geoCom workshops, translated from the original German at:

http://geos-printarchiv.de/archive/workshops/geocom-workshop

(last updated 2018-06-24 by Cenbe)

GeoCom Workshop #1

Author: Olaf Dzwiza

Introduction to GeoCom

Recently, there is a new, very powerful, compiler-based program development system for GEOS 64/128. This workshop will give an introduction to programming with this software, simply called "GeoCom".

Interpreter versus Compiler

The GeoCom handbook often compares to GeoBasic because GeoCom is also BASIC-oriented. It should not be forgotten that GeoBasic is an interpreted language, while GeoCom is compiled. A program that runs with an interpreter is processed by the computer completely different than a compiled program and is therefore structured differently. While an interpreter translates each program line into machine language if it is needed, and then "forgets" the translation again, a compiler translates the program, the so-called "source code" into machine language once, thus generating a code which can be executed immediately by the computer. Both options have advantages and disadvantages:

Interpreter:
The program can be tested at any time, even partially; depending on the interpreter, the input of "RUN" or similar is sufficient. The fact that each line must be translated individually and on reaching each time, interpreter programs are usually quite slow. The possibility of running only individual program sections leads to a very confusing programming, the so-called unstructured "spaghetti code" is created quickly.

Compiler:
The translation process takes place only once. This allows the program to run very fast, but never to be tested (partially) immediately during writing. A compiler forces a clear, structured design and thus promotes clear programming.

Basically, compiler programs are constructed differently than interpreter programs. The source code is divided into two sections: the declaration section and the instruction section.

The declaration section sets the program name of the code to be generated, passes parameters to the compiler, and - most importantly - lists all the variables used in the program. The type of the variable must be declared so that the compiler knows how to handle the variable.

GeoCom knows the following types:

- **Byte**: A variable that is exactly one byte in size, that is, can take values from 0 to 255.
- **Integer**: A variable that is two bytes in size. It is divided into LOW and HIGH bytes and can take values from 0 to 65535.
- **Real**: A variable that can store "normal" real numbers with decimal places.
- **String**: A variable that stores a string of up to a maximum of 255 characters.
- **Row**: A set of variables of a freely selectable type, comparable to one-dimensional arrays in standard BASIC.

How these variables are declared, we will get to know in the course of the workshop. In addition, there are still declarations for objects, labels and data series, but more on that later.

The GeoCom development system

The source texts are, as some of you may already know from MegaAssembler, written with geoWrite. The full comfort of this word processor is thus available, the programmer does not have to struggle with a slow editor like GeoBasic. The compiler is started by double-clicking in the desktop or by the aid "Start GeoCom", which reloads it or, if desired, jumps to the object editor (in a later part). The source code is now immediately translated into memory(!), it must be saved afterwards by the programmer. If the source code is incorrect, a detailed geoWrite document will be created in which all errors are described, an example we will look at next time.

Structure of a GeoCom program

The compiler first requires the specification of the name under which the program should be saved, this can also be edited after compiling. Then the class must be specified (see corresponding entry in the info sector of a file) and finally the author. This information must always be at the beginning of a program.

Thereafter, declarations can be made for memory division (program, constant and variable range). If these do not follow, the compiler uses default values that are sufficient for most purposes. Finally, the variables must be declared and, if necessary, the operating mode (GEOS 64 or 128, screen mode) specified.

First tips for programming

Before the next time it really starts with the programming, here are some tips to try:

- PRINT is the only command with "?" abbreviate. However, I think that this should not be done for readability. The command must never stand alone. To output a blank line, use SETPOS or output an empty string. Likewise, PRINT can only output (!) Strings.
- so-called straight programs without END at the end crash after expiration. This is logical because the compiler installs the END routine to return to the desktop. If the command is missing, the program is considered not finished. In this case, the compiler does not provide an error message.
- PRINT does not scroll the screen. If accidentally positioned outside the screen, a crash may occur.

In the next part we will look at some points to consider in variables and comparisons, then step by step a program with graphical representations, printouts and disk routines will be developed.

Olaf Dzwiza

Note

When writing these lines, GeoCom is still not available.

According to a recent info from Olaf, GeoCom will definitely come out in late April! For this reason alone, part #1 of the workshop has already been started in this issue. Once GeoCom has been approved by the author, it can also be obtained from the Geos User Club. However, we can not promise whether the mentioned date can be met until the software is available.

Thomas Haberland

GeoCom Workshop #2

Author: Olaf Dzwiza

Introduction to Programming

After the introduction in GeoCom in the last issue, the use of variables and comparisons will be demonstrated this time with the example of a small calculation program with real numbers.

