer that "width" is just so. The type "unsigned char" is therefore the preferred type when their byte variables applies.

Now when we think about what fonts are valid for z88dk, we only reiterate that the source is encoded in ASCII. This is not surprising, because the compiler yes on today's operating systems is normal, simple texts in ASCII encoding.

To find out what character set at runtime is, we write the following little program:

```
# Include <stdio.h>
```

```

int main (void) {
    char character;

    sign = '0 ';
    printf ("%d = %c \n", character, character);

    sign = '9 ';
    printf ("%d = %c \n", character, character);

    sign = 'A';
    printf ("%d = %c \n", character, character);

    sign = 'Z';
    printf ("%d = %c \n", character, character);

    return 0;
}

```

And the result: apparently also the ASCII is used. The output is then used by the routine of character output automatically to the recoded ZX81 character set, which is also documented in such, and the behavior can be changed. But that's fodder for a later part.

We would also like the special character in the definition of character and string constants (the apostrophe, the quotes and the backslash) can enter (even if these characters when ZX81 almost all do not exist: we have not only to do with this PC, but may also work 'times are on another.'). This is possible (English: escape) by a special escape character that is used as the escape character, you know it already, because you have so that the newline character encoded in the output text. It is - tataa - the backslash! The following table shows the possible combinations that lead to one character:

tabelle.gif

Note 31: Well, actually, here, the value must be 10, but the z88dk has apparently swapped with '\r'!
 Anemrkung 32: Well, actually, here, the value must be 13, but this has apparently z88dk with '\n' confused!

What probably the three combinations '\?', '\', And '\ are "? The last two things logically, they are used in character constants and string constants, so the use of the apostrophe and the quotation mark as limiting. When there is a question mark but just those in their faces! Well, that comes from a very unhappy selected and unknown "feature", the three-character sequences. With this, certain characters can also be specified on systems with limited character sets, code ... But so I will not burden that uses eh 'nobody'!

Interesting to the table for Bitfrickler are the last two lines. You can encode any characters in octal or hexadecimal. This is very handy for special characters.

So if we leave now, the original chapter's events, we find that the functions in C and CHR \$ CODE is required at all. A value of type char is always at the same characters and integer, and its interpretation depends only on the context. You will notice that this is not a problem.

As we at the (low-resolution) graphics characters come, material for another part will be. Until then have fun with the past!

(check this section for the table from the main article (part 11))

Part 12 - Looping

So far, our C programs were indeed rather boring because they always ran straight, maybe even broken up by a branch. In this section we come to the "looping"!

Shorthand for calculations

But before it goes off, you shall have some nice shorthand for common instructions learn. Such statements are calculations that look for example like this:

```
variable = variable + 23;
```

The content of the variable is therefore associated with a value and the result is again assigned to the variable. First, that's fine, but the double typing the variable name is the one nuisance for another somewhat prone to error if I mistype me. Therefore knows C a short form of such calculations:

```
variable += 23;
```

The operator can be any of the binary operators, except the logical ("&&" and "||"). Why not just, well, who knows? The expression on the right side does not have to be a constant, but can be complex. And for the most common of these calculations, there is a further simplification.

```
variable ++;  
variable --;
```

... you can each reduce the increment and decrement operators:

```
variable ++;  
variable --;
```

Now you guess why C ++ is as follows: it is the next step after C, so C is increased by one. Why is it not D? That you have to ask Bjarne Stroustrup, its inventor. Incidentally, there is actually a language D (<http://www.digitalmars.com/d/index.html>), which is also derived from C. ...

Maybe I've already told you this: each assignment is also simultaneously an expression. He has the data type and the value of the assigned income. So it is with the increment and decrement. There are,

however, divided in two variants:

```
var1 = 42;
var2 = var1 + + / * var1 is then 43, as var2 * /
var1 = 81;
var2 = var1 - / * var1 is then 80, var2 but 81! * /
```

It is therefore the position of the operator: he stands in front of the variables, only the calculation is performed, and then the value of the overall expression determined. If the operator appears after the variable, the "old" variable value is the value of the overall expression.

For Loop

The well-known for loop C of course knows. She comes from a single keyword:

```
# Include <stdio.h>

int main (void) {
    int i;

    for (i = 1; i <10; i + +) {
        printf ("%d \n", i);
    }

    return 0;
}
```

As in "if" after the keyword is a pair of parentheses. It has three parts, separated by semicolons: the initialization, the loop condition (an expression that evaluates to true or false), and the loop statement. Each of these parts can also be empty. If the loop condition is empty, the for loop is an infinite loop. With the following construct the program does nothing except "to spin in a circle":

```
for (; );
```

After the parenthesis is a single statement or a block of statements is enclosed in braces. That's the part that is between the BASIC and FOR TO NEXT.

In the example above you can see a simple counting loop that counts from 1 to 10 and outputs the respective count. But there are also pretty nifty loops still great to read. In the following example, the arrangement of the parts is broken up in order to make the structure clear:

```
for (file = finde_erste_datei (), /* 34 */
     file = KEINE_DATEI;! /* 35 */
     file = finde_naechste_datei ()) { /* 36 */
    bearbeite_datei (file) /* 37 */
}
```

Note 34: This function determines the first file from a set of files, for example the current directory.

Note 35: This is a symbolic constant that represents the event "No file found". This value is returned by the two functions when a file is (more) in the crowd.

Note 36: This function determines the next file from a set of files, for example the current directory.

Note 37: This feature finally makes something with the found file.

While Loop

C has two more loops, which are frequently used, a loop with one foot operated and loop. Let's start with the head-controlled loop:

```
# Include <stdio.h>

int main (void) {
    int i;

    i = 1;
    while (i <= 10) {
        printf ("%d \n", i);
        i ++;
    }

    return 0;
}
```

It is so called because its loop condition is at the beginning, so the head. If this condition is false right the first time, the corresponding statement block is not even running!

The example is the way completely equivalent to the example of for loop. So you can see even when which part of the for loop is executed. The initialization statement is executed exactly once as the very first. Before each iteration of the loop (even the first!) Is the loop condition calculated and evaluated. If the loop condition is true, the statement block is executed first. Only as a last resort, the loop is executed.

Do..While Loop

Now we come to Do...While loop. Here, the loop condition is checked only at the foot of the loop if the corresponding statement block was already passed:

```
# Include<stdio.h>

int main (void) {
    int number;

    do {
        char line [10];
```

```

        printf ("A NUMBER (0 = END):");
        gets (line);
        sscanf (line, "% d", & number);
        printf ("\ nThe% d OF DOUBLE IS% d \ n", number, 2 * number);
    } While (num = 0!)

    return 0;
}

```

So far, so good! C has two key words that change the course of a loop:

- "Break" ends a loop once in a for loop, the loop statement is no longer running.
- "Continue" aborts the execution of the statements in the statement block and continues with the loop condition. In the for loop, but before that the loop is executed.

An example will certainly help in understanding. Please try to predict what the program prints before you try it:

```

# Include <stdio.h>

int main (void) {
    int i;

    for (i = 0; i <200; i + +) {
        if (i <10) {
            continue;
        }
        printf ("IN:% d \ n", i);
        if (i> 20) {
            break;
        }
    }
    printf ("END WITH% d \ n", i);

    return 0;
}

```

The jump instruction goto

Yes yes, there are also in C jumps. Fortunately, however, the whole control structures such as branches and loops are almost always sufficient, so a jump almost never needs to be used. He actually destroys whatever the beautiful building program.

But anyway, we still want to watch again! Just where to go to the jump? C. he has no line numbers ... The answer: C has labels as the assembler freaks among you have been known:

```

# Include <stdio.h>

```

```

int main (void) {
    int i;

    i = 1;
MARKE1:
    printf ("%d \n", i);
    i ++;
    if (i <= 10) {
        goto MARKE1;
    }

    return 0;
}

```

Homework

- Writes the counting program with the do-while loop.
- What value does the variable i after the loop?

```

for (i = 30; i > 7; i -= 12) {
    i += 4;
}

```

- Writes the counting program so that it counts down. Tried it with all three variants loop (for, while, and do-while loop).
- Writing a program that calculates the sum of all the integers between 10 and 20 (both inclusive).
- What's wrong with the following program segment?

```

for (count = 0; count < 10; counter ++);
    printf ("COUNTER = %d \n", count);

```

- What's wrong with the following program segment?

```

Index = 1;
while (index <= 10)
    printf ("INDEX = %d \n", index);

```

Part 13 – SLOW & FAST

In the same chapter number Vickers leads us into the secrets of "SLOW & FAST" a. To come straight to the point: I have not managed within a C program the mode switch back and forth. Which one of you the riff can even publish his method in the forum. Note: the two functions are called `zx_fast()` and `zx_slow()`. Instead, you get some special features offered, which were provided by the z88dk-makers

specifically for the ZX81. At HRG enters a later part.

Manipulation of the entire screen

Let's start with two functions that handle the entire screen. The whole screen? No, a small village ... um ... Incorrect text ... So, there are only 22 of the 24 lines, although expenditure on C use all 24 lines.

```
# Include <stdio.h>
# Include <zx81.h>

int main (void) {
    char input;

    puts ("HELLO, WORLD.");
    gets (& input) / * Please enter only NEWLINE! * /
    invtxt ();
    gets (& input) / * Please enter only NEWLINE! * /
    mirrortxt ();

    return 0;
}
```

This small example writes the first known text on the screen. After NEWLINE the output screen is inverted. After another NEWLINE the output screen is flipped horizontally. Whoever takes the ...

Conversions between ASCII and ZX81 character set

As you have already learned, there are two sets of characters in C: one for the source, who is with us ASCII, and secondly to the current program, which is for us the ZX81 character set. The conversion between the ASCII character that reads a C program and outputs, and the ZX81 character we can while running the program switch.

The following sample program shows in the first half off and on again the character set converter. When turned off, of course, the corresponding converter ZX81 codes are used, so I've used there, according to figures of the encoding table in Annex A of the ZX81 BASIC manual.

In the second half of single characters between the character sets are converted. Above all, the assembler freaks who write drivers for floppy or MMCs know, these conversions already.

```
# Include <stdio.h>
# Include <zx81.h>

int main (void) {
    puts ("Hi.");
    zx_asciimode (0);
    printf ("% c% c% c% c% c% c \n", 45, 38, 49, 49, 52, 27);
    zx_asciimode (1);
}
```

```

    puts ("Hi.");

    printf ("% x% d \ n", 'H', ascii_zx ('H'));
    printf ("% x% d \ n", zx_ascii (38), 38);
    printf ("% x% d \ n", 'L', ascii_zx ('L'));
    printf ("% x% d \ n", zx_ascii (52), 52);

    return 0;
}

```

Contact the BASIC program

Yes, you read that right: the C program can interact with the BASIC program. First, something simple:

```

# Include <stdio.h>
# Include <zx81.h>

int main (void) {
    printf ("BASIC BYTES =% d \ n", zx_basic_length ());
    printf ("BYTES VARIABLES =% d \ n", zx_var_length ());

    return 0;
}

```

The two functions give us the length of the BASIC program and the variable area. To try ye can not enter more BASIC lines and / or create BASIC variables to get different outputs.

Much more exciting I find the ability to execute individual BASIC lines (after compiling and loading you should enter a few simple lines with line numbers between 100 and 200):

```

# Include <stdio.h>
# Include <zx81.h>

int main (void) {
    int l;

    for (l = 100, l <= 200, l + = 10) {
        printf ("LINE% d:% d \ n", l, zx_line (l));
    }

    return 0;
}

```

Apparently, the error code of the BASIC version is returned as an int of `zx_line ()`. If the desired line number is not found, the BASIC interpreter looks as expected after the next higher line number. It is also performed in each case only one line. Attempts to call subroutines with GOSUB, were unfortunately not successful. Also here you can announce their success stories in your forum ...

We come to the exchange of values between BASIC and C. Unfortunately, the z88dk been "only" strings back and herkopieren, and in the process bringing a BASIC string to C must also have the last character to be deleted separately, which makes the selected row . That looks like an error in z88dk, perhaps, the corrected in a future version, yes. For other computers as the Zeddy the z88dk also provides functions for exchanging numeric variables are available only as a perspective.

```
# Include <stdio.h>
# Include <string.h>
# Include <zx81.h>

int main (void) {
    v char [20];

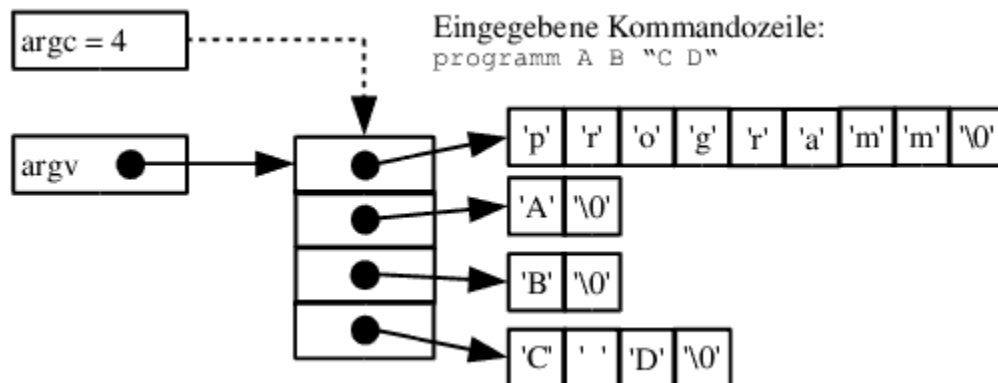
    printf ("GET% s", zx_getstr ('A', v));
    v [strlen (v) - 1] = '\0'; / delete * <--- the last character * /
    printf ("= \"% s \ " \" n", v);
    zx_setstr ('B', v);

    return 0;
}
```

After compiling and loading the program, you need a variable A \$ to a string create less than 20 characters, whether directly (not then start with RUN!) Or as a line of code before the call to the C program. If this variable is not in the running of the C program exists, you can cause the computer to crash ...

That would be nice ...

C actually stems from a time in which every computer user did not use a command line. Therefore, a normal C program can also arguments on the command line can be specified. Unfortunately, this is not the z88dk, although it would be a nice way yes. Nevertheless, I want to show the method for the case that once you write for other computers C programs.



Our main function takes two parameters to the traditional `argc` (argument count) and `argv` (argument vector). The first indicates how many arguments are present, while the second is a pointer (pointer we will discuss later ... The programmers know assembler pointer already: there are normal addresses!) Refer to an array of pointers to the strings of the arguments image . The first string is the name entered in the program accessed.

Disassembling the command line arguments in the individual concerned while the C library. To pass arguments to with spaces, you must put this in quotation marks, as in "CD" in the image. The sample program can compile indeed, documented, however, that the `z88dk` the two parameters has filled with meaningless values:

```
# Include <stdio.h>

int main (int argc, char * argv []) {
    printf ("%d ARGUMENTS \n", argc);
    if (argc > 0) {
        printf ("%s \n", * argv);
        ++ argv;
        argc--;
    }

    return 0;
}
```

As a normal PC program but it works as expected, a test showed with the GNU compiler. So, that's it for this part, the next time it comes to your own functions!

Part 14 - Subroutines

Before, we were limited to a "flat" program and were only allowed to call functions that other people have. This part deals with "subroutines", ie subroutines. Some languages distinguish between procedures and functions for which the former never return a value, but the latter always. In C all subroutines are called functions.

