

BASIC-128

FOR THE COMMODORE 128

The *complete* BASIC compiler
and development system



BASIC

128

Make your BASIC
programs run
LIGHTNING SPEED!



ADVANCED DEVELOPMENT PACKAGE

A = CODE-GENERATOR:	P-CODE
B = LOAD SYMBOL-TABLE:	OFF
C = SAVE SYMBOL-TABLE:	OFF
D = LINE-ADDRESS-TABLE:	OFF
E = MEMORY	
F = COMPILER:	SLOW
G = RUNTIME-MODULE:	ON
H = EXTENSION:	OTHER
I = ENVIRONMENT	
J = TRACE:	OFF
K = TRAP:	OFF
L = OVERLAY:	OFF
M = DISK-COMMAND	
N = DIRECTORY	
O = OPTIMIZER:	I

A DATA-BECKER PRODUCT FROM Abacus  Software

BASIC 128

The complete BASIC compiler
and development system

By Thomas Helbig

165793

A Data Becker Product

Published by

Abacus  Software

P.O. Box 7211
Grand Rapids, MI 49510

Copyright Notice

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

Warranty

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

Second Printing, January 1986

Printed in U.S.A.

Copyright © 1985

Copyright © 1985

Data Becker GmbH
Merowingerstr. 30
4000 Dusseldorf, West Germany
ABACUS Software, Inc.
P.O. BOX 7211
Grand Rapids, MI. 49510

ISBN 0-916439-53-4

Preface

BASIC 128 is an optimizing BASIC compiler for the Commodore 128 that makes your programs faster and more efficient. BASIC 128 has all of the options of the well-known BASIC 64 compiler and contains additional important new features, such as complete compatibility with BASIC 7.0 and an improved code generator.

The P-code generated by BASIC 128 is up 15 times faster and, more importantly for longer programs, up to 30% shorter than an uncompiled program. BASIC 128 can also compile your program into super-fast machine code for speed increase of up to 35 times. You can also switch between the two types of code in your program. In addition to this, BASIC 128 supports the FAST mode of the Commodore 128, which is used to double the speed of compiler.

In addition to the efficient optimizing feature of the compiler, the run-time system contains a function so that programs that cannot be optimized during the compilation can still be accelerated by four to ten times. This is noticeable in programs with many floating-point variables and strings. Programs with integer data are automatically accelerated by the optimizing function of the compiler.

The floating-point functions TAN, ATN, SIN, COS, ^, SQR, EXP, and LOG were slightly accelerated by other compilers such as BASIC 64. BASIC 128 uses routines previously used only by the floating-point processors of considerably more expensive computers to calculate these functions. These functions are accelerated on the average by a factor of five (factor of 11, maximum). This makes BASIC 128 indispensable for scientific programs. These functions also speed up complex graphic operations.

BASIC 128 compiles programs of any size and also allows overlay packages. The speed of the compiler is about 1-2 Kbytes per minute, depending on the type of drive. The entire memory space not occupied by the compiled program can be used for data storage.

Depending on the program size, this leaves data storage of from 60K to over 100K.

BASIC 128 offers you many more features, such as two levels of optimization, a variable code starting location, variable memory usage, automatic linkage or disabling of the floating-point module, redefinition of the data types of variables, calculation of constant expressions and strings during compilation, optimization and transposition of formulas, syntax checking, creation of a line-address list, data interfaces to assemblers, user-friendly operation, warnings on errors, restart the compiler without reloading, etc.

BASIC 128 together with the built-in BASIC 7.0 interpreter forms an ideal program development system for the Commodore 128. Fast and efficient programs can be written without programming in machine language.