For our program, we first need the declaration part:

```
NAME "Calc Example"
CLASS "Workshop-PRG  V1"
AUTHOR "Olaf Dzwiza"
```

We will use three real numbers that are simply called a, b, and c. The corresponding definition must therefore be:

```
REALVAR a, b, c
```

This line follows immediately above. For further programs, please note that a maximum of 127 characters in a line and a maximum of 12 values in the definition section may be after a type designation. It is also important that command words are always uppercase, variables and labels are always lowercase! Through these declarations, which seem funny at first, the programmer takes some of the compiler's work off and thus ensures a quick translation.

But now to the actual program: Two real numbers (5.41 and 1.41) should be subtracted from each other, that result (4) is stored in a third variable and compared with the integer number 4. To do this we first have to assign the desired values to the variables:

```
a = 5.41
b = 1.41
```

Now we want to show our actions on the screen, it makes sense to clear it first (CLS) and absolutely necessary to specify the output position (SETPOS x, y):

```
CLS 0
SETPOS 10,10: PRINT "Real values:"
SETPOS 5,20: PRINT "a = 5.41"
SETPOS 5,30: PRINT "b = 1.41"
SETPOS 5,45: PRINT "Calculated: c = ab"
```

The calculation takes place immediately:

```
c = a - b
```

Now we are able to output the result, i.e. the value of the real variable c. However, the PRINT command can only display strings, so the result must be converted to one:

```
SETPOS 5,60: PRINT "Result:";: PRINT (STR c)
```

The result is expected to be 4. By a comparison we want to check this:

```
IF c = 4
 SETPOS 10,80
 PRINT "Right!"
ENDIF
```

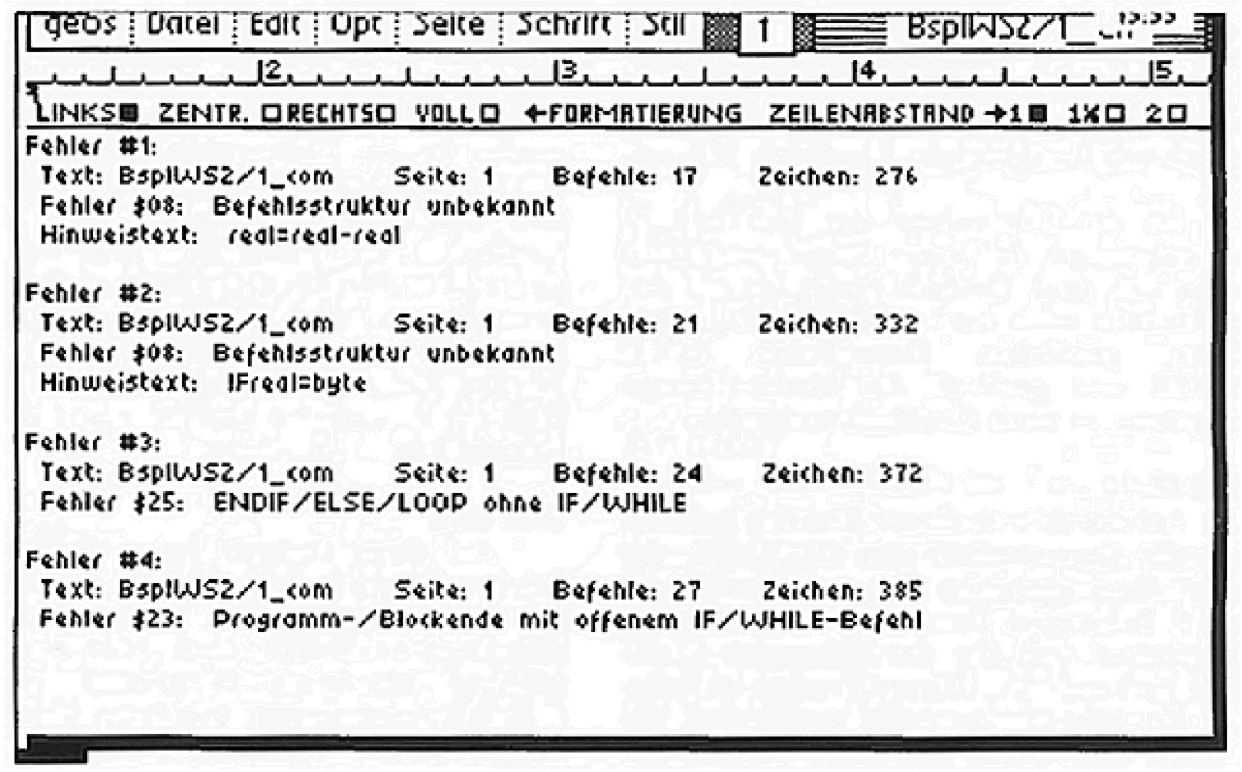
Now we leave the picture briefly on the screen and then return to the desktop:

```
WAIT 200
END
```

The END must not be forgotten in any case, otherwise the compiled program runs over the end and crashes. If the compiler finds the END command, it will use the routines to return to the desktop.