Yes because C has no line numbers, there is another, more natural way to identify a function. You can easily get a name. This name is finally called up. What, then, constitutes a function? For this purpose we look at the function that we write every time we write a C program:

```
int main (void) {
    puts ("HELLO WORLD.");

    return 0;
}
```

The "visual" interface, which is what needs to know a caller called footprint, because it is the footprint

of the function in a way. It consists of:

- Data type of the return value
- Name of function
- Argument list (each argument is a data type, and optionally a name)

To publicize a function to the compiler, without defining them entirely, we are given for each function to be called a declaration in the form of a prototype. These are e.g. in the header files like "stdio.h" included. For our main feature is the prototype:

```
int main (void);
```

The difference between this declaration to definition is that after the argument list in parentheses following a statement block in curly braces, but just a semicolon.

A simple example

Let us assume that we need in a program the sign function: it provides +1 if the argument is positive, 0 if the argument is zero, and -1 if the argument is negative. So that we can already write down the prototype, under the assumption that only integers are passed as arguments:

```
int sign (int);
```

Now we need only the implementation, but that's easy. When defining a function, we must of course give the arguments names, which we have omitted in the declaration yet:

```
int sign (int num) {  
    int result;  
  
    if (number <0) {  
        result = -1;  
    } Else if (number == 0) {  
        result = 0;  
    } Else {}  
        result = +1;  
    }  
  
    return result;  
}
```

Now we are expanding into a test program, because we are careful programmer:

```
# Include <stdio.h>
```

```
int sign (int);
```

```
int main (void) {
```

```

        printf ("sgn (%d) = %d \n", -999, sgn (-999));
        printf ("sgn (%d) = %d \n", -1, sgn (-1));
        printf ("sgn (%d) = %d \n", 0, sgn (0));
        printf ("sgn (%d) = %d \n", +1, sgn (+1));
        printf ("sgn (%d) = %d \n", +999, sgn (+999));

        return 0;
    }

    int sgn (int num) {
        int result;

        if (number < 0) {
            result = -1;
        } Else if (number == 0) {
            result = 0;
        } Else {}
            result = +1;
        }

        return result;
    }

```

As you can see, a function is called by their name with a pair of parentheses. In brackets the current parameters. If a function returns a value, it can be used in an expression.

The order in which it stores the individual function in the source code is basically the same. The compiler must know already the only footprints when he sees the calls. Many programmers, therefore at the beginning towards the prototypes, the main function, and then the other features, most exactly.

More details

Let us each point once a closer look. After splitting up a problem-solving in the right features (and data, but that is then almost "object-oriented" development) is the lifeblood of the program. Divide a program into functions is, follow the principle of 'divide and rule ". A well designed program reveals itself to the reader but self he must not look to the function definitions, to find out what make the called functions.

The return value of a function can e.g. directly be the desired result. If multiple values are returned, we need other mechanisms that we learn about in later parts. Otherwise, a function can return secondary information, such as printf () returns, for example, the number of output characters. Quite often is also the return of success or failure. If a function does not return anything, must be specified as the data type void.

```

void putchar (int) / * get a character that delivers nothing */
int getchar (void) / * gets nothing, provides a character */

```

The function name should be chosen so that their functionality is immediately recognizable. Many programming guidelines require that the first piece corresponds to a verb in the imperative. As a sign of the usual suspects are allowed, ie small and large letters, numbers (but not the first character), and the underscore. According to ISO, at least the first six characters are used to distinguish, most compilers use all characters. As always, big and small letters are different! Examples:

```
int v24_open (void);
void v24_close (void);
int v24_receive_character (void);
v24_send_character int (int);
```

When a function receives arguments, the prototype must be at least their data types can be specified. More namely is not interesting for the compiler, in the final program, the name no longer exist anyway. When defining a function's arguments must also be given names so they can be referenced in the statement block. When a function receives no parameters, it's good practice to write a void in the parentheses.

Some programming languages hae the passing arguments by value or by reference. It means by value, that of the actual argument, the value is computed and is passed as a copy to the function. In contrast, if passed an argument by reference (This is probably only with variables of the caller.) Is instead directly use the variable in the called function. Whose content can be changed!

In C, you need to make you not worry about it. All arguments are passed by value, and even you can look like local variables. The example below illustrates this:

```
# Include <stdio.h>

void function (int parameter) {
    printf ("START FUNCTION, PARAMETERS =% d \ n", parameter);
    parameters / = 2;
    printf ("END FUNCTION, PARAMETERS =% d \ n", parameter);
}

int main (void) {
    int variable;

    variable = 81;
    printf ("MAIN START VARIABLE =% d \ n", variable);
    function (variable);
    printf ("MAIN END VARIABLE =% d \ n", variable);

    return 0;
}
```

In this sample program, the sequence of main and the called function is reversed compared to the first example. Therefore, because it is already known when the function of the footprint, a line can be missing with the prototype.

The statements inside a function implement the desired functionality. They are for the caller, so the user of the function actually uninteresting, the main thing they do the right thing (TM).

Calling a function is carried out as described by specifying the name and a pair of parentheses. The number of arguments and their types must match with the prototype, where C specific type conversions (for example, all integer numbers are interconvertible, of course, undesirable side effects, such as the number of overflows.) And then allowed himself to perform. If a function returns a value, it must not necessarily be used. It can also be safely ignored, with consequences. ;-)

A function can have only one "input" because it has nothing but a name that can be specified when calling. However, a function can always be left with return. If the function returns a value, it is written down immediately after the return, there are also complex expressions possible. The sign function can also be written like this:

```
int sgn (int num) {
    if (number <0) {
        return -1;
    } Else if (number == 0) {
        return 0;
    } Else {}
        return +1;
    }
}
```

Unlike BASIC no precautions are necessary so that a main program not in a function "in progress". For each of the outermost pair of braces limited the scope of the function.

Homework

- Writes the prototype of a function with three char-arguments that returns an int. You should "tu_es" hot.
- What is the prototype for a function called "print", which receives an integer as an argument and returns nothing?
- What is wrong with this code? Tip: There are several mistakes ... Corrects the program.

```
# Include <stdio.h>

void print_msg (void);
int main (void) {
    print_msg ("The text to be");

    return 0;
}

void print_msg (void) {
    puts ("issued TEXT");
}
```

```

        return 0;
    }

```

- Writes a function that gets two integers as parameter and divides the first by the second. The ratio should be returned. Ensures that no division by 0, and makes you think about which results should then be returned.
- Wrote a test program for the function of Problem 4

Part 15 – Making Your Programs Work

It is interesting that in Vickers' manual only now the chapter "Making your programs work" comes. Because is not that the goal of all our efforts from the beginning? Anyway, this time it's about flow charts and troubleshooting. We are going to miss the flowcharts, because we know you already know ...

The easiest way to debug a program, insert at appropriate locations expenditure with the values of important variables. The first is completely independent of the programming language, even when programming the microcontroller can be tiny wiggle one or the other port pin in certain rhythms. In C, we do of course with `printf()`.

For an example let's take another new tax structure that knows it not, the multi-branch switch case. The following program will ask the user a choice and then work accordingly, but somehow it does not work as desired:

```

#include <stdio.h>

int main (void) {
    int result;

    result = 0;
    for (; ;) {
        char input [10];
        int value;

        puts ("PLEASE CHOOSE:");
        puts ("END.");
        puts ("RESULT = OUTPUT");
        puts (" + NUMBER VALUE FOR ADDITION ENTER");
        puts ("NUMBER VALUE FOR ENTERING SUBTRACTION");
        gets (input);
        puts ("");

        sscanf (input + 1, "%d", &val);

        switch (input [0]) {

```

```

        case '.':
            return 0;
        case '=':
            printf ("RESULT =% d \ n", result);
        case '+':
            result + = value;
        case '-':
            result - = value;
    }
}

```

Before we go to the troubleshooting, first a brief explanation of switch-case. After the first key word in parentheses is an integer expression is specified. Here we can choose a character, because even characters are encoded in integers. After the parenthesis is a block with braces. This can distinguish different cases of the term, but here are only allowed constants (integers or single characters). For example, the following works:

```

switch (int_var * 3 - 1) {
case 12:
    /* Statements if the expression is equal to 12. */
case 3:
    /* Statements if the expression is 3. */
case 81:
    /* Statements if the expression is equal to the 81st */
}

```

Between two's case may not necessarily be instructions.

What makes our program now wrong? First of all, everything looks right, but we can not add value. The result does not change! So we build a first edition that tells us the current state of the result:

```

...
printf ("DATE =% d \ n", result);
puts ("PLEASE CHOOSE:");
...

```

Well, that helps us do not, but only confirms the observations. Maybe we should check for any change in the result of the contents of the variables?

```

...
case '+':
    result + = value;
    printf ("DATE =% d \ n", result);
case '-':
    result - = value;

```

```
printf ("DATE =% d \ n", result);
```

...

Ah yes, if we now want to add a value, we get two copies! And if we can output the result, we will also appear twice our test output. It seems as if though the switch corresponding to the first character of the input to the beginning of the instructions properly selected, but all statements are executed after ...

And that is indeed the case - and it is so true. The multiple branching to work so, so eg lead to the following example more selections to the same instructions:

```
switch (sign) {  
case 'a':  
case 'e':  
case 'i':  
case 'o':  
case 'u':  
    /* Statements for the vowels */  
...  
}
```

If we can not prevent, "the flow of the program falls through" to the next branch. The prevention is done with a keyword that we already know from the loops: break.

Multiple branching knows yet another keyword which initiates a branch in the event that no other branch fits. This keyword is default, it's good, it must always be specified, even if nothing is running! An example:

Code: Select all

```
switch (menu) {  
case 1:  
    /* Statements for the first item */  
    break;  
case 2:  
    /* Statements for the second menu item */  
    break;  
case 3:  
    /* Statements for the third menu item */  
    break;  
default:  
    break;  
}
```

Homework

- Corrected the example program so that it works correctly.
- Extended the sample program to a default case, the answer to a wrong input to the output of

the help text. Then the recurrent expenditure within the loop is no longer necessary.

Part 16 – Tape Storage

Chapter 16 is in the BASIC book "Tape storage" ... but there are no ready z88dk functions that save and load or. Therefore, we will discuss in this part again on another.

Uninitialized variables

So far we have always applied only variables. Depending on the location of the definition then they are automatically initialized to 0 (zero) or have a random content. To a variable to be a specific value, we wrote a corresponding statement in the program text.

That's not bad at first, but there is another way. We may be the same when you create a variable, assign it a value:

```
int value = 81;
int primes [10] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 27};
char text1 [100] {' ' 'H', 'A', 'L', 'L', 'E', '\0'};
char text2 [100] = "HELLO.";
```

As you can see, we can also initialize arrays with values by the values in braces separated by commas perform. The last two lines show that for character arrays (Commonly referred to as "strings" means ;-)), where C is allowed even the short form with a string constant. These last two lines are equal, only the second option is much more readable.

For arrays, there is even a convenient automatic: we need not specify the size when we set it on the initialization. That you can try the following short program:

```
# Include <stdio.h>

int values [] = {1, 4, 9, 16, 25, 36};
char character [] = "HELLO, ZX81.";

int main (void) {
    int count;
    int index;

    count = sizeof (value) / sizeof (int);
    printf ("VALUES [] CONTAINS ELEMENTS% d \n", count);
    for (index = 0; index <count; index + +) {
        printf ("VALUES (% d) =% d \n", index values, [index]);
    }

    count = sizeof (character) / sizeof (char);
```



```

    printf ("MARK [] CONTAINS ELEMENTS% d \ n", count);
    for (index = 0; index <count; index + +) {
        printf ("SIGN (% d) = '% c' (% d) \ n", index, symbol [index], sign [index]);
    }

    return 0;
}

```

The z88dk had problems with the first draft of this program:

- The two arrays initialized should be locally in main (). That may be the z88dk not, too bad.
- The calculation of the number of elements in an array is calculated professionally as follows:

No. of values = sizeof / sizeof values [0]. As you can see, it displeased the z88dk not, therefore, used the top variant.

The advantages of this type of initialization and the use of sizeof is that we do not need to determine the size of the array itself and the rest of the program automatically cope with it. In the example, we have stored the number of elements in a variable yes, that's not necessary, if we calculate at the appropriate point directly enter. If the compiler is good (. Indeed z88dk the compiler has generated the comparison with the constant "6"), he also notes that the number is a constant and simple machine code generated according to:

```

# Include <stdio.h>

int values [] = {1, 4, 9, 16, 25, 36};

int main (void) {
    int sum;
    int index;

    sum = 0;
    for (index = 0; index <sizeof (values) / sizeof (int) index + +) {
        sum + = values [index];
    }
    printf ("TOTAL =% d \ n", sum);

    return 0;
}

```

When the variables are initialized for now filled with their values? That depends on whether static or dynamic (see next section) are: static variables are directly applied to their initial values in the machine code. For dynamic variables, the compiler generates the relevant instructions; that's for sure why the compiler of z88dk can not: it complicates the compiler.

Modifiers and Storage Classes

auto and static

So far we have seen two different types of variables: static variables and dynamic variables. This is part 6 of the treated, and if you no longer remember, it looks at you again.

If you angebt neither "auto" or "static", are global variables that are outside of a function that automatically static and local variables declared within a function (maybe even within an inner statement block), are automatically dynamically. As part 6 showed you can make local variables static by their definition of a "static" front of them.

By contrast, global variables can not be made dynamic, because they exist anyway during the entire duration of the program. Nevertheless, there is the keyword "auto" for but it can be written to local variables. Due to the definition in the ISO standard, but it is redundant:

```
void function (void) {
    auto int variable;
}
```

register

The assembler freaks among us will look at certainly one or the other time in the generated machine code and find that the compiler some variables not in memory creates, but for processor registers used. This of course is great, because the term is often shorter in. The variables that are now placed in processor registers, is the secret of the compiler, these are mostly the local variables of the innermost blocks and those that are frequently used. Sometimes, the compiler decides not the way we want the programmer. For this case we can give a hint to the compiler, by the variable definition, a "register" in front. But this is really just a tip, the compiler need not necessarily stick to it! The following example may indeed the register pair HL used for "quick", but because the machine code generated by this constant value in the stack stores and pulls out again, there is no real advantage over the static variables. As we note again that the compiler generates the z88dk not really great machine code.

```
# Include <stdio.h>

int main (void) {
    static int static;
    register int quickly;

    puts ("START");
    for (static = 0; static <10000; static + +) {
        /* Do nothing */
    }
    puts ("BETWEEN");
    for (fast = 0; fast <10000; fast + +) {
        /* Do nothing */
    }
}
```

```

    puts ("END");

    return 0;
}

```

volatile

Another, now and needed more modifier is the key word "volatile". It features a variable as "volatile", ie the compiler can optimize away not read and write requests. Good optimizing compilers provide common ground when it several times in succession to access the same variable, and then optimize the machine code so that the values are cached optimally. This is not necessarily the location of the variables ... We are concerned with the z88dk not because the compiler knows the key word though, but it ignored but even generates therefore a warning.

When this modifier is needed? For example, when programming with interrupts or multithreading and both the main program and the interrupt functions use the same variable.

An example:

```

volatile int interrupt_kam;

void interrupt (void) {
    interrupt_kam = 1;
}

int main (void) {
    interrupt_kam = 0;
    while (interrupt_kam == 0) {
    }

    return 0;
}

```

const

The last presented here modifier is "const". Unfortunately, this is the z88dk recognized but (thankfully with a warning) ignored.

With "const" variables are marked as read-only. The compiler will then report an error when write access to such a variable. If the generated program to a corresponding operating system such as Linux or a Windows NT-generation (not the Win95 generation!) Is running, even throw an error at runtime.