Thomas Helbig
October 1985

Table of Contents

Chapter 1: The BASIC 128 Compiler	1
Chapter 2: Survey of Options	3
2.1 Options	3
2.2 The advanced development package	3
2.3 Compiler directives	4
2.4 A quick example	5
Chapter 3: The optimizer	9
3.1 The general operation of compilation	9
3.2 Optimizing formulas	12
3.3 Processing strings	13
3.4 Integer optimization	14
3.5 The machine code generator	17
3.6 Run-time optimizations	19
Chapter 4: Details of the Compiler	21
4.1 The operation of the compiler	21
4.2 Error messages	22
4.3 The Line List	27
4.4 Fast floating-point calculations	28
4.5 Array dimensioning	30
4.6 Direct mode commands	32
4.7 Integer loops	32
4.8 Special handling of certain BASIC 7.0 commands	33
4.9 Loading and saving the compiler settings	36
4.10 Restarting the compiler and compiled programs	36
4.11 The FAST mode	37
4.12 Interrupting compiled programs	38

Chapter 5: Special commands of BASIC 7.0	39
5.1 Error handling	39
5.2 IF...THEN...ELSE, BEGIN...BEND	39
5.3 Undocumented commands and command options	40
5.4 BASIC extensions	41
Chapter 6: Overlays and Run-time module	43
6.1 Compiling overlay packages	43
6.2 The run-time module	46
Chapter 7: Speed	47
7.1 The optimizing levels	47
7.2 P-code (speed code) and machine language	50
7.3 The integer value-range for the POKE command	53
Chapter 8: Memory addresses and machine language	55
8.1 Memory layout	55
8.2 Memory addresses	57
8.3 Special commands	59
8.4 Symbol tables	61
Chapter 9: The features of BASIC 128 and there use in overview	65
Chapter 10: Differences between BASIC 128 Compiler and BASIC 64 Compiler	77
Chapter 11: Additional applications	79
11.1 Input/output	79
11.2 High-resolution graphics on the 80-character screen	80

Chapter 1: The Basic 128 Compiler

Although BASIC 128 is very easy to use, you should still read this manual in order to understand the compiler completely. The operation of the BASIC 128 compiler is quite simple:

- Save your BASIC program on a diskette. The disk need not be completely empty, but it should have sufficient free space (up to 400 disk blocks for larger programs). You don't have to make a backup of your program since BASIC 128 only reads it and doesn't erase it, but a backup is still a good idea.
- Insert the disk containing BASIC 128 in the drive and start the compiler with `RUN "BASIC 128"`.
- After the compiler responds with "BASIC 128 Compiler" and the version number, several options appear on the screen. Remove the BASIC 128 compiler diskette from the drive and insert the disk containing the program to be compiled.
- Press the `<RETURN>` key. This selects option number 1, which you could also have selected with the `<1>` key.
- The compiler asks you for the name of the program to be compiled. Enter the program name and press `<RETURN>`.
- The compiler then compiles your program into a P-code program. The screen will show the line number currently being processed, and following this the compiler outputs some information about the generated program. If your program contained errors, the compiler informs you of this with appropriate error messages. The compiler does not stop when it encounters an error, but continues in order to find other errors.
- When the compiler is done, it responds with "READY" and an audible beep. Press the `<N>` key in order to tell the compiler that you don't want to compile any more programs. You can restart the compiler by pressing any other key.

- If your program contained errors, these must be corrected , the program resaved and compiled again.
- If your program compiled correctly, the compiled program is on the disk. In order to differentiate it from the BASIC program, the compiler placed a "P-" in front of the program name. Start the program with RUN "P-program_name".
- This program will now run considerably faster than a BASIC program.
- If there are errors in the running program, the corresponding BASIC error messages are displayed. No line numbers will be printed. Instead the memory locations at which the errors occurred is displayed. With the help of the Line List you can find the erroneous section of the original program.
- If the program still isn't fast enough for you, there are other options for changing this (optimization level 2, generating a machine language program, using integer variables, etc.). These are explained in the appropriate chapters.
- Several compiler settings must be changed from the menus in order to compile some of the BASIC 7.0 commands (such as overlay packages). In these cases the compiler will output an appropriate warning (see 4.2).

Chapter 2: Survey of Options

2.1 Options

After you have started BASIC 128 (with RUN"BASIC 128"), the main menu appears on the screen. The four options on this menu are numbered and can be started by pressing the corresponding numeric key. The most important option of this menu is option 1 which starts the compilation of a program with the specified optimization level. The difference between the two optimization levels is described in chapter 7. You can also select this option with the <RETURN> key instead of <1>.

Option 2 loads all of the compiler settings from a disk file and displays them in a sub-menu, to which you can also get via option 3. Menu option 4 is used for compiling overlay packages. This is described in chapter 6.