We can start the compilation process and ...

ERROR! (see picture below)


geos
Datei
Edit
Opt
Seite
Schrift
Stil
1
BsplWS2/1_...
LINKS■ ZENTR. □RECHTS□ VOLL□ ←FORMATIERUNG ZEILENABSTAND →1■ 1½□ 2□

```
Fehler #1:
 Text: BsplWS2/1_com     Seite: 1    Befehle: 17     Zeichen: 276
 Fehler $08:  Befehlsstruktur unbekannt
 Hinweistext:  real=real-real

Fehler #2:
 Text: BsplWS2/1_com     Seite: 1    Befehle: 21     Zeichen: 332
 Fehler $08:  Befehlsstruktur unbekannt
 Hinweistext:  IFreal=byte

Fehler #3:
 Text: BsplWS2/1_com     Seite: 1    Befehle: 24     Zeichen: 372
 Fehler $25:  ENDIF/ELSE/LOOP ohne IF/WHILE

Fehler #4:
 Text: BsplWS2/1_com     Seite: 1    Befehle: 27     Zeichen: 385
 Fehler $23:  Programm-/Blockende mit offenem IF/WHILE-Befehl
```

The program clearly shows in the if command how scrupulously to pay attention to the correct parentheses and type declaration...

IF c = 4 ... does not work:

The argument must be in parentheses, otherwise the IF command will be recognized as incomplete (message 3, 4):

```
IF (c = 4) ...
```

A little note on parentheses: Only the contents of two variables or the value of two numbers can be linked. These should be enclosed by parentheses. Should more than two numbers be charged, z. For example, d = a + bc, an order must be given by parentheses: d = ((a + b) - c). (see message 1)

Message 2 tells us that c is a real variable, but 4 is a byte value, different types can not be compared. So we have to make sure that type conflicts do not occur. Powerful commands for the conversion of the types are provided for this purpose.

In our case, the real variable must be converted into a byte variable. This is only possible with a small detour:

c first to convert into an integer variable ...

```
IF ((INT c) = 4) ...
```

... but note that GeoCom always expects the function in parentheses, the argument is connected directly (INT c). The outer parentheses are the parameter limit for the IF command. Now the integer variable only has to be converted into a byte number.

Since 4 lies in the value range from 0 to 255, only the LOW byte of the integer number (INT c) is set. The high byte is set to zero. So we just need to filter out the LOW byte: (LOW parameter) is the default syntax. The parameter is (INT c), so we get:

```
IF ((LOW (INT c)) = 4) …
```

A compilation attempt shows that it still can not be done. A single equals sign always interprets the compiler as an assignment, a comparison is represented by a double equal sign:

```
IF ((LOW (INT c)) == 4) …
```

Now the program is running as planned!

At first, it takes a bit of effort for such a small routine. For more complex programs, the variables are chosen more meaningfully, so that the problems described here can not occur. The intention was only to show that the programming has to pay close attention to the declarable variable types and that the compiler has certain requirements for parenthesis rules.

Next time, it's about graphics commands and the problems of different GEOS systems. Have fun programming

Olaf Dzwiza

GeoCom Workshop #3

Author: Olaf Dzwiza

Before the workshop continues as planned, the supplement to the test, which was promised in GUP 32, will be briefly followed here.

At the end of June, I received the final retail version of GeoCom and can fully confirm the test result.

Although the compiler (albeit a bit hidden) was announced for the first time in early 1993, a long waiting period has passed, but the development time of two years has been worthwhile. Unfortunately, such a clean product is less and less common.

The floppy commands are error free (in complete contrast to GeoBasic), so that now non-assembler programmers can create sophisticated GEOS software.

But beware: the system is not suitable for beginners!

The object editor is comfortable and allows the creation of menus (in any variations!), Dialog boxes, file headers, sprites, bitmaps, rows of data. It’s too bad that generated menu and dialog box objects are not displayed, the routines to do it yourself inevitably switch between GeoWrite, GeoCom and ObjectEdit quite often. Anyone who has no or only a very small REU will have to invest a lot of time.

The price has remained at 55, - DM (GUC members) and 59, - DM (all others) in an order at the GUSS (see GUP 32). At least in the beginning you should turn to the direct distributor D. Döhler, because it there may still be improvements and it will be the quickest way to stay informed.