Mark a variable as read-only, for example useful if a table is set up so that should not be changed.

An example:

```

const int primes [] = {2, 3, 5, 7, 11, 13, 17, 19, 23};

```

Homework

- To calculate the number of elements in an array in the first example program: you considered the disadvantage of the method used to the "professional" way.

Part 17 – Printing with Frills

In this section we come back to practical matters, after the last part was more of a theoretical nature. When Vickers is the chapter "Printing with frills", and so it's also something of outputting with all the trimmings.

While rendering, we must distinguish two things:

- In what way are converted using `printf ()` expressions in strings. Mainly that's what this part.
- How and where to land the output characters on the screen. That's true with Vickers, but that are beyond the rudimentary, really deep that will concern us in Part 20, I promise!

Positioning on the screen

New Line: If you use the `PRINT` command of BASIC, automatically at the end of a line feed is performed, unless you give a semicolon as the last character. In C there is not this happen automatically, if we neglect `puts ()`, which actually attaches itself a `'\n'`. But this is the only exception! Instead, we must be where we want to have a line break, explicitly specify a `'\n'`.

Beginning of the line: Actually should `'\r'` just cause a carriage return, so put the cursor at the beginning of the current line. But the `z88dk` functions thus creating a new row as with `'\n'`.

Scrolling: The output routines from `z88dk` also scroll the screen automatically when the last line is full or there, a newline character is output. This is handy, especially after it is the usual behavior of terminals. There are text-mode also not a function or control characters to recreate the BASIC command `SCROLL`. By the way, uses the `z88dk` all 24 lines!

Clear screen: But there is a control character corresponding to the BASIC command `CLS`: `'\f'`. We had all the special characters to part 11, there was the sign already declared with "form feed", ie feed. Normally in a printer is requested to eject the current page and to start a new one. And then also makes the screen - and discards its contents. ;-)

So, the whole thing can understand her with this little test program:

```
# Include <stdio.h>

int main (void) {
    int screen;
```

```

        for (screen = 1; umbrella <= 5; umbrella + +) {
            int line;

            printf ("\ fSCHIRM% d \ n", screen);
            for (row = 1; row <= 40; row + +) {
                printf ("\ tZEILE% d \ n", line);
            }
        }
        return 0;
    }
}

```

Free positioning: In text mode, we have no way to replicate the BASIC function AT.

Tab: Even something like TAB, there is unfortunately not the only failure is the z88dk-makers. 'T \' because there is the control character that just is there ...

Formatting

So far we have used only simple printf () output formats. But the function is very powerful, with all the features it can be very large, I have heard from more than 10 KB machine code. The z88dk has three different versions:

- The mini version is just the simple formats for integers and strings, without frills.
- The complex version mastered some formatting options that are presented in this chapter, such as width and alignment. The documentation for it is but can not issue large integers (long), and the experiments show fortunately something else.
- And then there's the version for floating point numbers. Unfortunately, this version does not seem to exist for the ZX81 ...

The compiler decides on the basis of the source code, which version should be used. Let's hope he succeeds always right! Here's an overview of what can be spent on data types, some of which you already know:

Format string data type =

- % C = int as unsigned char, is interpreted as a sign
- % D = signed int, decimal output
- % U = unsigned int, decimal output
- % X = unsigned int, is output in hexadecimal
- As =% ld% d, only signed long
- As u =% lu%, only unsigned long
- % As% lx = x, only unsigned long
- % F = float, is not the z88dk
- =% S float, is not the z88dk
- % S = char array
- % P = address pointer, is not the z88dk
- %% = A percent is spent

Between the percent sign and the actual format characters can now be even more further information. These are used for finer control, the following list shows the usual understood by printf () options:

- A plus sign '+' calls for the issuance of a sign and positive values. This may be the printf () from z88dk not ...
- A minus sign '-' provides a left-justification, see also the next point.
- A number (even more digits) without a leading zero indicates the number of characters that are least used. Output is normally right-aligned, then left to have spaces.
- A zero (before the specified width) calls that can be used to fill the spaces instead zeros. This is handy for tabular output.
- A number after the decimal point has different meanings depending on its type. For floating point numbers so that the number of decimal places is determined. For string is thereby increased to the maximum number of characters specified.

The next example program shows some examples, and leads also to try as you can, making what format specification. Of course, this is not an exhaustive example, here is a lot of room to experiment. Even the professionals try, if they have some ideas ...

```
# Include <stdio.h>

int main (void) {
    printf ("c =% d \"% c \" \" n", 65, 65);

    printf ("% d of d =% d \" n", +81, +81);
    printf ("% d u =% u \" n", +81, +81);
    printf ("x =% d% \" n", +81, +81);
    printf ("% d of d =% d \" n", -81, -81);
    printf ("% d u =% u \" n", -81, -81);
    printf ("x =% d% \" n", -81, -81);

    printf ("% ld ld =% ld \" n", +123456, +123456);
    printf ("lu% ld =% d \" n", +123456, +123456);
    printf ("lx% ld =% lx \" n", +123456, +123456);
    printf ("% ld ld =% ld \" n", -123456, -123456);
    printf ("lu% ld =% d \" n", -123456, -123456);
    printf ("lx% ld =% lx \" n", -123456, -123456);

    printf ("% d 4d = <% 4d> \" n", +81, +81);
    printf ("% d = 04d <% 04d> \" n", +81, +81);
    printf ("% d +4 d = <% +4 d> \" n", +81, +81);
    printf ("-d of% 4d = <% 4d> \" n", +81, +81);
    printf ("% d 4d = <% 4d> \" n", -81, -81);
    printf ("% d = 04d <% 04d> \" n", -81, -81);
    printf ("% d +4 d = <% +4 d> \" n", -81, -81);
    printf ("-d of% 4d = <% 4d> \" n", -81, -81);

    return 0;
}
```

```
}
```

Possible issues of strings shown in the next example. For the full enjoyment we recommend the parallel viewing of source code and the actual issue!

```
# Include <stdio.h>
```

```
int main (void) {
    printf ("s from \"% s \ " = \"% s \ " \ n", "ZX", "ZX");
    printf ("s from \"% s \ " = \"% s \ " \ n", "HELLO", "HELLO");
    printf ("s from \"% s \ " = \"% s \ " \ n", "LONG WORD", "LONG WORD");

    printf ("4.6s from \"% s \ " = \"% 4.6s \ " \ n", "ZX", "ZX");
    printf ("4.6s from \"% s \ " = \"% 4.6s \ " \ n", "HELLO", "HELLO");
    printf ("4.6s from \"% s \ " = \"% 4.6s \ " \ n", "LONG WORD", "LONG WORD");

    printf ("6s of \"% s \ " = \"% 6s \ " \ n", "ZX", "ZX");
    printf ("6s of \"% s \ " = \"% 6s \ " \ n", "HELLO", "HELLO");
    printf ("6s of \"% s \ " = \"% 6s \ " \ n", "LONG WORD", "LONG WORD");

    printf ("-6s from \"% s \ " = \"%-6s \ " \ n", "ZX", "ZX");
    printf ("-6s from \"% s \ " = \"%-6s \ " \ n", "HELLO", "HELLO");
    printf ("-6s from \"% s \ " = \"%-6s \ " \ n", "LONG WORD", "LONG WORD");

    printf ("06s of \"% s \ " = \"% 06s \ " \ n", "ZX", "ZX");
    printf ("06s of \"% s \ " = \"% 06s \ " \ n", "HELLO", "HELLO");
    printf ("06s of \"% s \ " = \"% 06s \ " \ n", "LONG WORD", "LONG WORD");

    return 0;
}
```

Refactor

The last example can also be simplified good. The Professional's called "refactoring", or "rebuilding". A very good principle in the development of, for example, the DRY principle: Do not Repeat Yourself! The repeated statement of very similar strings and the same parameters simply begging to be saved. Because the z88dk unfortunately dominated not multidimensional arrays, I have used various sub-programs:

```
# Include <stdio.h>
```

```
void gib_aus (char * type, char * text) {
    char format [30];

    sprintf (format, "% s from \"%% s \ " = \"%% s \ " \ n", art, art);
    printf (format, text, text);
}
```

```

void teste_texte (char * type) {
    gib_aus (art, "ZX");
    gib_aus (art, "HELLO");
    gib_aus (art, "LONG WORD");
}

int main (void) {
    printf ("%31s \n", "WITH SUBROUTINE") / * <- see text */
    teste_texte ("s");
    teste_texte ("4.6s");
    teste_texte ("6s");
    teste_texte ("6s");
    teste_texte ("06s");

    return 0;
}

```

Several points are of interest in the program and you should definitely check it precisely:

- In order that the right printf () version is used, the selected output had to be inserted.
- There is not only the printf () function, which (on the screen actually writing printf () in the so-called standard output channel. This is by default the screen ... There is also the fprintf () function, which in a "stream" such as can write to a file.) writes, but also a function sprintf () that writes to a string. This is specified as the first parameter before the format string.
- The space used for the output on the screen format string is not generated until the duration of the program. Therefore, the above-mentioned sprintf () function is used.
- printf () and comrades do not necessarily get a string constant as a format specification. Just as well we can also use the contents of a variable.
- The sizes of the original program and the improved program are hardly different. We can conclude that the constant strings occur only once in the binary code. And a look into this confirms the conjecture that the compiler has optimized beautiful!

Let's look at the matter with sprintf () once more in detail:

```

sprintf (format, "% s from \"% s \ " = \"% % % s \ " \ n", art, art);

```

The first parameter is the target output, the variable is therefore only "format" because it is the format string for the next call to printf ().

The second parameter is the format string for the call to sprintf (), except it contains a constant text three format specifications, actually four! The first ("% s") takes the third parameter ("art") and outputs it as a string. The second ("% s %") generates just "% s", a double percent sign is yes issue a single percent sign. The third ("% % % s") initially produces a percent sign and then takes the fourth parameter (again "art") and outputs it as a string.

If we, for example, as "art", the string "4.6s" passed, the format string "of 4.6s \"% s \ " = \"% 4.6s \ \"

n" is created. If you is not entirely clear, the program adds to quiet all the necessary debug statements, see Part 15

Graphic character of the ZX81

As promised, we return to the special character of the ZX81. Of course, there are no equivalents in ASCII, so we need to turn off the output character set converters (see Part 13). The individual characters we write for example then with \ x ... described in the string, as in section 11. The next program is from the 16 possible low-resolution character that produces "11" quotes for a nice form:

```
# Include <stdio.h>
# Include <zx81.h>

int main (void) {
    zx_asciimode (0);
    printf ("% c% c \ x01 \ x02 \ x03 \ x04 \ x05 \ x06 \ x07% c \ n", 11, 0x00, 11);
    printf ("% c \ x80 \ x81 \ x82 \ x83 \ x84 \ x85 \ x86 \ x87% c \ n", 11, 11);

    return 0;
}
```

Floating-point numbers sometimes go yet?

The more we z88dk with the fiddle, the more we discover: some appear to work but the floating point numbers. Unfortunately, as I said is the matching printf () function is not available, therefore, the following sample his home-built output function. Thus, the program is also generated without error, you have to specify the option "-LM81" in the command line, so that the floating-point library is mitgelinkt. Then you can use the data type "float" (with about 9 points) and "double" (no difference to "float").

```
# Include <math.h>
# Include <stdio.h>

void print_float (char * format, float value) {
    int digits before;
    int decimal;

    if (value <0.0) {
        printf ("-");
        value = value;
    }
    integer += (int) value;
    Decimal = (int) ((value - integer +) * 10000.0);
    printf (format digits integer, decimal);
}

int main (void) {
```

```

float angle;

printf ("% d% 04d PI IS \ n", 3, 1416) / * for printf ()-selection * /
for (angle = 0.0; angle <= 2.0 * 3.141593; angle + = 0.31415926) {
    print_float ("sin (% 04d% d) =", angle);
    print_float ("% d% 04d \ n", sin (angle));
}

return 0;
}

```

Too bad that the whole thing is rather unstable. The program crashes same with the first output line. What a pity! But possibly even work other programs that you write with floating point numbers.

Homework

- Why can the ZX81 character with the code 0 (zero), not directly as "... \ x00 \ x01 \ x02 ..." to write to the string?
- Write a program that creates a table right justified in the first column the numbers 1 to 20 outputs and left-aligned in the second column of the squares.

Part 18- Graphics

This part is really exciting! For as in the BASIC book says "Graphics"! In this part, it will 'give again no new things to C, but we introduce you to the HRG libraries z88dk. With these collections of subroutines can program their MRO applications. The libraries are documented in Wikiwiki (For example <http://www.z88dk.org/wiki/doku.php/library:monographics>) of z88dk, but also the header files (These are graphics.h gray.h and, as in the To see sample programs.) give enlightening insight. ;-)

The z88dk us offers several display modes that are selected by the option "-startup" on the command line. We use already for quite some time "-startup = 2" to make work the C program in SLOW. With "startup = 1" we would not see anything because the C program works in FAST. This mode is obviously the default. The modes 3 to 7 are now HRG modes:

- "Startup = 3" uses 256×192 pixels, the program ends SPACE must be pressed in order to get back to the text mode. This is also convenient for our test programs.
- "Startup = 4" used 256×192 pixels, but when you return to BASIC, the text mode is switched automatically.
- "Startup = 5" is like "startup = 3", but produces a mode with 256×64 pixels, which makes MUCH time for the run.
- "Startup = 6" is like "startup = 4", but also with 256×64 pixels.
- "Startup = 7" generates a 256×64 pixel screen that it has 4 different shades of gray. When you return to the BASIC text mode is automatically turned on again.

Thus, when linking the HRG functions to be linked, nor the appropriate library must match the option

"lgfx81hr192" (at 192 pixels high) or "lgfx81hr64" are given (at 64 pixels high).

Graphics mode 256 × 192

To test this mode, we used the following simple example program:

```
# Include <graphics.h>
# Include <stdio.h>
# Include <zx81.h>

# Define WIDTH 256
# Define HEIGHT 192

int main (void) {
    int x;
    int y;

    printf ("\n% d GRAPHICS WITH PIXELS X% d \n", WIDTH, HEIGHT);
    clg ();
    copytxt (txt_or);

    drawb (0, 0, WIDTH, HEIGHT);

    for (x = 0; x <WIDTH, x ++ ) {
        y = (x * HEIGHT) / WIDTH;
        plot (x, y);
        plot (x, HEIGHT - 1 - y);
    }

    return 0;
}
```

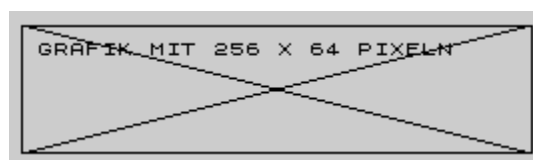
It clears the HRG-screen, copy the text into the text screen, then paints a rectangle outermost plots and finally the two diagonals. The output will look like in the next mode, just hold 192 pixels high ...

Graphics mode 256 × 64

To test this mode, we used the same test program, only the image height was adjusted:

```
# Define HEIGHT 64
```

And so it looks on the screen:



Graphics mode 256 × 64 in Grayscale

In this mode Bodo was particularly excited, as it has done something similar in his demo PUZZLE99. And on the FPGA prototype it also looks pretty good, in contrast to the representation in emulators (Can EightyOne now anyway? Please reply in the forum ...). In the gray areas likely tube screens flicker clearly!

This is the test program for this mode, it shows that the gray variants of HRG features a prefix "g_" possess:

```
# Include <gray.h>
# Include <stdio.h>
# Include <zx81.h>

# Define WIDTH 256
# Define HEIGHT 64
# Define LEVELS (G_white + 1)
# Define across 2

int main (void) {
    int g;
    int p;
    int x;
    int y;
    char input;

    for (g = 0, g <LEVELS, p ++ ) {
        for (x = (g * WIDTH) / LEVELS;
             x <((g + 1) * WIDTH) / LEVELS;
             x ++ ) {
            g_draw (x, 0, x, HEIGHT - 1, g);
        }
    }

    printf ("\n% d% d GRAYSCALE ON PAGES \n"
            "WITH% d% d PIXELS X \n", LEVELS, PAGES, WIDTH, HEIGHT);
    for (p = 0, p <PAGES, p ++ ) {
        g_page (p);
        copytxt (txt_xor);
    }

    g = 0;
    for (x = 0; x <WIDTH, x ++ ) {
        g_plot (x, 0, g);
        g_plot (x HEIGHT - 1, g);
    }
}
```

```

        g+ +;
        if (g> = LEVELS) {
            g = 0;
        }
    }
    for (y = 0, y <HEIGHT; y + +) {
        g_plot (0, y, g);
        g_plot (WIDTH - 1, y, g);
        g+ +;
        if (g> = LEVELS) {
            g = 0;
        }
    }

    for (x = 0; x <WIDTH, x + +) {
        y = (x * HEIGHT) / WIDTH;
        g = LEVELS - 1 - (LEVELS * x) / WIDTH;
        g_plot (x, y, g);
        g_plot (x HEIGHT - 1 - y, g);
    }

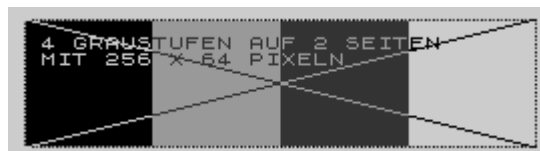
    gets (& input) / * only enter ENTER! */

    return 0;
}

```

The shades of gray are produced by appropriate switching of two HRG image storing. One is obviously used twice as often as the others, this results in the various intermediate stages. Unfortunately, the driver has a small flaw: the two "intermediate grays" are produced reversed ...

The test program revealed yet another problem: the function `g_draw ()`, as it is called there, actually begins only with the second pixel to draw. The screenshot has been reworked with the graphics program GIMP to show the shades of gray:



We are very excited to see what you come up with programs for this mode!

Examples of graphics features

HRG libraries are the most important functions. Some of them we have tried the following test program:

```
# Include <graphics.h>
```

```
# Include <stdio.h>
```

```
int x [] = {  
    0, 32, 65, 100, 100, 85, 120, 135, 130, 196, 141, 168,  
    168, 200, 250, 139, 108, 44, 60, 30, 0,  
    -1, 120};
```

```
int y [] = {  
    162, 130, 145, 145, 130, 114, 100, 128, 108, 108, 141, 141,  
    133, 126, 151, 170, 170, 165, 182, 190, 162,  
    -1, 144};
```

```
int main (void) {  
    int coordinate;  
    int size;  
    int index;  
  
    clg ();  
  
    draw (8, 8, 92, 8);  
    draw (92, 8, 92, 92);  
    draw (92, 92, 8, 92);  
    draw (8, 92, 8, 8);  
    coordinate = 10;  
    for (size = 81, size > 0, size - = 2) {  
        drawb (coordinate, coordinate size, size);  
        coordinate + +;  
    }  
    uncircle (50, 50, 20, 1);  
    uncircle (50, 50, 25, 2);  
    uncircle (50, 50, 30, 10);  
    uncircle (50, 50, 35, 90);  
  
    plot (108, 8);  
    DrawR (84, 0);  
    DrawR (0, 84);  
    DrawR (-84, 0);  
    DrawR (0, -84);  
    xorborder (110, 10, 81, 81);  
    xorborder (113, 13, 31, 31);  
    circle (150, 50, 20, 1);  
    circle (150, 50, 25, 2);  
    circle (150, 50, 30, 10);  
    circle (150, 50, 35, 90);  
  
    for (index = 1; x [index] >= 0; index + +) {  
        draw (x [index - 1], y [index - 1] x [index], y [index]);  
    }  
}
```

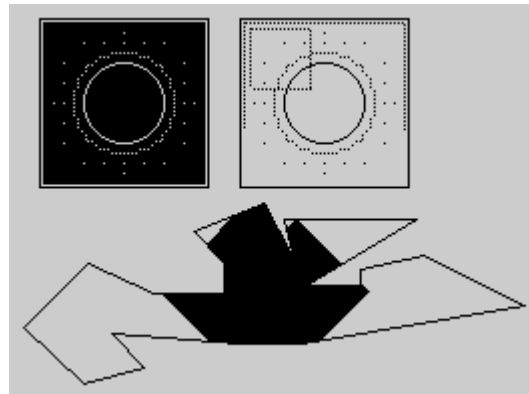
```

    fill (x [index + 1], y [index + 1]);

    return 0;
}

```

This program contains a typical construct again: initialized arrays. Anyone who has looked out of you have the bigger the ZX81 BASIC will have produced similar with DATA lines. The two arrays contain the coordinates for a pretty nasty polygon that is used to test the fill () function. It is very exciting to watch this feature!



The snapshot was taken during filling: you can imagine, is filled by the algorithm ... And we detect an error in the function xorborder (), which is distinguished from a certain size no longer the lower part.

ZX81 special function for HRG modes

The library for the ZX81 has functions for graphics modes. The following example examines the kinds of copying of a text screen that dominates the copytxt function ():

```

#include <graphics.h>
#include <stdio.h>
#include <zx81.h>

#define WIDTH 256
#define HEIGHT 64
#define DISTANCE 32

gibaus void (int row, int column, char * text) {
    printf ("\ f");
    if (line> 0) {
        printf ("\ n");
        line -;
    }
    while (column> 0) {
        printf ("");
        column -;
    }
}

```

```

        printf ("%s", text);
    }

kopieretext void (int line, char * text, int mode) {
    gibaus (line, 0, text);
    copytxt (txt_or);
    gibaus (line, 16, "0123456789ABCDEF");
    copytxt (mode);
}

int main (void) {
    int x;
    int y;

    clg ();
    for (x = WIDTH / 2, x < WIDTH, x++) {
        if ((x & (DISTANCE / 2)) == 0) {
            draw (x, 0, x, HEIGHT - 1);
        }
    }

    kopieretext (0, "TXT_AND" txt_and);
    kopieretext (1, "TXT_AND_CPL" txt_and_cpl);
    kopieretext (2, "TXT_OR" txt_or);
    kopieretext (3, "TXT_XOR" txt_xor);
    kopieretext (4, "TXT_OR_R" txt_or_r);
    kopieretext (5, "TXT_OR_L" txt_or_l);
    kopieretext (6, "TXT_AND_R" txt_and_r);
    kopieretext (7, "TXT_AND_L" txt_and_l);

    return 0;
}

```

The function always copies the entire text screen, but only where there are signs, some of them ever happened. To add still get the desired output, a few pull-ups are necessary. Trying to understand the source code in the. Experimentation is expressly desired! Again, the obligatory snapshot:



Other

There is still the functions `hrg_off ()` and `hrg_on ()`, but their investigation is left to the curious of you - we do not want you always chew everything!

Part 19 – Time & Motion

When Vickers is the chapter 19 "Time and Motion". We therefore want to deal here with functions which are used for user input and time recording. So that a part of the necessary functionality for games can be realized.

Because this is a course in C and not a course in game programming, yes, "only" a presentation of selected functions follows each with a sample program.

What's interesting for you budding C programmer it is: functions are abstracting. You are not as set in BASIC on a functionality, but you can achieve this by calling another, often self-penned feature a desired functionality. Thus allowing for example the below-described "real" keyboard driver the automatic repetition of keys.

Query on key pressed

Often, it is only important if the user has pressed any button, which button was which is, on the other hand do not care. There are blessed since DOS times the function kbhit (), which then returns "true" if a key was pressed. These are the header file "conio.h" are given:

```
# Include <conio.h>
# Include <stdio.h>

int main (void) {
    puts ("WAITING FOR BUTTON ...");
    if (kbhit ())
        puts ("DONE");

    return 0;
}
```

INKEY \$ "in C"

Of course, you all want to know how her beloved query can also be performed in C. The corresponding function is called when z88dk getk (). It returns the character '\0' (this usually indicates the end of a string, see Part 7), if no key is pressed, and the symbol on the key (the way that the space ... No more BREAK, yippee!) if one has been pressed. How INKEY \$ also blocked the call is not the function of the program run.

```
# Include

int main (void) {
    int key;

    puts ("END = Q");
```

```

do {
    key = getk ();
    if (key != '\0') {
        printf ("%02x key", key);
        while (getk () != '\0')
            puts ("release");
    }
} While (key != 'Q')

return 0;
}

```

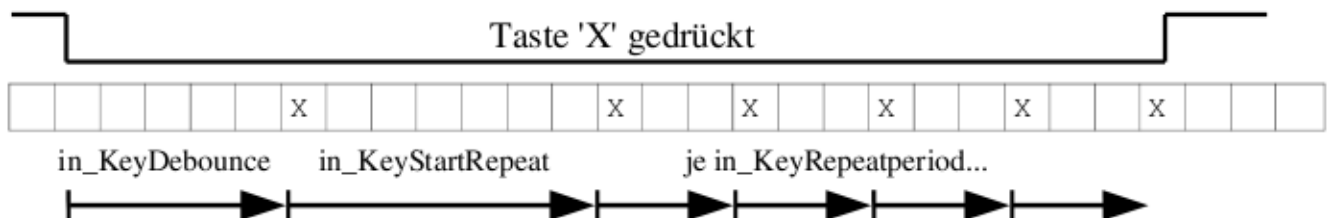
The z88dk contains a "real" keyboard driver

That was a surprise for us: z88dk has in the supplied libraries be a wonderful support for user input. The corresponding header file is named "input.h" from which we present here first enabled for keyboard scanning.

What else needs to be implemented with an optional hardware can actually also create software, see ASZMIC. This refers to the automatic repetition of keystrokes when the user holds down the button. There are two functions, `in_GetKeyReset ()` prepares the driver, and `in_GetKey ()` provides us with the next character from the keyboard. Furthermore, we need to create four global variables and fill with the following values:

- `in_KeyDebounce` contains the number of views `in_GetKey ()` before a key is accepted as a key. This value is used for debouncing.
- `in_KeyStartRepeat` contains the number of views `in_GetKey ()` before the automatic repetition in permanently holding down begins. This effect you can see also on your PC keyboard.
- `in_KeyRepeatperiod` contains the number of views `in_GetKey ()`, which lie between two auto-repeat keys. Therefore, the repetition rate is determined.
- `in_KdbState` is required for the internal management of the keyboard driver.

A small drawing shows the time response. In the box of the views `in_GetKey ()` are plotted, the 'X' marks the return of the character, otherwise '\0' is returned:



```

#include <input.h>
#include <stdio.h>

```

```

uchar in_KeyDebounce = 1;
uchar in_KeyStartRepeat = 255;
uchar in_KeyRepeatperiod = 5;
uint in_KbdState;

int main (void) {
    int key;

    in_GetKeyReset ();

    puts ("WITH AUTO-REPEAT (END = X):");
    do {
        key = in_GetKey ();
        if (key != '\0') {
            printf ("%02x key", key);
            while (in_GetKey () != '\0')
                puts ("release");
        }
    } While (key != 'X')

    puts ("WITHOUT AUTO REPEAT (END = Q):");
    do {
        key = in_Inkey ();
        if (key != '\0') {
            printf ("%02x key", key);
            while (in_Inkey () != '\0')
                puts ("release");
        }
    } While (key != 'Q')

    return 0;
}

```

If you indeed want to use this input method, but does not auto-repeat, you use the function `in_Inkey ()`. Why is the name of the well?

As you can see in the test program, the functions do not block the run, but they deliver as `getk ()`, the character `\0` if no button is pressed.

If the program is to wait

In `z88dk` there are two features that the program execution can be "slowed down". In `in_Pause ()` allows the user to interrupt the break, by pressing a button. `in_Wait ()`, however, is stubborn and ignored all the keystrokes. Both functions take the time in milliseconds be waited ...

```

# Include <input.h>
# Include <stdio.h>

```

```

int main (void) {
    int i;

    for (i = 0; i < 3; i++) {
        puts ("PAUSE ...");
        in_Pause (5000);
        while (in_Inkey () != '\0')
    }

    puts ("WAIT ...");
    in_Wait (5000);

    return 0;
}

```

Homework: Why is the line "while (in_Inkey () != '\0!)" After calling in_Pause ()?

Joystick emulation

PC games it's you accustomed that you can select and configure the input method. Similarly, the z88dk even if you use the functions for joysticks. This must invest their one variable of a specific type (eg "user_defined_key") and fill it with values determined. Then the function in_JoyKeyboard () will return a value indicating in its individual bits, which the user wants to reading:

```

Bit 7: Fire button
Bit 6: (nothing)
Bit 5: (nothing)
Bit 4: (nothing)
Bit 3: Right
Bit 2: Links
Bit 1: Down
Bit 0: Rauf

```

The values that are placed in the special variable must provides us the function in_LookupKey (). For professionals: this is the so-called scan code of the key ...

```

#include <input.h>
#include <stdio.h>

int main (void) {
    struct in_UDK user_defined_key;
    int joystick;

    user_defined_key.fire in_LookupKey = ("");
    user_defined_key.right in_LookupKey = ('8 ');
    user_defined_key.left in_LookupKey = ('5 ');
}

```

```

user_defined_key.down in_LookupKey = ('6 ');
user_defined_key.up in_LookupKey = ('7 ');

do {
    joystick = in_JoyKeyboard (& user_defined_key);
    printf ("%x \n", joystick);
} While (in_Inkey () = 'Q')

return 0;
}

```

Delays by default

Up ahead you have the delay functions of specially met z88dk. But there is also a function that (almost) every C system: `sleep ()` waits for the specified number of seconds.

For all the great tricks has z88dk even a function that you can the required number of processor cycles as a parameter. The minimum is 160!

```

# Include <stdio.h>
# Include <stdlib.h>

int main (void) {
    int i;

    printf ("SLEEP 5 SECONDS ...");
    sleep (5);
    printf ("OK \n.");
    printf ("CA 5000000 BEATS ... WAITING.");
    for (i = 0; i <100; i ++ ) {
        delay (50000) / * minimum of 160 * /
    }
    printf ("OK \n.");

    return 0;
}

```

Times take fully compatible

Since "time immemorial", there is a standard function that knows every C system. It is called `clock ()` and returns the number of clock cycles since the system started. Now you ask yourself why you want to be fully compatible, since each system is indeed have a different beat. This is also considered correct, but it defines the header file "time.h" a constant `CLOCKS_PER_SEC` that precisely reproduces this number of cycles per second. The small test program shows the application:

```

# Include <stdio.h>
# Include <time.h>

```

```

int main (void) {
    int second;

    printf ("%d \n", CLOCKS_PER_SEC);
    for (second = 0; second < 20; second++) {
        unsigned short start;
        unsigned short now;

        start = clock ();
        do {
            now = clock ();
        } While (now - start < CLOCKS_PER_SEC);
        printf ("EXPIRED %d \n", second + 1);
    }

    return 0;
}

```

When using z88dk you must still note one thing, otherwise you're looking for an error, which you have not done. The function `clock()` returns the content of system variable `FRAMES`, which is only 16 bits wide, as a 32-bit value. By appropriate programming of this value must be reduced to 16 bits, the sample program does it with a local variable, the appropriate width.