But now to the workshop - this time quite short, in the end a few more general hints.

GeoCom allows the creation of programs for GEOS 64 and GEOS 128 (40 and 80 columns), so for all modes. Where the finished code should run is defined by "STARTFLAG byte" in the definition part. Where "byte" is:

```
$00 (0): if GEOS 128, then 40 character mode
$40 (64): if GEOS 128, then 40 and 80 characters
$80 (128): only GEOS 64
$c0 (192): if GEOS 128, then 80 character mode
```

You should always write only those programs that you can test completely(!). So if you have a C64 (like myself, important for the following parts), you should stop using the 80-character screen with STARTFLAG $00 or omit GEOS 128 altogether (STARTFLAG $80) to avoid any errors.

Programs written for 40-character screens can be easily adapted to 80 characters by doubling the x value of a coordinate through DBL (double command). Example:

```
LINE 10,10,300,10
```

draws a straight line from 10/10 to 300/10 in 40-character mode. Logically, in 80-character mode, it only goes to the middle. With

```
LINE (DBL 10), 10, (DBL 300), 10
```

the screen is used completely.

A project

In the next episodes we will eventually develop a more extensive program. It should simulate dice, whereby the distribution of a long series of random numbers is graphically represented in a diagram. Of course, the graphics should then be printed and saved. It should also use menus, shortcuts and a useful VLIR structure.

Since I can only develop the project on a C64, you will always find STARTFLAG $80 in the head of the listings. For users of GEOS 128, however, it should not be a problem to adapt their programs to 40-character or even 80-character through the basics shown in this and the previous section.

Recommendation

GeoCom is not subject to any significant constraints as far as the graphical design of the applications is concerned. However, a graphical user interface only makes sense if all programmers stick to uniform guidelines (when designing).

If you are not sure about this, you should take a closer look at the standard applications such as geoWrite and geoPaint and help yourself with the creation of your own screens with hardcopies of screens of professional software. Many programs from the PD/SW sector are unfortunately wrong in this respect.

In GUP 11 and 12 (1990) is "GeoNorm", which is quite useful for this purpose and really recommended to everyone. If you do not have the booklets, you can order them from Jörg Sproß or, if only the GeoNorm articles are of interest, you can get a copy of the corresponding three GUP pages for DM 3, -.

Olaf Dzwiza

GeoCom Workshop #4

Author: Olaf Dzwiza

A dice simulation

Finally the time has come, we start with our program project (Details: GUP 35). In this episode, a series of random numbers should be created and an evaluation graph constructed.

In the main program, the variables must first be initialized, then the rest of the process is controlled from here. A data set 'dice_number' stores the distribution of numbers, an integer variable and a real number are used to calculate the average. You also need: One counter, one buffer for the number you generate and one for the converted number of dice and finally the random number function. The address $850a is rewritten with an integer value for each interrupt, so it can be used as a "random" function. Definition and main part thus look like this:

```
NAME "Würlelstat.1.0a"
CLASS"WStat    V1.0a"
AUTHOR "Olaf Dzwiza"
BYTEVAR counter, dice_value, random_number
INTVAR total_value
INTVAR AT $850a; random
REALVAR average
ROW 6 BYTEVAR dice_number
LABEL main_program
LABEL number_series, graphics_setup
@main_program
(dice_number <0>) = 0
(dice_number <1>) = 0
(dice_number <2>) = 0
(dice_number <3>) = 0
(dice_number <4>) = 0
(dice_number <5>) = 0
total_value = 0
generate_numbers
graphics_setup
WAIT 255: WAIT 255
END
```

Before you start compiling, the labels generate_numbers and graphics_setup should be included in the program text.

generate_numbers should generate 200 random numbers and increment a corresponding counter (BYTEROW dice_number) for each number. We use only the least significant byte of the "random" function and assign the following dice numbers:

```
0 - 41: 1
42 - 82: 2
83-123: 3
124-164: 4
165 - 205: 5
206 - 246: 6

@generate_numbers
counter = 0: random_number = 255
REPEAT
 WHILE (random_number> 246) DO random_number = (LOW random)
 LOOP
 dice_value = 1
 IF (random_number> 41) THEN: dice_value = 2: ENDIF
 IF (random_number> 82) THEN: dice_value = 3: ENDIF
 IF (random_number> 123) THEN: dice_value = 4: ENDIF
 IF (random_number> 164) THEN: dice_value = 5: ENDIF
 IF (random_number> 205) THEN: dice_value = 6: ENDIF
 dice_display
 INC (dice_number < (dice_value = 1)>)
 INC counter
 random_number = 255
UNTIL (counter == 200)
RETURN
```

The label dice_display is still to be integrated in the declaration section, in the program code is sufficient here temporarily:

```
@dice_display
RETURN
```

Now draw a bar chart with the results. The graphics commands probably do not need to be explained in a big way, but at most the many calculated coordinates are a bit confusing. If in doubt, but help tables and a sheet of paper. For the exact listing unfortunately the place is missing. Many things in this section of the program (number ranges, bracket rules) have already been explained in the previous workshop sections.