A summary example

Actually this should be fine watches from the BASIC program manual once represented in C. Unfortunately, the trigonometric functions do not work because of the lack of support for floating point numbers, apart from the increased complexity of graphics. Therefore, you can see here a program to determine the response time of the user:

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main (void) {
    int countdown;
    clock_t time;
    unsigned int seconds;
    unsigned int milli;

    puts ("RESPONSE-KNIFE");
    printf ("WHEN READY, PUSH BUTTON:");
    (Void) fgetc_cons ();

    printf ("\nNOTE");
    for (countdown = rand () % 10; countdown > 0; countdown--) {

```

```

        sleep (1);
        printf (".");
    }
    printf ("LOS");

    time = clock ();
    (Void) fgetc_cons ();
    time = clock () - time;

    seconds = (unsigned int) time / CLOCKS_PER_SEC;
    milli = (1000ul * (time CLOCKS_PER_SEC%)) / CLOCKS_PER_SEC;
    printf ("\nZEIT =% u% 03U SECONDS \n", seconds, millisecond);

    return 0;
}

```

Here is another new feature is used: `fgetc_cons ()`. This waits for a keystroke and returns the corresponding character. The name is derived from a standard function `fgetc ()` (file get character) and expanded to include the abbreviation "cons", probably for the "Console", ie screen / keyboard. The fantasy of z88dk-makers are no limits ...

With the cast (see section 6) to the data type void, we make it clear that we, the returned value is not interested. This is an example of self-explanatory source, because the machine code generated is not different from that which would be produced at less detailed source. So writes rather extensively, you will look forward to you about it later!

Notes for more chic functions

In the Wiki z88dk found many other functions that are useful for programming games. Particularly interesting sprites appear: these are small graphic objects (once defined) on and off and can be moved around the screen.

So, who wants to immerse themselves in the game programming should call the pages within the wiki and also study the sample programs there. This is bound to a field for wonderful, week-long experiments! Enjoy it!

Part 20 – The ZX81 Printer

And once again we have a section of abuse Vickers' Handbook are: "The ZX81 Printer". For, according to current knowledge, we can not print directly from C..

Print from C

Moment, it was just always another way? Well, with the findings of Part 13 so can we let the BASIC! The text to be printed is written to in a BASIC string variable, and then a BASIC line called with the

appropriate LPRINT command. "Oh, so it is!" Anyway, theoretically, please tell of your results in the forum.

Now we have the heading done enough ... ;-)

Varied edition - the VT / ANSI emulator

Oh dear, what does that mean again? Well, it is called the z88dk in the documentation, we want to introduce you today. Homosexuality is a "driver" for text output, and now here comes the explanation of the different abbreviations and terms:

- VT This is the acronym for Video Terminal, as part of the name of the manufacturer of computer terminals, DEC, in our case, the VT100.
- This ANSI stands for American National Standards Institute, the American equivalent of the DIN. In this case, however, the set of control characters is meant is at issue here.
- Our ZX81 emulator is not VT100 yes, but he will act as if he were one. And if a hardware and / or software mimics something, it's called emulation (<http://de.wikipedia.org/wiki/Emulator>).

The z88dk-makers were so friendly, in a variant of HRG driver also provide a text output, which recognizes certain control characters and accordingly draws text on the screen. In order not to reinvent the wheel, they have resorted to a known standard (<http://inwap.com/pdp10/ansicode.txt>), which was implemented on the VT100.

To give you an appetite, here is a small test program that demonstrates some of the capabilities:

```
# Include <graphics.h>
# Include <stdio.h>

int main (void) {
    clr ();

    puts ("\ 033 [10; 11H \ 2334mUnterstreichen goes \ 2330");
    puts ("\ 23320; 41H \ 033 [7mInvertieren goes \ 033 [0m.");

    return 0;
}
```

When you compile the program, you have to take the option "+ zx81" option "+ zx81ansi" use. And of course you must not "startup= 3" or forget another graphics mode, because it is the only emulator in HRG! The command line is therefore example:

```
zcc zx81ansi-vn +-Wall-create-app-startup = 3-lgfx81hr192 CfBASIC_20-1.c-o CfBASIC_20-1.bin
```

Before we forget: there is a set of characters with 4 pixels wide by 8 pixels high is used so that we can spend 64 characters in 24 lines. The character set can be changed without further not, according to the

documentation it comes z88dk but by re-compilation of the HRG library. This is just something for freaks!

First, we can now use all 96 characters in the ASCII character set. Here, upper and lower case letters, numbers, all the major characters, but no accents. By the following control characters are recognized correctly and executed:

Table: From the VT / ANSI emulator solved ASCII control character

<i>Kombination</i>	<i>Wert</i>	<i>ASCII</i>	<i>Bedeutung</i>	<i>Funktion auf z88dk</i>
<code>\n</code>	13	NL	<i>new line</i> , Zeilentrenner	Neue Zeile
<code>\t</code>	9	HT	<i>horizontal tabulator</i> , horizontaler Tabulator	Auf nächste 8er- Spalte
<code>\b</code>	8	BS	<i>backspace</i> , Rückschritt	Rückschritt
<code>\r</code>	10	CR	<i>carriage return</i> , Wagenrücklauf	Neue Zeile
<code>\f</code>	12	FF	<i>form feed</i> , Seitenvorschub	Bildschirm löschen

That was the easy part, now it's a little more difficult - but actually not. Somehow, we do have now (yes, yes, the emulator ...) the VT100 inform what we want from him. For this purpose a special character is used, the pretty name of the ASCII ESC (code 0x1B, decimal 27, responded to ESC stands for escape, that is, to escape German There is also a nice little story, an editor on the Atari ST.. pressing the ESC key with a box in which was written: "Why Escape, where?") bears. The German name for this is called escape character, and this character is a command to the VT100 initiated. The following characters are not so spent!

Interestingly, there are two alternatives for the initiation of a command:

- ESC and '[': With these two characters, the commands are usually represented in the documentation. This combination is suitable for data channels that carry only 7 bits of a character - yes, the whole thing is so old!
- 0x9B, decimal 155: This is the ESC character, but with set the eighth bit. On the ZX81 we can use this to save each command byte.

If you want to specify that character in C, we recommend the octal code `\ 233` (see Part 11, Table 1). You can write and as individual characters (`\ 233`) and as a character in a string (`"\ 233"`) use.

The actual command is the last character! As so often with "real" computers, a distinction is made between uppercase and lowercase letters. Between the command and the beginning of the command string parameter can be, often there are reasonable defaults. Of the many commands of the real VT100

emulator z88dk can only run the following commands:

- Command 'm' parameter type: set character attribute, the type is specified by a number:
 - 0 = normal (all attributes)
 - 4 = underlined
 - 7 = inverted
- Command 'A', the number of parameters: Move the cursor to number rows up, number is optional, the default is the first
- Command 'B', the number of parameters: Move the cursor to number lines down; number is optional, the default is the first Unfortunately, this command is implemented incorrectly, the cursor always ends up in the bottom line. :-(
- Command 'C', the number of parameters: Move the cursor to the right number of characters, number is optional, the default is 1
- Command 'D', the number of parameters: Move the cursor to number of characters to the left, number is optional, the default is the first
- Command 's': save current cursor position, so you can restore with 'u' can.
- Command 'u': the Restore's saved cursor position.
- Command 'H', parameter row, column: Move cursor to the row and column position. Caution, the line is numbered like 0-23, but the column! 1-64
- Command 'b'. Clears the top of the screen until the cursor Unfortunately the crash from Zeddy with this command ...
- Command 'Y' parameter range: Deletes a part of the screen area gives a point at which the field:
 - 0 = from the current cursor position to the end of the screen (Unfortunately, the z88dk here the mistake that every other line is deleted.)
 - 1 = from the top of the screen to the current cursor position (Unfortunately the Zeddy crashes from here also ...)
 - 2 = the whole screen (This is an easier way with '\ f'.)
- Command 'o': Deletes from the beginning of the line to the current cursor position, including.
- Command 'K' range of parameters: Deletes a part of the line where the cursor is, the field is a point at which area:
 - 0 = from the current cursor position to the end of the line
 - 1 = from the beginning of the line to the current cursor position
 - 2 = the whole line
- Command 'l', the number of parameters: number Deletes rows from the row where the cursor is. The lines below it are pushed up. Number is optional, the default is 1

We could really explain now the strings of the sample program in detail. But the learning curve is certainly greater with ...

Homework

- Explains the strings of the sample program. If you have trouble, you share the string into component characters and then tries to identify VT100 commands and actual character to output.
- Writes a program that lets you use the cursor keys ('5 to '8') a star (the '*') can be controlled via the screen. Doing so, the movement can be stopped at the edges by the program.

Part 21 - Substrings

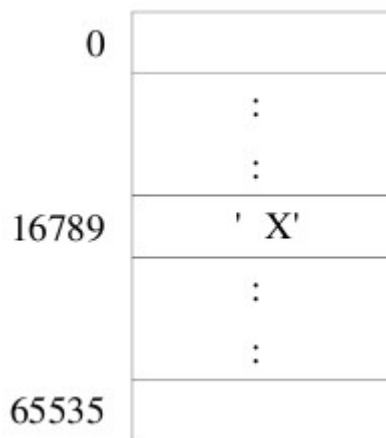
Today we finally (?) to the C concept, which is considered the most difficult. I (Bodo) do not believe this, but I also have a "machine language background." This concept is called (dangerous music in the background) ... HANDS!

Oh by the way, when Vickers is the chapter 21 "substring", just for the record. This fits quite well, because to edit strings can be used to advantage pointer.

What is a pointer?

This is simply the address of a value!

Hm, you is too simple? Good: The value of a pointer corresponds to the address in memory at which the data value is stored to which the pointer is pointing. The address of a byte is sort of the house number of a byte in memory. How, it is still unclear? Maybe this will help:



The numbers on the left indicate the address of bytes in memory. The box on the right are the corresponding bytes, of which only three are shown as examples. At the address 16789 is the sign 'X' is stored. A pointer to the 'X' then the value 16789th

Let's say you would set "value" are there, then would this state as caused by the following C-lines:

```
char value;  
value = 'X';
```

The address of this variable, we receive a value by writing the character '&' in front of the variable name:

```
printf ("ADDRESS OF VALUE =% u \ n", & value);
```

If you tried a program with this statement, certainly a different address is issued; said does not change the basic principle.

What pointers point

Pointers can point to any data type. There are also pointers to functions for which we give, but only the name of the function without the parentheses:

```
printf ("ADDRESS OF MAIN =% u \ n", main);
```

And finally there is the typeless pointer. This is an address to which a non-standard type is defined object. Which can therefore be anything that a data value, as well as the command of a function.

Pointer Variables

Of course, that's not much use to us. We would like to have a variable in which we can specify an address. The value of such a variable is thus the address of another variable (very subtle, the pointer can also point to themselves!).

So, we need a data type of "pointer to data type". This is e.g. for a sign with "char *" expressed. The asterisk marks "on hand ..." the property. An example?

```
char character;  
char * pointer;  
&zeichen; pointer = / * Now "pointer" points to "sign". * /
```

As stated above, pointers to any data type show, even on the "non-data type" void:

```
z1 int *, / * pointer to an int * /  
long * z2, / * Pointer to a long * /  
z3 void *, / * Pointer to a value of any type * /  
int ** z4, / * pointer to a pointer to an int * /
```

Because of the priorities of operators requires the definition of a pointer to a function of the right bracketing (See, however, the comments below the z88dk is limited here!.)

```
void (* z5) (void) / * pointer to a function with no arguments  
                    which returns nothing * /  
long (* z6) (int p) / * pointer to a function with an int  
                    Argument, which returns a long * /  
int * z7 (int p1, int p2);  
/ * This is, however, a prototype of a function called z7, the  
* Gets two int argument, and a pointer to an int  
* Returns!  
* /
```

As a "pointer" Incidentally, the value of the pointer variable is known, not the variable itself! If I change the value of the variable, I assign it to another pointer, but the variable itself is the variable.

Use values, pointing to the pointer

Now we can indeed create tags and store the address of other variables in it. As we are now able to access the value to which the pointer is pointing? Also cause the star is used, we speak of "dereferencing":

```
int number; /* Still, the value of "number" is undefined. */
int * pointer; /* Also "watch" shows anywhere (!). */

number = 81; /* OK, so has "paid" a defined value. */
&zahl; pointer = /* And "watch" now points to "three". */

/* Read access to "hand": */
printf ("* POINTER =% d \n", * pointer) /* Returns "81" from */.

/* Write access via "hand": */
* Pointer = 23; /* This is the value of "number" changed */

printf ("COUNT =% d \n", number) /* / Is "23" from */.
```

Numeric Literals

There are only a literal (A literal is a constant given explicitly, such as "23" for an integer.) Of type "pointer". This is "ZERO". Usually, but not always, this value is the number 0 This value can be marked invalid pointer values. Many systems report an error when a program wants to dereference such a pointer.

Arithmetic with pointers

With pointers we can expect also, but only perform useful operations:

- By the addition of an integer to a pointer, he is "raised".
- By subtracting an integer from a pointer it is "degraded".
- By subtracting two pointers (the same type!), We obtain an integer.

Now what does the integer to a pointer? It is the number of data values of the type to which the pointer points. This is very convenient because the addition of a "1" to show the pointer to the next value in memory can be. Assembler for you freaks: the integer is thus multiplied by the size of the data type before the addition or subtraction is the address.

You have already in part 7 arrays of characters created. In the following example, we use an array of ints:

```
int array [5];
int * pointer;
frame pointer = /* "pointer" points to the first element. */
pointer ++ /* "pointer" is now pointing to the second element. */
```

Actually, we would have liked better "& array [0]" written instead of "field", because that corresponds to the previously depicted. By experimenting with a "real" compiler you can see that "field", "field + 0" and "& array [0]" are equivalent expressions, the compiler of the z88dk can evaluate only the first term correctly, at least he is otherwise A warning from.

Conversion of pointers

In contrast to other data types, all pointers are always the same size, in bytes, for they are addresses of the main memory (the exception proves the rule: some compilers for microcontrollers such as the 8051-series actually know different types of pointers that are also of different sizes.). And thus may be a pointer to a data type easily converted to a pointer to another pointer type. This is done by a cast that knows her from Part 6:

```
char * zeichen_zeiger;  
int * ganzzahl_zeiger;  
  
ganzzahl_zeiger = (int *) zeichen_zeiger;
```

Quite "universal" of course the pointer type is "void *" because it the data type of the memory to which the pointer is pointing at all, determines.

Application in the string functions

As already mentioned in the beginning, pointers are particularly well suited to manipulate arrays. Arrays such as we learn in the next section to know, so we restrict ourselves to strings.

By BASIC we know the "slicing", the syntax with TO and the brackets. In C, this is not quite so simple, because we have to address itself to the space management. But beyond that there are fortunately finished functions in the standard library, you can use are:

```
# Include <stdio.h>  
# Include <string.h>  
  
int main (void) {  
    char * quell_string = "HELLO, WORLD.";  
    char ziel_string_1 [5 + 1];  
    char ziel_string_2 [4 + 1];  
  
    strncpy (ziel_string_1, quell_string + 0, 5);  
    ziel_string_1 [5] = '\0';  
    strncpy (ziel_string_2, quell_string + 7, 4);  
    ziel_string_2 [4] = '\0';  
    printf ("\"%s\" - \"%s\" \n", ziel_string_1, ziel_string_2);  
  
    return 0;  
}
```

In this program you can see in a typical application of a pointer: `quell_string` the variable is a pointer to a string. We do not know exactly where the compiler lying down or how many characters there are.

In the actual program, we use the `strncpy ()` function, the first parameter a pointer to the target, the second parameter the cursor gets to the source, and as the third parameter, the (maximum) number of characters to be copied. Because this function does not automatically target a string terminator attaches, we must make the following statement in itself.

The program is therefore equivalent to the following BASIC program:

```
10 Q $ = "HELLO WORLD".
20 Z1 = $ Q $ (1 TO 5)
30 Z2 = $ Q $ (8 TO 11)
40 PRINT "" "" ""; Z1 $; "" "-" "" , Z2 $; "" ""
```

At the end of this chapter I want to introduce more examples of three functions that are useful in the string processing. But there are some more, as a look in the header file "string.h" shows.

- `char * strchr (char *, char)`: The function searches the given string (first parameter) from the beginning of her after the specified character (second parameter). If it is found, it returns the pointer to it. If it is not found, it returns a NULL pointer.
- `char * strrchr (char *, char)`: The function works like `strchr ()`, except that from the end being sought.
- `char * strstr (char *, char *)`: This function works just like `strchr ()`, it only looks for the first occurrence of the string passed as the second parameter.

Calling functions via pointers

If we can define a pointer to a function, we have to go also can call the function to which the pointer points. And that is indeed possible, and it looks quite logical:

```
# Include <stdio.h>

void function (void) {
    puts ("A FUNCTION");
}

int main (void) {
    void (* pointer) ();

    puts ("BEFORE CALLING");
    pointer = function;
    pointer ();
    printf ("TO CALL");

    return 0;
}
```

Unfortunately, this may not actually be the correct definition of the pointer, which is called:

```
void (* pointer) (void);
```

Analogous to the usual REM line in BASIC, which contains a machine program is also available in C for such a possibility. For this we create an array of strings, in which we put our machines. In the example, it's easy just to load a value as the return value of a subroutine that is invoked:

```
# Include <stdio.h>

machine-static unsigned char code [] = {
    0x21, 0x34, 0x12, /* LD HL, 1234H */
    0xC9, /* ret */
};

int main (void) {
    int (* function-pointer) (void);
    int result;

    function-pointer = machine code;
    result = function-pointer ();
    printf ("RESULT =% 04x \n", result);

    return 0;
}
```

But that is a roundabout way abundant, partly because of the variables and on the other because of the manual preparation of the machine code. In part 26, we will learn to write assembly language directly in the C program.

And finally, there is the option to jump to any address as a function, according to the USR function in BASIC. The responsibility for the correct call as always carries the programmer:

```
void (* pointer) ();

pointer = irgendein_wert;
pointer ();
```

Multiple return values

This part has already become so long ... Therefore, we will only have one last application for pointers. We can use it well, if a function is supposed to return more than one value.

Suppose we want a 16-bit value share in its 8-bit halves. That would perhaps like this:

```
# Include <stdio.h>
```



```

void splitter (unsigned char * hi_zeiger, unsigned char * lo_zeiger,
               unsigned short val) {
    * Hi_zeiger = value >> 8;
    * Lo_zeiger = value & 0xff;
}

int main (void) {
    unsigned char hi;
    unsigned char lo;

    splitter (& hi & lo, 0x5678);
    printf (".% 02x = HI, LO =% 02x \n", hi, lo);

    return 0;
}

```

Homework

- Writes a program that dereferences a NULL pointer. As you get out, and the pointer shows? Explaining the results.
- What this result means that if you two pointers subtracting from each other? To solve this problem certainly helps a small test program:

```

#include <stdio.h>

int main (void) {
    int array [5];
    int * Pointer1;
    int * pointer2;

    Pointer1 field + = 4;
    pointer2 = field;
    printf ("Pointer1 =% u \n", Pointer1);
    printf ("pointer2 =% u \n", pointer2);
    printf ("DIFFERENCE =% u \n", Pointer1 - pointer2);

    return 0;
}

```

- Implements the functions strchr (), strrchr (), and strstr () Checks own success with a test that compares the results of your functions with the results of the complete functions in the standard library. To do this you have to of course call your functions differently, eg with a prefix "mein_". A little tip: Just think of some mean tests before you start your own implementations. Interesting are all searching for the first or last or nonexistent characters ...
- Explaining the results following program:

```

#include <stdio.h>

```

```

int main (void) {
    char char1;
    char characters2;
    int * pointer;

    char1 = 0x12;
    characters2 = 0x34;
    pointer = (int *) &zeichen2;
    printf ("dereferenced VALUE =% 04x \n", * pointer);

    return 0;
}

```

Part 22 - Arrays

Today's theme fits very well to the previous part, with Steven Vickers is the 22nd Chapter "Arrays". Because arrays and pointers are related.

Creating Arrays

We can create in C arrays of any data type:

```

zeichen_array char [5];
ganzzahl_array int [23];
void (* function-pointer array [81]) ();

```

Yes, the last definition is determined you have conjured a slight frown on his face ... Thus, the definition of an array of 81 pointers looks out on features. Unfortunately, like the z88dk this definition, therefore we remain a sample program you owe.

More about creating arrays we had in Part 16, please look into it there again.

Indexes

Just as repetition: the indices are in C from 0 (zero), not from 1 (one)! This is because the index of an element is a pointer operation actually. The term "array [0]" is the compiler as "* (array + 0)" considered, the notation with square brackets is syntactic sugar ;-).

The name of the array is thus of the type "pointer to a value of the type of the first member."

Multidimensional Arrays

Of course, in C multidimensional arrays are also possible. However, this is not supported by z88dk (This is a legacy from the origin of the z88dk, especially small-C.), Therefore there is not an example

program. Instead, we show only the notation:

```
drei_dimensionales_array int [10] [20] [30];
```

In the memory, the elements are then in the following order, though this is only for illustration:

```
[0] [0] [0]
[0] [0] [1]
...
[0] [0] [29]
[0] [1] [0]
[0] [1] [1]
...
```

If you want to use multi-dimensional arrays, you have to therefore create an appropriately sized one-dimensional array and run the index calculation itself.

Why C can be as fast

Unlike other programming binary compiled C is frequently very fast. Remember that the indexes are not checked for compliance with the limits! You can try this with the following example with:

```
# Include <stdio.h>

int main (void) {
    int before;
    int array [4];
    int it;
    int index;

    before = 123;
    for (index = 0; index < sizeof (array) / sizeof (int) index ++ ) {
        array [index] = 11 * index;
    }
    then = 987;

    for (index = -1; index <= sizeof (array) / sizeof (int) index ++ ) {
        printf ("%d: %d \n", index, array [index]);
    }

    return 0;
}
```

Homework

- Write a program that reads 10 numbers. Thereafter, the numbers are displayed in reverse order.
- The program from Task 1 should be amended so that the numbers are written in ascending order.

Whoa, that requires thinking up and implementing a sorting algorithm: it does not have to be beautiful or fast, he just needs to work! What needs to be changed so that the numbers are displayed in descending order?

- We want to know how many and which vowels are in a text to type. The program should therefore read individual lines of gets (), until a point is entered as a single character. The vowels are to be counted cumulatively. In the end, the numbers displayed and the program is ready.

- For professionals: Write a program that stores an array of pointers to character menu. It will show this menu: each row is a number and the text of the menu item. The user can then enter a number and the program returns the corresponding text. The whole is to be repeated until the user selects the last entry, the example "End" may mean.

Part 23 – When the Computer Gets Full

In the BASIC book by Steven Vickers is the chapter 23 "When the computer gets full", and so we will in this part of the C-course deal with the issue "memory".

No computer has (local) infinite amount of space available. Therefore, it is no problem even soon come to a small program to the limits. Today we will examine different scenarios ...

Too much space required for local variables

Many C programmers are not aware of this, but local variables are usually created on the stack of the processor. The local variables include the way the parameters are passed to functions. In our Z80 growing stack of end of the RAM towards low addresses at which lie but the BASIC program and also our C program.

We can estimate relatively simple, how much space we have available. The following test program offers an approach that is not included in the reasons of space, the BASIC memory. It is only the start address of the main function "main" and the address of the local variable "x" is issued:

```
# Include <stdio.h>

int main (void) {
    char x;

    printf ("% 04x MAIN IN, IN STACK% 04x \n", main, & x);

    return 0;
}
```

With a further experiment, we show that the compiler of the z88dk performs no tests during program

execution if the stack does not grow in the area into the BASIC encounters. Therefore we call recursively always the same function until the stack eventually destroyed the program:

```
# Include <stdio.h>

static void recursion (int depth) {
    speicher_verbrauch char [100];

    printf ("CALL% d \n", depth);
    recursion (depth + 1);
}

int main (void) {
    recursion (1);

    return 0;
}
```

A BASIC program, the interpreter in ROM, however, always check if he still has. However, one can aptly argue whether the run-time error "4" for a user is so helpful. Programmers, however rejoice because their program then (usually) still there ...

Too much space required for static variables

Just as we can occupy too much memory by local variables, static variables even when the problem occurs. The following program can compile without error:

```
# Include <stdio.h>

speicher_verbrauch static char [16000];

int main (void) {
    return 0;
}
```

However wanted the ZX81 (in its emulated 16K form) can not load the finished program, he came back with a reset. Due to the additional option "-m" by the way the compiler can be made to produce a so-called map file that lists all the addresses once again sorted alphabetically by name and numerically by address. This map (in English, in German: "Card") is helpful in estimating the size of a program.

Even if we have a program with a large amount of memory static variables (or machine code) get loaded, we will eventually run-time stack of too large: but this problem has the first chapter of this part treated.

Dynamic Memory Management

So after we first rolled problems, we now come to solutions!

Sometimes we do not know how much space we need for data. Then it is not useful to provide arrays with the maximum size, because it enables us to limit the users yes, depending on memory configuration. Moreover, it may also be that, depending on program execution times, the times must grow one or the other variable.

This can e.g. be a name management that will dominate different lengths and different names large numbers of names. One user knows louder Eve, Udos, Tim and Maya, but plenty. The other user for a noble acquaintance of nothing counts "of and" that is but limited in number.

The standard library provides us with dynamic memory management (http://www.z88dk.org/wiki/doku.php/library:memory_allocation) initially on two basic functions. With these we can (English, in German: "heap") from the heap can be an area and return this even if we no longer need it.

```
void * malloc (unsigned int size);
```

This function returns a pointer to a storage area at least the size in bytes transferred. If there are not enough memory for it, it returns a NULL value - we need to check the result is!

```
void free (void * address);
```

This function is used to the fetched by malloc () memory back "return" so that it again for the next requirements by malloc () is available. Incidentally, as the address must also be NULL, then of course nothing happened.

Whence, then, do the C program, the address at which it is how much space is available? In environments with a "real" operating systems, the heap is retrieved from the operating system. On our Zeddy we must take before calling malloc () once into their own hands:

```
long heap;
```

This global variable is required for administration.

```
void mallinit (void);
```

With this function, we initialize the administration.

```
sbrk void (void * address, uint size);
```

And with this function, we give the owner or manager, from which address how many bytes are available. The call is also more possible, if we have several free areas.

Let's try it again from the same:

```
# Include <malloc.h>  
# Include <stdio.h>
```

```

long heap;
freier_platz static unsigned char [1000];

int main (void) {
    void * pointer;

    mallinit ();
    sbrk (freier_platz, sizeof (freier_platz));

    pointer = malloc (998) / * more is not * /
    printf ("Address =% 04x \n", pointer);
    free (pointer);

    return 0;
}

```

The test program passes the memory that it has created with the static variable "freier_platz" to the memory management. If your Zeddy has anywhere else lying around memory, you can also specify this. ;-)

Attention! If you want to create the program, you must specify the option when linking "lmalloc" so that the functions of the standard library can also be added.

The usual standard library includes two other useful features that the allocating (That's the name of the requesting memory from professionals, "to allocate" the English, therefore = malloc allocate memory) more comfortable.:

```
void * calloc (unsigned int number, unsigned int size)
```

This function allocates space for number elements of size size. The memory is still initialized with zero bytes, which malloc () does not fact. On failure, they also return NULL. Unfortunately, the linker will not function. :-)

```
void * realloc (void * address, unsigned int size)
```

From time to time you would like to change the allocated memory in size. That is what makes this function, but it can happen that it is changing the address. Internally, the function retrieves namely first malloc () with the new size, if successful, the old contents copied in accordance with the size, and then the old area with free () returned. So anyway, does the simple implementation ... In case of error it returns NULL, then you can continue to use the old memory unchanged. Unfortunately, the version z88dk not deal with a given address of NULL.

This type of memory management tends to fragmentation, that is, that in case of multiple malloc () and free () sometime the managed memory is allocated in small pieces. Then it can happen with a request that the total number still enough memory was free, but not a coherent piece of the desired size.

Therefore the z88dk-makers, two other types of memory management are provided. However, these are not standard, so that we (make yourself learn faster smart!:-P) refer to the documentation.

Homework

- Writes the memory functions for the above-mentioned name administration. Test them with a test program. This can e.g. First determine the maximum amount which can be applied short name, which can be applied and then the maximum number of long names. Are the results plausible?
- Think about what to do if the simple implementation of `realloc()`. If you ignore this is, you write a small test program that eg the allocated memory will always be slightly larger, until it can not go any further. For analysis, the program can e.g. the current address and the size of output. Who here wants to go deeper, may consider methods to circumvent the unpleasant aspects ...

Part 24 – Counting on Your Fingers

Chapter 24 is in Vickers' Counting on your fingers "and treats binary and hexadecimal counting. But we can all (do not you?) And it's not really specific to any particular programming language. Before we start in the last parts of a real sprint, so there is this time a colorful collection of ...

Homework

- First, you can write a few simple programs:
- Write a program that asks the user for two numbers, and then all the numbers from the input first output until the last, inclusive.
- Write a program that reads the length, width and height of a cube, and outputs the volume! Sample output:

```
ENTER THE LENGTH: 2
ENTER THE WIDTH: 5
ENTER THE HEIGHT: 4
VOLUME SHALL 40th
```

- Write a program that asks the user whether he wants to continue. There are answers as accepted only strings "YES" and "NO". To start the program, the output "1 TEST" will appear. Each subsequent "YES", the next attempt ("2 TRIAL", "3 EXPERIMENT", ...).
- Writing a program that converts all the "r" and "R" of an input sentence (up to 31 characters) in "l" and "L". This yields a set, which pronounced that sounds like German with a Chinese accent.
- Write a function "exchange", which swaps the contents of two variables specified above. Writes an associated main program for testing. Note: Surrender the addresses of the two numbers and then work with the hands.
- Write a recursive function that calculates the power of 3 to the passed number. For example, if 4 is passed, the function returns the 81st Tests the functionality of a program.

The following programs are a little more difficult:

- Writing a program which reads an initial number and a final number. Ensures that the initial count is

less than the ending! Should be output every third number, which is located between the initial count and the final number. Used to:

- a while loop
- a do-while loop
- a for loop

- Sample output:

```
ENTER THE FIRST NUMBER: 1
ENTER THE NUMBERING: 15
NUMBERS IN BETWEEN: 2 5 8 11 14
```

- Write a program that lets the user enter as many numbers. After entering a 0 is the input to be terminated. Are to be issued, the number of input numbers, the sum of the input numbers and the (integer and rounded) average of all input numbers. Attention! The input 0 is not included in the calculation! Sample output:

```
INPUT (0 BREAKS DOWN): 1
INPUT (0 BREAKS DOWN): 2
INPUT (0 BREAKS DOWN): 3
INPUT (0 BREAKS DOWN): 0
POSTED ON THE NUMBERS: 3
TOTAL: 6
AVERAGE: 2
```

- "Ggt" write a function that calculates the greatest common divisor of two integers. The figures are to be transferred from the program and the greatest common divisor is returned. Note: This function must be recursive.

- Write a program that accepts any number of notes. The number is to be queried by the user when the program starts. The program aims to fulfill the following tasks:

- Reading the notes;
- Determine minimum and maximum;
- Calculation of the (integer, rounded) average value;
- View a list of all scores with average, minimum and maximum.

- Realized the whole program with dynamic memory management.

- Writes a feature called "add_arrays" that gets passed two arrays of the same size. You should then add the elements of the two arrays with the same index and store the sum in a third array passed the same size. To test their writes a program that the values of all three arrays done by

- Work outputs tabulated. Tip: The function gets passed the address of the first element of each array.
- To debug an error routine is useful to get an error number, line number and module name. This routine is to format and print an error message and exit the program. Used when calling the predefined macros of the preprocessor. Such is an example output:

```
CFBASIC_24-C.C (LINE 2): ERROR 17
```

- Speaking during troubleshooting, perform the following tasks can train her your analytical skills:

- Find out what's wrong:

Logs nothing:

```
for (x = 10; == x +20, x + = 1)
    printf ("%d \n", x);
```

Runs continuously:

```
for (x = 10, x <30, x = +20)
    printf ("%d \n", x);
```

Runs indefinitely:

```
for (x = 10, x + 20, x = 30)
    printf ("%d \n", x);
```

- Why number2 is not counted as expected?