The most interesting thing is the INT function:

The LINE command expects the number format LINE int, byte, int, byte. But since "counter" is declared as a byte and multiplied by another constant byte value, which again results in one byte, we have to convert this value into an integer number. INT byte1, byte2 converts byte1 (low) and byte2 (high) into an integer number. Then the average is calculated and output.

```
@graphics_setup
PATTERN 0
RECT 45,40,275,175: FRAME 45,40,275,175
LINE 60,55,60,145: LINE 60,145,260,145
LINE 258,143,260,145: LINE 258,147,260,145
counter = 0
REPEAT
 LINE 58, ((15 * counter) + 62), 60, ((15 * counter) + 62)
 SETPOS 50, ((counter * 15) + 65)
 PRINT (STR (counter + 1))
 INC counter
UNTIL (counter == 6)
counter = 0
REPEAT
 LINE (INT ((20 * counter) + 60), 0), 145, (INT ((20 * counter) + 60), 0), 147
 SETPOS (INT ((counter * 20) + 58), 0), 157
 PRINT (STR (counter * 20))
 INC counter
UNTIL (counter == 10)
counter = 0
PATTERN 1
WHILE (((dice_number<counter>) <> 0) AND (counter < 6)) DO
 RECT 60,((counter*15)+60), (INT(60+(dice_number<counter>)),0),
    ((counter*15)+64)
 SETPOS (INT (64 + (dice_number<counter>)), 0), ((counter * 15) + 65)
 PRINT (STR (dice_number<counter>))
 INC counter
LOOP
counter = 0
REPEAT
 total_value=(total_value+((INT(dice_number<counter>), 0)*(INT(counter+1), 0)))
 INC counter
UNTIL (counter == 6)
average = ((REAL total_value) / 200)
SETPOS 65,170: PRINT "/BAverage:";:PRINT (STR average)
RETURN
```

And now a little task: Add the subroutine @dice_display as follows:

```
@dice_display
INCLUDE "Wüdarst_mod"
RETURN
```

"Wüdarst_mod" should be a module that simulates a dice display for every number generated.

A possible solution can be found on the **floppy disk for the workshop**, which is now available from me (price 5 DM or equivalent for a disk (1541!) With stamped return envelope, address at the end of the text).

On the disk, which I will add to each additional workshop part, you will find all source code and compiled programs.

The next part is about menu and keyboard queries.

Finally, some general information

News from the GUSS (rumors and speculation from Datex-J, no guarantee of accuracy)

- Falk Rehwagen is working on a translation of GeoCom into English, which makes this compiler one of the first German programs to go the other way than usual.
- You may be expecting an increase in memory under GeoCom (up to $8,000, instead of $6000 up to now), assuming that background storage is not in use and a REU is in place.
- First steps to a GEOS 3.0 should have been made. An unofficial version of TopDesk should already be color-capable. However, it will take some time until the final completion, as the translation and extension of GeoCom are pending.

Olaf Dzwiza

GeoCom Workshop #5

Author: Olaf Dzwiza

Menu and keyboard routines

So far our program was supposed to consist of the sections @main_program (only two WAITs and one END - this will change this time), @generate_numbers, @graphics_setup and @dice_display (from module or the main source text; anyone who had problems with it can get the solution from me) and the definition part at the beginning.

In this issue we want to develop a menu structure and for the first time use ObjectEdit (please create a file "Workshop5_obj"). For this we need:

1. A main menu. Identifier: h_menu, alignment: horizontal unlimited, number of entries: 2 (0 to 1)
   a) entry 0: text: "GEOS", call submenu m_geos
   b) entry 1 : text "OPTIONS", call submenu m_opt
2. The submenu GEOS, identifier: m_geos, alignment: vertical, number of entries: 1
   a) entry 0: text: "Info", call routine r_info
3. The submenu OPTIONS. Name m_opt, alignment: vertical, number of entries: 5 (0 to 4)

   a) entry 0 : text: "new simulation series", call routine r_sim
   b) entry 1 : text: "load", call routine r_load
   c) entry 2 : text : "save", call routine r_save
   d) entry 3 : text: "print", call routine r_print
   e) entry 4 : text: "end", call routine r_end

Gently experiment a bit with the settings, the orientations and positions. But please use the above information to proceed. The distance between two subordinate menu items must be exactly 14 pixels, otherwise it will look less beautiful on the screen afterwards.