```
# Include <stdio.h>

int main (void) {
    int num1 = 0;
    int num2 = 0;

    while ((num1 ++ <5) || (num2 ++ <5)) {
        printf ("%d = NUMBER1 NUMBER2 =%d \n", number1, number2);
    }

    return 0;
}
```

- What's wrong with following Quelltextstückchen?

```
record = 0;
if (record <100) {
    printf ("\ entry to be%d", record);
    printf ("\ nHOLE next entry ...");
}
```

- The following program has a problem. Attempts to detect it before it eingibt her and tentatively compiled. Which line causes the error?

```
# Include <stdio.h>

int main (void) {
    printf ("THIS IS A PROGRAM");
    do_it ("WITH A PROBLEM!");

    return 0;
}
```

```
}
```

- What does the following program? First think, then try! If there are differences between your prediction and the actual output, corrected for the program!

```
# Include <stdio.h>

void print_letter2 (void);

int ctr;
letter1 char = 'X';
Letter2 char = '=';

int main (void) {
    for (ctr = 0; ctr <10; ctr + +) {
        printf ("% c", letter1);
        print_letter2 ();
    }

    return 0;
}

print_letter2 void (void) {
    for (ctr = 0; ctr <2; ctr + +) {
        printf ("% c", Letter2);
    }
}
```

- At close to get some comprehension questions:

- Given two integer variables named number1 (value = 5) and number2 (value = 10) and two pointer variables. Writes to complete each step, change the values to:

```
Pointer1 = &zahl1;
pointer2 = &zahl2;
* Pointer1 + = 20;
* Pointer2 - = 3;
* = * Pointer1 pointer2 + 5;
* = * Pointer2 Pointer1 - number2;
```

- Writes a function prototype for a function called "do_it" which takes three arguments of type char and then a long returns.

- Writes a function prototype for a function called "print_a_number" which receives a single int and nothing returns to the caller.

- What type of data will return the following functions?

```
int print_error (long error_number);
long read_record (record_number int, int size);
```

- Writes a declaration for a pointer to a char variable. Renames the pointer "char_pointer".

- A program can have both a global and a local variable with the same name? Write a program to prove

your answer.

- Writes the nested query from scratch with one and if a logical link:

```
if (x > 1)
    if (x < 10)
        statement;
```

- An uninitialized variable is read. Under what circumstances has the value? Trying to prove that their content is random.

Part 25 – How the Computer Works

Chapter 25 explains Steven Vickers "How the computer works" who dabble so as the individual circuits in Zeddy together to finally end up with PEEK and POKE. We coins on the other hand, to the software that can be so written in parts.

Therefore, today we bring you closer as you can modularize a larger program. Here you will also realize what it even is a linker, which we have mentioned in Part 8. And of course, we will show how you realized PEEK and POKE in C.

Why modularize?

So far we have posted a rather trivial programs, the more rare than one screen were high. But this is a special case, because most "real" programs are much larger.

When you think of a sufficiently complex program, you can safely identify individual parts that belong together thematically. This can for example his reading of user input, the output on the screen, the implementation of algorithms, or conversion functions.

In BASIC programs you've solved the sure fact that you've grouped related programs, perhaps even in the same number in the thousands line number. In C, it is worth it, however, write to related functions into a separate source file. The resultant module can be compiled alone, and it can also be used in several different applications.

For a complete program you will then select the modules that are needed, and "bind" them together. And that is precisely the task of a linker: Connect to link =.

In this way it provides you a collection of useful modules. Exactly what has happened also in the C standard library. There often even each function has been compiled as a module. Because of a library only the required functions are tied to the program, a program is only as large as necessary.

But why this intermediate step with the compiler? This comes from the days when computers were still small and slow. The compilation takes a really long time as opposed to the left, and the source is usually larger than the machine code generated. To save time and space, so it was cheaper in advance translated object files (these are the results of the compiler or assembler) use.

Incidentally, modules written in other languages, are admissible if the format for the object files is compatible. When z88dk the offers for modules in assembler.

An example

We think of a small sample! The program is to read in two numbers, add them and display the result. Granted, this is trivial, but we want to flay no sides! :-)

Each of these three steps, we implement in a separate module, and then we need a main program that uses all three.

This is the module "CfBASIC_25-1a.c" to read in numbers:

```
# Include <stdio.h>
# Include "CfBASIC_25-1a.h"

lies_zahl int (char * prompt) {
    char line [10];
    int number;

    printf ("% s", prompt);
    gets (line);
    puts ("");
    sscanf (line, "% d", & number);

    return number;
}
```

And this is the module "CfBASIC_25-1b.c" from for adding:

```
# Include "CfBASIC_25-1b.h"

int add (int summand1, summand2 int) {
    + return summand1 summand2;
}
```

We also need a copy "CfBASIC_25-1c.c":

```
# Include <stdio.h>
# Include "CfBASIC_25-1c.h"

void gib_zahl_aus (declarationofcompliance char *, int num) {
    printf ("% s=% d \ n", declarationofcompliance, value);
}
```

Finally, we put the program in a "CfBASIC_25-1m.c" together:

```

#include "CfBASIC_25-1a.h"
#include "CfBASIC_25-1b.h"
#include "CfBASIC_25-1c.h"

int main (void) {
    int num1;
    int number2;
    int sum;

    number1 = lies_zahl ("FIRST NUMBER");
    number2 = lies_zahl ("SECOND NUMBER");
    sum = add (number1, number2);
    gib_zahl_aus ("SUM IS", sum);

    return 0;
}

```

If ye anseht precisely the sources, you will fall on # include lines that in quotes (Why in quotes instead of angle brackets? See Part 9!) Stand. And you wonder why? Well, at least in the case of the main program, you should find the answer with a little thought: the called functions must be known to the compiler, so before calling. And the inserted header files do with corresponding prototype:

For the read-in module contains the header file "CfBASIC_25-1a.h":

```
extern int lies_zahl (char * prompt);
```

For the addition module contains the header file "CfBASIC_25-1b.h":

```
extern int add (int summand1, summand2 int);
```

And for the output module contains the header file "CfBASIC_25-1c.h":

```
extern void gib_zahl_aus (declarationofcompliance char *, int number);
```

It is interesting that the z88dk the keyword "external" desperate to see - according to the standard it is not necessary, does not harm either.

The reason for inserting in the modules themselves is a professional: the compiler will then receive both the prototype and the implementation of any function to translate. And then bitched automatically when the prototype in the header file 'feature will not be implemented.

Each of these modules may be compiled separately, as we learned in Part 8:

```

zcc zx81-vn +-Wall-c-CfBASIC_25 1a.c
zcc zx81-vn +-Wall-c-CfBASIC_25 1b.c
zcc zx81-vn +-Wall-c-CfBASIC_25 1c.c

```

```
zcc zx81-vn +-Wall-c-CfBASIC_25 1m.c
```

This results in corresponding object files, each with the file extension ". O".

And all left together!

Now the program can be tied together:

```
zcc + zx81-vn-Wall-create-app-startup = 2 CfBASIC_25-1m.o CfBASIC_25-1a.o  
CfBASIC_25-1b.o CfBASIC_25-1c.o CfBASIC_25-o-1.bin
```

This actually creates a program, unfortunately it does not work. The reason is that when z88dk each call to "zcc" the file "zcc_opt.def" in the current directory deletes and therefore go the gathered settings will be lost, which are necessary when linking. There is the option "preserve" that prevents the deletion, but the file continues to grow.

Therefore, you can save yourself the individual compiling the source code and the same one on the command line:

```
zcc + zx81-vn-Wall-create-app-startup = 2 CfBASIC_25-1m.c CfBASIC_25-1a.c  
CfBASIC_25-1b.c CfBASIC_25-1c.c CfBASIC_25-o-1.bin
```

And make it work now! Our today's computers have indeed no shortage when it comes to storage space and processing power, so the previous compilation is really time-saving.

Global program, module and global function locally

We have been off and on talking about the area in which the names of functions and variables are visible. We can distinguish three basic areas:

- Program globally: Such functions and variables are known throughout the program. Their declarations are marked as "external" and may not have the source code implementing the "static". Variables of this type can not be defined inside functions.
- Module-global: These functions and variables are known only in the source module in which they are defined. There, however, they are known in every function. They are marked as "static"; such variables must also be defined outside of functions.
- Local: This area is only for variables. A local variable is known only within the block in which it is defined.

An example should make sure equally clear:

```
/* This is only the prototype */  
extern void programm_globale_funktion (void);  
  
/* This function is known only locally in the module. */  
static void modul_globale_funktion (void);
```

```

/* This is not created out of space! */
extern int programm_globale_variable;

/* The value is maintained throughout the lifetime of the program. */
static int modul_lokale_variable;

/* Finally, memory for the above variables: */
int programm_globale_variable;

/* And the implementation of the global function: */
programm_globale_funktion void (void) {
    static int statische_lokale_variable;
    {
        int automatische_lokale_variable;
        modul_lokale_funktion ();
    }
}

```

Through these areas you are able to for your module-specific functions and global variables and all local variables to use any name. You need to make any contortions to avoid conflicts. Even if you are in any function a local variable named "index" takes all these are independent!

And that's the moral of the story: the visibility reduced to the absolute essentials, and you will have the least problems.

Libraries of useful functions

Now you've written to you a lot of nice and useful modules for all sorts of purposes, but it is too cumbersome, for a program exactly specify the correct list. How would it be if the linker finds out itself which modules are added to tie for a program? That would be great, right?

And exactly that happened too! These modules are all gathered in one or more libraries that will eventually be scanned by the linker. Nothing else is happening since the beginning of the course, for example, if the input and output functions from the standard C library is used.

Let us assume that we have except the addition function (file "addiere.c") ...

```

#include "CfBASIC_25-2-lib.h"

int add (int summand1, summand2 int) {
    + return summand1 summand2;
}

```

... also a subtraction (file "subtrahiere.c") ...

```

#include "CfBASIC_25-2-lib.h"

```



```
int subtract (int minuend, subtrahend int) {
    return minuend - subtrahend;
}
```

... and a multiplication function (file "multipliziere.c") ...

```
# Include "CfBASIC_25-2-lib.h"

int multiply (int factor1, factor2 int) {
    return factor1 * factor2;
}
```

... and finally, a division function (file "dividiere.c") ...

```
# Include "CfBASIC_25-2-lib.h"

int divide (int dividend, int divisor) {
    return dividend / divisor;
}
```

Yes, we know, the example is trivial beyond words! But it's only a matter of principle, your modules can be arbitrarily complex, and they may even build on each other, ie so that their functions call each other.

Such is its header file "CfBASIC_25-2-lib.h" from (The __ LIB__ is necessary for the z88dk ...):

```
extern int __ LIB__ add (int summand1, summand2 int);
extern int __ LIB__ subtract (int minuend, subtrahend int);
extern int __ LIB__ multiply (int factor1, factor2 int);
extern int __ LIB__ divide (int dividend, int divisor);
```

And this is the program "CfBASIC_25-2m.c" that uses a function from the library. For simplicity also the input and output modules of the first embodiment can be reused:

```
# Include "CfBASIC_25-1a.h"
# Include "CfBASIC_25-2-lib.h"
# Include "CfBASIC_25-1c.h"

int main (void) {
    int num1;
    int number2;
    int difference;

    number1 = lies_zahl ("FIRST NUMBER");
    number2 = lies_zahl ("SECOND NUMBER");
    difference = subtract (number1, number2);
```

```

        gib_zahl_aus ("DIFFERENCE IS", difference);

    return 0;
}

```

Well, so you have all source code in mind. First we compile all the modules you want in the library:

```

zcc + zx81-vn-make-lib-Wall addiere.c
zcc + zx81-vn-make-lib-Wall subtrahiere.c
zcc + zx81-vn-make-lib-Wall dividiere.c
zcc + zx81-vn-make-lib-Wall multipliziere.c

```

Here it is, a (weitere!) Special Note of z88dk: the name of the source file must be the name of the contained function meet. And apparently allowed only one code per program-global function may be present.

For the construction of the library we call the linker (in his appearance as z88dk-assembler) directly, but the program "zcc" apparently knows no way to carry out such a call easier:

```

z80asm-e-ns-nm-Mo-2-xCfBASIC_25 lib.lib addiere.o subtrahiere.o dividiere.o multipliziere.o

```

The various options you can in the help (by calling with "z80asm-h") read:

"-D": translation only if source newer than object code, seems to be necessary, the z88dk-makers use it too.

"Ns": generating a symbol table off.

"Nm" means the production of a map file off.

"Mo": file type of object files here: "* o. '.

"X ...": A library with the specified name to be generated from the modules.

So now we have a library called "CfBASIC_25-2-lib.lib". Unfortunately, the z88dk not any directory (not the current!) Search libraries, so we need to copy the file to the z88dk directory for libraries. For me (Bodo) that is "/ home/bodo/ZX81/ZX-C/z88dkv1.8/z88dk/lib/clibs /".

Finally, the actual program can now be compiled and linked! This is done with this command line:

```

zcc + zx81-vn-Wall-create-app-startup = 2 CfBASIC_25-2m.c CfBASIC_25-1a.c
CfBASIC_25-1c.c-lCfBASIC_25-2-o-lib.lib CfBASIC_25-2.bin

```

You see the additional option "-lCfBASIC_25-2-lib.lib", right?

For a single program of this effort is of course higher than the "normal" method, and so it is worthwhile even if you only developed a program Sun But it is starting to pay off, if you use such a library in multiple programs. Or if you give them more to others. For example, I can imagine that there may be an alternative HRG library, or a library for accessing SD card, or a library for a UART or a modem, the sky is the limit!

PEEK and POKE in C

Finally, we come back again content on Vickers. And also appeared in the forum early questions about PEEK and POKE.

In C, the request runs on arbitrary memory locations actually quite simple, namely pointers. To do this you define a pointer to the appropriate data type. You can find him even initialize it with the address if it is not to be used for different addresses:

```
unsigned char * pointer = 0x4009;

read = * pointer; /* PEEK */
* Pointer = zu_schreiben /* poke */
```

But if it's fun for you, you can write to you of course functions for PEEK and POKE:

```
unsigned char PEEK (unsigned char * address) {
    * return address;
}

void poke (unsigned char * address, unsigned char value) {
    * = Address value;
}
```

C even offers you the opportunity to access the same with the correct data type to the memory. For data types with more than one byte size the individual bytes of course must be in the correct order in store!

But these ideas, the makers of z88dk had beforehand, so the standard library includes the following functions, which are declared in the "stdlib.h":

```
bpoke void (void * addr, unsigned char byte);
WPOKE void (void * addr, unsigned int word);
unsigned char bpeek (void * addr);
WPEEK unsigned int (void * addr);
```

So, this part is yes' again become extremely long! In the next part things remain just as interesting as we will deal with the interface to machine language ...

Part 26 – Using Machine Code

This item is sure to taste the whole Bitpfriemler who are familiar with assembler: Chapter 26 is at Steven Vickers "Using machine code". And here we will explore halfway, how we can use assembler and C together.

Inline assembler

The z88dk offers two alternatives for assembler directly in C source code:

- Between the preprocessor "# asm" and "# endasm" we can write any assembler source. This is even treated by the optimizers - if you do not want that, you have to choose the third option, described below on a separate module.
- The assembler source code is passed as a string to `asm ()`. This is not really a call to a function, but rather a kind of macro preprocessor. The string can contain multiple lines, but 't \ ' works correctly with (tabulator) and '\ n' (newline) should be formatted!

A simple example illustrates the application. The program sets and reads the I-register, which is known to be used for the screen. After setting up 0x0E will appear on the screen so "scrap", please do not scare!

```
# Include <stdio.h>
# Include <stdlib.h>

static unsigned char lies_i (void) {
# Asm
    ld a, i
    ld h, 0
    ld l, a
# Endasm
}

static void setze_i (unsigned char value) {
# Asm
    ld hl, 2
    add hl, sp
    ld a, (hl)
    ld i, a
# Endasm
}

int main (void) {
    unsigned char old_i;

    old_i = asm ("ld \ ta, i \ nld \ th, 0 \ nld \ tl, a \ n");
    printf ("I CONTAINS% 02x \ n", old_i);

    asm ("ld \ ta, 0x0E \ nld \ ti, a \ n");

    sleep (5);

    printf ("I CONTAINS% 02x \ n", lies_i ());
    setze_i (old_i);
}
```

```
        return 0;
    }
```

Somewhat disconcerting is the warning that the call to `setze_i ()` throws:

L: 28 Warning: # 33: Call to function without prototype

But the finished program is correct ...

The pedant will note now that when reading the I register, a 16-bit value is produced. Well, this seems the so-called integer-promotion (C all integer calculations fundamentally expanded to the width of an int Unabridged, however, that would be in long calculations indeed counterproductive.) To z88dk not be correct. Anyway, we need to delete the upper 8 bits really even ...

Input and output

Of course, we can write in this way also functions or macros for the Z80 instructions for input and output. But the z88dk-makers have already done that for us, in the "stdlib.h" declared the following functions:

```
unsigned int inp (unsigned int port);

void outp (unsigned int port, unsigned char byte);
```

The macro variants to `inp` and `M_INP` `M_OUTP` and be equally applied. They have the advantage that no function call is generated, but the machine code is inserted just at the point of the call. However, the parameters ("port" and "byte") to be constant!

The port addresses are 16 bits wide, the way how the data type is already showing.

Modules in assembly

There is also a third option, Assembler and C with each other, by writing a module entirely in assembly language. Then we have to comply with various conventions.

First, the name of the global functions are exported with `xdef`. They must begin with an underscore, which is used by the C compiler to get no conflict with reserved names.

When we call a C library function of the assembler module out, we have to declare their names with `LIB`. Such functions have a name without any additional underscore - well, that's z88dk after all ...

If we want to have parameters passed in registers instead of on the stack, the function prototype and the `__FASTCALL__` receive.

This is how such a module, for example, the mirror function `()` should reflect the parameters passed bitwise:

Xdef _mirror; announce start address

LIB puts; declare a library function

Unsigned short __ FASTCALL__ mirror (unsigned short);

```
_mirror:
    ld b, # 8
loop1:
    add hl, hl
    rra
    djnz loop1; reflect the first 8 bits

    ld l, a; save half results in L

    ld b, # 8
loop2:
    add hl, hl
    rra
    djnz loop2, flip the second 8 bits

    ld l, h
    ld h, a, composed entire result

    push hl

    ld hl text,
    push hl
    call puts
    pop hl; ready signal output

    pop hl
    ret

text:
    defm "DONE.", 0
```

Of course, we write the function prototype in a header file, which can then be the use C program included by # include:

```
extern unsigned short __ FASTCALL__ mirror (unsigned short value);
```

The example would not be complete without the test program:

```
# Include <stdio.h>
# Include "CfBASIC_26-2a.h"
```

```

int main (void) {
    printf ("%x MIRRORED =%x \n", 0x1248, mirror (0x1248));
    printf ("%x MIRRORED =%x \n", 0x3DE6, mirror (0x3DE6));

    return 0;
}

```

Extensions to C

To assembler functions even better to mix with C functions, the authors have come up with the z88dk three new keywords:

- Works like `return_c` return sets, but the carry flag.
- Works like `return_nc` return extinguished, but the carry flag.
- `IfError` This is best compared to "if (carry flag)."

Here, too, certainly helps a small example that even has no assembler source, it's also very beautiful, no.

```

# Include <stdio.h>

static void is_null (int value) {
    if (value == 0) {
        return_c;
    }
    Else {}
    return_nc;
}

static void test (int val) {
    is_null (value);
    IfError {
        printf ("%d IS NULL \n", value);
    }
    Else {}
    printf ("%d IS NOT NULL \n", value);
}

int main (void) {
    test (0);
    test (81);

    return 0;
}

```

If you look at the generated machine code (option "-a", see Section 8), you'll notice that even these extensions generate faster code than the standard version with a small integer as boolschem value.

Part 27 – Organization of Memory

So far we have met on our way from BASIC to C not really revolutionary new concepts. This should change now, because we will extend our knowledge of the type concept in C. And because data types for variables, which in turn occupy memory, which fits well with the Chapter 27 of Vickers' book called "Organization of memory".

Structures

Suppose, you want to save a person's first and last name and age. Previously you had to create one variable. How easy would it be if there were a data type that possessed all three data storage! And that is what goes by her defining a structure for each of the three data has ever a component (also known as field or element). Who now thinks of a database, is not such a bad!

Such a structure is introduced by the keyword "struct". Thereafter, optionally followed by a structure name, but not necessarily. You need the structure name if you later want to define multiple variables of the same type.

Then comes a block in braces, for each component contains a data type and a name. This component names have nothing to do with variable names, but there are really only the names of the components!

You can also equal to the structure declaration to specify a variable name, as the following example shows:

```
# Include <stdio.h>
# Include <string.h>

int main (void) {
    struct {
        first name char [30];
        last_name char [30];
        unsigned char old;
        Person};

    strcpy (person.vorname, "BODO");
    strcpy (person.nachname, "Wenzel");
    person.alter = 46;

    printf ("% s% s% u IS YEARS OLD. \ n",
            person.vorname person.nachname, person.alter);

    return 0;
```



```
}
```

The individual components are accessed by a point after the variable name and the component name is written. Of course, both read and write accesses are possible!

Unfortunately, the z88dk no nested structures, ie structures which have in turn components with a structure data type. They can be declared, but the experiments were so ... um ... disappointing. "Normal" C compilers have it but no problems.

Superimposed structures (unions)

Sometimes it makes sense to store data of different types in the same store. Which can e.g. be the case when you are in a variable once the size of a rectangle (width and height), other times but want to save the diameter of a circle.

A simpler example shown in the following program which elicits the order in which values of more than one byte can be stored. This is the so-called "endianness" (<http://de.wikipedia.org/wiki/Byte-Reihenfolge>, the name comes from "Gulliver's Travels" back. There is a country whose people are divided into two camps, the breakfast eggs at the blunt or pointed end open).:

```
# Include <stdio.h>

int main (void) {
    union {
        unsigned short s;
        unsigned char c [2];
    } Test;

    test.c [0] = 0x12;
    test.c [1] = 0x34;
    if (test.s == 0x1234) {
        puts ("BIG ENDIAN (MSB FIRST)");
    } Else if (test.s == 0x3412) {
        puts ("LITTLE ENDIAN (LSB FIRST)");
    } Else {}
        puts ("endianness can not be determined");
    }

    return 0;
}
```

Such a union is just declared as a structure. Only occupy the same space all the components, the union is only as large as necessary for the largest component.

Type Definitions

If we want to create multiple variables of the same type of data, we have eg Type struct or union name

and then write down in detail. Also, from time to "unsigned long" way too long, or is "int" us to machine-dependent. Then we can create your own data types, by the keyword "typedef", a data type, and then write down a type name. So no variable is defined, but a new data type!

This is also interesting when you write a library, but the users want to protect the internals. The details do not need to know, and they are actually not even play around with it. It is also like a data type such as "uint_8" defined, which is an unsigned integer type with 8 bits:

```
typedef unsigned char uint_8;
```

The following example creates two new data types POINT and RECT be defined. These can then be used for variables and parameters:

```
# Include <stdio.h>

typedef struct {
    unsigned int x;
    unsigned int y;
POINT};

typedef struct {
    unsigned int l;
    unsigned int o;
    unsigned int r;
    unsigned int u;
RECTANGLE};

static int ist_punkt_in_rechteck (POINT * point, RECTANGLE * rectangle) {
    return point-> x> = rectangle-> l && point-> x <= rectangle-> r &&
        point-> y> = rectangle-> o && point-> y <= rectangle-> u;
}

int main (void) {
    static rectangle RECTANGLE = {30, 40, 150, 200};
    static POINT points [] = {
        {10, 10}, {60, 90}, {110, 170}, {160, 250}
    } / * Cool, as this array is initialized! */
    int index;

    printf ("RECTANGLE (% u,% u) - (% u,% u) \n",
        rechteck.l, rechteck.o rechteck.r, rechteck.u);
    for (index = 0; index <4, index ++ ) {
        printf ("POINT (% u,% u) IS"
            points [index] x, points [index] y).;
        if (ist_punkt_in_rechteck (points + index, and rectangular)) {
            puts ("IN.");
        }
        Else {}
    }
}
```

```

        puts ("OUTSIDE.");
    }
}

return 0;
}

```

Here is a nested structure for RECTANGLE would have made sense, but that has not generated z88dk correct code for it.

As you can see, you can also declare pointers to structures. Especially with the transfer of functions and saves the code memory, because a pointer is often less than the actual structure. You must not forget, however, that the function via the pointer can change the contents of the original structure!

Access to the components of a structure, a pointer to the present, can be initially constructed entirely logical:

```
(* Zeiger_auf_struktur). Component
```

But there is a nice shortcut that looks optically like a pointer. This has also been used in the example:

```
zeiger_auf_struktur-> component
```

Bitfields

Another way to declare components, the so-called bit fields. This is again interesting for people with the ear of the hardware, as it enables individual bits in a byte declared beautifully in high-level language:

```

struct {control_word
    unsigned int global_enable: 1;
    unsigned int selector: 3;
    unsigned int unused: 2;
    unsigned int enable_a: 1;
    unsigned int enable_b: 1;
}

```

This could, for example, declare a byte following structure:

```

Bit 7 global_enable
Bit 6 selector (bit 2)
Bit 5 selector (bit 1)
Bit 4 selector (bit 0)
Bit 3 (not used)
Bit 2 (unused)
1 enable bit A
Bit 0 enable B

```

As you can see, for a bit field of the int data type is used, the best is yet further specified by "unsigned" or "signed". After the component name followed by a colon and the number of bits that should have this component.

Unfortunately, the C standard does not specify in which direction and in which order the individual bit fields as are grouped in bytes! If you use the words to e.g. access control bits of a PIO, you should ensure through experimentation that your compiler makes in your sense.

Unfortunately the z88dk support (yet) bitfields ...

Return Types

C also eventually allowed to return a user-defined data type, a structure or a union. Unfortunately dominates z88dk not even ...

Homework

- Declares a structure called the "time", which has three integers as components. This will save hours, minutes and seconds.
- Write a program in which a structure is defined, which can store the name, surname, date of birth and 5 marks of a student. In addition, an array can be defined, which can hold 15 students a class. The information and grades of each student should be entered. The user can then decide if he wants to have spent all data or all of the notes of the students as a table.

Part 28 – System Variables

Chapter 28 tells us of the Vickers "system variables", therefore we briefly. Otherwise, this course concludes with this part, but this is not the end! Quite the opposite goes for those of you that have been detected by the C virus, only now the real fun on ...

System Variables

To easily access the system variables of the ZX81, it can look a corresponding structure (see last section) to write to a *. H file. The following template, this structure still stands directly in the code, but here we show you another extension of z88dk. With "@" and a number you can declare an external variable to a fixed address.

```
# Include <stdio.h>

struct {
    unsigned char ERR_NR;
    unsigned char FLAGS;
    unsigned short ERR_SP;
    unsigned short RAMTOP;
```

```

        /* And so on ... */
SYSVAR} @ 16384;

int main (void) {
    printf ("%x =%x \n", & SYSVAR.ERR_NR, SYSVAR.ERR_NR);
    printf ("%x =%x \n", & SYSVAR.FLAGS, SYSVAR.FLAGS);
    printf ("%x =%x \n", & SYSVAR.ERR_SP, SYSVAR.ERR_SP);
    printf ("%x =%x \n", & SYSVAR.RAMTOP, SYSVAR.RAMTOP);

    return 0;
}

```

Additional standard features

Throughout the course, you have already some of the functions of the standard library met. But there is still much to discover, this you can see by the various header files. Not only standard C, but also provides you the z88dk much room to play with.

To function correctly classify found, it is worth reading the associated comments, the search in the example programs of z88dk, and especially the research on the internet or reading a good book on C.

Homework

A final homework we want to make more, which treats a typical problem in computer science, which is also popular with modified C. It's about linked lists.

Below a list of her can imagine something already. Within a program, such are often used for storing a previously sorted undetermined number of objects.

Of course we can add us to an array, which is of varying number always adapted by `realloc()`. Consuming it but also operations that fit somewhere in the middle of the list, or delete objects. Because the computer must be so constantly large amounts of data back and herkopieren.

A better solution is to generate each object individually with `malloc()` and remove, if necessary with the `free()` again. The actual list is created when one or more for each object pointer is stored on neighboring properties - and that is the "daisy chain"!

To make things not to do too complex, we limit ourselves to a single-linked list, which takes integer values in ascending order. We need functions to add values (without creating double), to find existing values, and remove values.

The whole thing is to be tested with a main program that performs the following steps. After each step or sub-step to the whole list will be issued a check:

- Search 23 => no discovery.
- Do not remove from 81 => there exists, nothing happens.
- Adding 42, 23, and 81 => the output for each sub-step shows that the list is sorted.

- Remove from 42nd
- Adding 0 and 666 This is a test of whether is correctly added at the beginning and at the end.
- Adding 23, 666, and 0 => no change because they already exist.
- Search 80 => is not found.
- Do not remove from 123 => there exists, nothing happens.
- Search 81 => is found.
- Removal of 666, 0, 23, and 81 => at the end of the list is empty.

That's it

We hope you had a lot of fun and new experiences with the language C. Some of the experiences you can also make sense to use in BASIC programs. We are excited about what caused for great programs!