The x-position settings will require some patience. Before you can also view the menu, our program has to be supplemented a bit:

1. For the main_program label:

Instead of the WAITs, the END and the two routine calls, other commands must be indicated here. The variable assignments must be moved.

The program section now looks like this:

```
@main_program
MENU h_menu, 0
MAINLOOP
```

2. In the definition part, the generated object file has to be integrated:

```
OBJFILE "Workshop5_obj"
OBJECT m_opt, m_geos, h_menu
```

Important: Integration must take place from the lowest menu to the highest level, never vice versa.

Of course, all used routine calls must still be made available before they can be used:

```
LABEL r_info, r_sim, r_load, r_save, r_print, r_end
```

First of all, you have to enter every label

```
@labelname
FIRSTMENU
RETURN
```

to dismiss the open menu and jump back to MainLoop. Attention! The label @r_end must necessarily be:

```
@r_end
FIRSTMENU
END
```

Otherwise, the program can only be left via a reset button, which you should have as a programmer, but that is not the best way.

If you are now satisfied with your menu after compiling, we can go ahead and fill the routines with life:

```
@r_info
FIRSTMENU
STRNBOX Program information "," "," "
RETURN
@r_sim
FIRSTMENU
(dice_number <0>) = 0
(dice_number <1>) = 0
(dice_number <2>) = 0
(dice_number <3>) = 0
(dice_number <4>) = 0
(dice_number <5>) = 0
total_value = 0
already_diced = 0
PATTERN 2
REGT 0,15,319,199
generate_numbers
graphics_setup
RETURN
```

The other routines will be added later. To keep some overview, the program should be structured like this:

```
Definition part
@main_program
All "@r _..." routines from top left to bottom right in the menu layout
@generate_numbers
@graphics_setup
@dice_display
```

Now only the keyboard query for short cuts is missing:

First, the object document is opened, then the object "m_opt" visited and behind every menu name separated by a comma the letter is displayed, which should be permissible as a key combination with C=. Then the distance from the left edge of the screen for the output of the keyboard shortcut is set and we are done except for a small program routine.

I have used the following keys for the commands:

```
new simulation - n
load - l
save - s
print (print) - p
end (quit) - q
```

Insert in the source text before the menu command

```
ON 0 GOTO keyrout
```

and define @keyrout as a label. Here it is bound to a keystroke:

```
@keyrout
IF (keydata == 238): r_sim: ENDIF
(keydata == 236): r_load: ENDIF
IF (keydata == 243): r_save: ENDIF
(keydata == 240): r_print: ENDIF
IF (keydata = = 241): r_end: ENDIF
RETURN
```

The comparison value after in the IF statement is the respective key code plus 128 for the Commodore key. The routine name after that is a GOSUB call of the respective function.

A little plastic surgery in the section @graphics_setup should still be done. The line beginning with SETPOS 65,170 should now look like this:

```
SETPOS 65,170: PRINT
"/BAverage:";: PRINT
(STR average) ;: PRINT
"/P"
```

That was all for this edition. Next time we will integrate resource management and write a print routine.

If you have problems with one or the other part, feel free to write to me (do not forget return postage, thank you!) Or contact BTX (A-side in * geos #).

Otherwise, of course, there is still the supplemented workshop disk from me; for more see GUP 35.

Olaf Dzwiza

GeoCom Workshop #6 - Something about tools and printers

Author: Olaf Dzwiza

On to printing!

Before that, however, a general note: If you print under GeoCom, you must know that it needs a lot of variable memory (exactly 2560 bytes = 2.5 kbytes). How to save space here by outsourcing to modules, should occupy us later.

We need: 1920 bytes for the printer driver as additional memory and 640 bytes buffer for graphics data. So we define:

```
ROW 640 BYTEVAR data_buffer
ROW 1920 BYTEVAR driver_buffer
```

Everything else follows now from the label r_print between FIRSTMENU and RETURN: First of all we need a query, if data already exists. In the case of "data available", the user gets a possibility to prepare the printer: STRNBOX "/BPlease turn on printer", "and load paper/P"."" Otherwise an appropriate error message is issued.

If there are no problems, the printer driver is searched for and initialized on the current drive:

```
PRINTINIT driver_buffer
```

Advanced users can write additional code to search all active drives. However, this is not typical in GEOS. I will not do a closer execution, because the basics of floppy programming are missing.

If the loading was error-free, the printing follows, otherwise a corresponding message. The graphics commands are there to hide the unneeded screen parts, as we only make a screen hardcopy:

```
@r_print
FIRSTMENU
IF (already_diced == 1) THEN
  STRNBOX "/BPlease turn printer on", "and load paper./P", ""
  PRINTINIT driver_buffer
  IF (iostat == 0) THEN
    PATTERN 0
    RECT 0,0,319,39: RECT 0,176,319,199
    RECT 0,0,44,199: RECT 276,0,319,199
    START PRINT
      HARDCOPY 0, (128 + 20), 25, data_buffer
    STOP PRINT
    PRINT DONE
    PATTERN 2
    RECT 0,0,319,199
    graphics_setup
    MENU h_menu, 0
    IMPRINT 0,0,74,14
  ELSE
    ERROR
  ENDIF
ELSE
  STRNBOX "/BFirst a number series", "must be generated!/P", ""
ENDIF
RETURN
```

The numbers in "HARDCOPY" have the following functions:

```
 0: Print from the 0th screen line

(128 + 20): 128 equals bit 7 set - if so, then bit 0 to bit 6 is distance in pixels
(here: 20 - % 0010100), so this number is responsible for the left margin.

 25: lines to be printed
```

Help - or the search for the DeskAccessories

First we need three variables:

```
BYTEVAR pos, last
ROW 8 STRVAR 16; tools
```

"pos" is the position of the selected tool in the GEOS menu, "last" is the last existing program and "tools" is the list of the first eight DAs on the boot drive of our program.

First of all you have to search for this tool (insert directly after @main_program):

```
FCLASS "" 5,8,tools
("”= no specific file class, 5 = GEOS filetype,
8 = Max. Number of files to be searched)
last = backbyte
```

Then correct the length of the GEOS menu if less than eight DAs are found:

```
POKE ((ADR m_geos) + l), (((9-last) * 14) +15)
POKE ((ADR m_geos) +6), ((9-last) OR% 10000000)
```

Here you might think ... right! The object "m_geos" has yet to be added, so that the POKE commands also fulfill their purpose.

"m_geos" must now have 9 entries, while entry 0 is already known from part 5. Entries 1 to 8 contain the file names found with FCLASS, so it must be named "tools, position" (see picture), "position" is the length of the names in the string: 16 characters plus zero bytes per entry, I.e. 0, 17, 34 etc.

The routine is called @help (define !!!):

```
@help
pos = menunum
FIRSTMENU
GETFILE (tools<(pos – 1)>), 0, curdrive, ""
RECOVER 0,0,319,199
CLRCOL
IF (iostat <> 0) THEN
  ERROR
ENDIF
RETURN
```

GETFILE loads the file whose position was selected by pos/menunum by user from current drive (curdrive); RECOVER restores the screen after leaving the DeskAccessory, the error routine protects against floppy disk problems.

Now we have the program compiled and tested and discover yet a small flaw: The menu is no longer visible after calling a tool (but still available!).

Quite simple: RECOVER retrieves data from the background memory, but the menu structure is only drawn to the foreground. So after the MENU command in @main_program we have to save the menu itself in the background:

```
IMPRINT 0,0,74,14
```

The last two coordinates are examples. To adapt to your own menu bar, they indicate the right bottom corner of the menu.

So, next time it's about VLIR structures. The workshop disk has been updated once again and is available for a fee.

Olaf Dzwiza

GeoCom Workshop #7

Author: Olaf Dzwiza

VLIR and overlay modules

Every major GEOS program is stored in a so-called VLIR structure. Without going into the floppy format of this structure, the meaning should be familiar to every GEOS user:

Many programs are so big that they do not fit in the memory in one go. Therefore, only a main part, which is needed for the control and query, is always loaded and then depending on the user command a program part, which is usually referred to as a "module", is reloaded; it also contains the necessary commands.

Memory is scarce under GeoCom. Only 14 kbyte of memory are available for program, variables and constants if the background memory from $6000 is to be used as such. Although our program, the dice simulation, could be loaded into memory all at once, it should be shown here, how such a division into different modules is possible.

There are basically three commands: GLOBALEND, OVERLAYMOD and GOTOMOD. In detail they have the following tasks:

GLOBALEND is an end marker. Everything that stands before this command is constantly in memory. If you do not use any tricks, it is menu, keyboard, screen area, and other calls will load the necessary module.

OVERLAYMOD marks the end of a module that is being reloaded as needed. This loading technique is called overlay technique, hence the name of this command. The last module of a program does not need to contain this command because OVERLAYMOD also prepares a record for the next program block on the disk.

GOTOMOD Finally, a jump command to reload an overlay module.

The use of each command will be explained below. First a few special features:

1. After starting a GeoCom program with VLIR structure, module 0 is loaded first, this is the first module after the resident main part. This is the best way to initialize all the variables and build the screen so that the main program is relieved of it. This module absolutely needs a jump command GOTO to a place in the global part.
2. Each GeoCom program is stored as a VLIR file, even if it has only one, sequential structure. One record then contains the sequential program, a second the GeoCom basic routines.

3. Suppose that two overlay modules (0 and 1) are used. The compiler now shows "3" in the status line at "Mod ..:". The third module is the main program, the resident part. In turn, there are four data sets on the disk, plus the GeoCom routine library. So do not let it confuse you.

But now to our statistics program. The global part should contain the menu handlers, the keyboard routine and the help system. The menu query, however, should only contain jump addresses to the respective modules to be loaded. This looks like this in detail (omissions are indicated by (...) and already known from the other parts):

```
(...)
BYTEVAR already_diced, pos, action
(...)
ROW 8 STRVAR16; Tools
(three lines omitted)
LABEL main_program, keyrout
(one line is omitted)
LABEL r_info, r_sim, r_load, r_save, r_print, r_end
(...)
@main_program
(...)
@r_info
(...)
@r_sim
FIRSTMENU
action = 1
GOTOMOD 1
RETURN
@r_load
FIRSTMENU
GOTOMOD 2
RETURN
@r_save
FIRSTMENU
GOTOMOD 3
RETURN
@r_print
FIRSTMENU
action = 2
GOTOMOD 1
RETURN
@r_end
(...)
@keyrout
(...)
@help
(...)
GLOBALEND
```

Module 0 will be used for program initialization. Module 1 contains the routines for simulation and printing, Module 2 will be added together with Module 3 only in the next section, where the floppy disk commands are.

The subsequent transformation proves to be less clear and in practice also not done. You will first design a concept and outline, then set up the modules right from the start. However, this was not possible during the workshop, as some basics should be considered first.

What is the new variable "action" for?

It should be a flag variable: which command in module 1 is to be executed, simulation or printing. Of course, one could also use a separate module for each routine, but it will be explained below why this makes more sense.

The main part would be complete, so we turn to the module 0 - the program initialization:

First, the variable "last" must be declared. It used to be in the main body, but is no longer needed there. Then the GEOS menu is set up (these are the first four lines of the program block) and a few variables initialized - so nothing new, only seen in another form:

```
BYTEVAR last
FCLASS "",5,8,tools
last = backbyte
POKE ((ADR m_geos) + 1), (((9 - last) * 14) + 15)
POKE ((ADR m_geos) + 6), ((9 - last) OR %10000000)
(dice_number <0>) = 0
(... etc to index "5" ...)
total_value = 0
already_diced = 0
GOTO main_program
OVERLAYMOD
```

"GOTO main_program" jumps into the program loop that controls everything, and OVERLAYMOD ends this part. So now Module 1 follows:

Here we first meet a few known variable and label definitions, then the "action" value is evaluated and the known routines follow:

```
BYTEVAR counter, dice_value, random_number
INTVAR AT $850a; random
ROW 640 BYTEVAR data_buffer
ROW 1920 BYTEVAR driver_buffer
LABEL simulation, printing, number_series,
      graphics_setup, dice_display
IF (action == 1) THEN: GOTO simulation: ENDIF
IF (action == 2) THEN: GOTO print: ENDIF
@simulation
(... here is the program code from @r_sim after the
     Command FIRSTMENU, see GUP 38 ...)
RETURN
@printing
(... here is the program code from @r_print
     the command FIRSTMENU, see GUP 38 ...)
RETURN
@generate_numbers
(...)
@graphics_setup
(...)
@dice_display
(...)
OVERLAYMOD
```

Now it becomes clear why the simulation and the printing come in one module: Both need the same graphics subroutines. Of course, you could now implement the @graphics_setup section twice, but that would be waste of disk space. Also, this label could be in the main program, only it is advisable to make that always as short as possible.

Professionals will not define a label for each command in the body, but will calculate the jump address through GeoCom's menu variables. However, this would go too far here.

We will make a short preparation for the next part, then everything is ready for this edition:

```
Overlay module 2 - Loading saved data
RETURN
OVERLAYMOD
Overlay module 3 - Saving data series
RETURN
```

Through this command sequence we have prepared the modules 1 and 3 on floppy disk. So that's it for now. Have fun programming.

By the way: The workshop disk is updated again.

Olaf Dzwiza