2.2 The advanced development package

For developing programs with special applications, BASIC 128 has the ability to change most of the attributes of the compiled program. To do this, select option 3 from the main menu. All of the changeable values then appear on the screen. All menu options are lettered in alphabetical order. By pressing the corresponding key you can change the setting (some options move you to other menus). Exact descriptions of the individual options of this menu are found in the appropriate sections of this book and in Chapter 9. A simple example of using the advanced development package is the selection of the code generator.

After you have selected option 3 from the main menu, the type of code to be generated by the compiler appears under menu option A. The P-code option is set by default when the compiler is started. By pressing the <A> key you can instruct the compiler to compile your program into machine language. Pressing <RETURN> brings you back to the main menu.

Changing other default values is just as simple. Naturally, multiple values can be changed. For the sake of simplicity, the options of the development package are designated "A" through "O". The options of the main menu, on the other hand, are designated by numbers "1" to "4". Note that only one of the options appearing on the screen can be selected at a time.

2.3 Compiler directives

The menu options are selectable only before compilation.

In many cases, you may want to select options during compilation. You can use *compiler directives* to do this. These directives are imbedded within REM statements so that the BASIC interpreter will ignore them. But the compiler will recognize and process these directives.

In order to recognize and process the directives, each is preceded with the at-sign (@). The format for a directive is as follows:

REM@ directive

A list of all possible compiler directives is found in Chapter 9. The most important compiler directive is the following:

REM@I=variable name, variable name

This directive causes all named variables to be interpreted as integer variables by the compiler. Using integer variables allows the development of significantly faster programs.

2.4 A quick example

This sample program calculates the prime numbers among the first 10000 numbers. This program is often used as a benchmark for comparing the speeds of computers, programming languages and compilers.

```
10 REM SIEVE OF ERATOSTHENES
20 DIM Z%(10000)
30 FOR I=1 TO 10000
40 Z%(I)=1
50 NEXT
60 PRINT Z
70 FOR I=1 TO 10000
80 IF Z%(I)=0 THEN 150
90 PRINT I*2+1
100 K=1
110 IF I+K*(I*2+1)>10000 THEN 150
120 Z%(I+K*(I*2+1))=0
130 K=K+1
140 GOTO 110
150 NEXT
```

If you run this program and measure the time for it to execute, the execution time is misleading. Most of the time is spent displaying the numbers on the screen. Output to the screen is very slow. To measure the computing speed, make the following change:

```
90 Q=I*2+1
```

In spite of this change, the program requires a fairly long time to run. It will run about 7 times faster if you compile it as described in Chapter 1. We also mentioned that we can increase the speed of execution by using several compiler options:

- Rewrite the program so that only integer variables are used. This is done by appending a "%" character to each variable name. Note that the BASIC interpreter does not allow integer variables in FOR/NEXT loops, so the program must be rewritten without FOR/NEXT loops to increase the speed.

- You can save yourself this work by using the following compiler instruction:

5 REM@I=K,Q

This makes sense especially for the loop variable `I`. Since integer variables in `FOR/NEXT` loops are not allowed by the BASIC interpreter, using the `@I` compiler directive eliminates having to rewrite the program.

- In this special case, even the compiler directive is unnecessary since the program works exclusively with integers. You can then compile it with the optimization level 2 (option "O" in the development package).
- In addition you can direct the compiler to create machine code. This is done with menu option "A".
- You can also use the following compiler directives instead of the menu settings:

1 REM@M
2 REM@O2

After you have selected the generation of machine code and the optimization level 2, the program runs more than 18 times faster than the BASIC program. Noting some rules when programming and compiling pays off.

Another example of a large speed increase is in sorting data:

```
1 REM@M
2 REM@O2
4 REM QUICKSORT
5 DIM SR%(20),SL%(20)
10 DIM A%(1000)
20 INPUT"NUMBER";N
30 FOR I=1 TO N:A%(I)=RND(1)*10000:NEXT
40 TI$="000000":PRINT"START"
50 GOSUB 1000
60 A$=TI$:PRINT "DONE"
70 FOR I=1 TO N:PRINT A%(I),:NEXT:
   PRINT:PRINT"SORTING TIME=".A$:END
1000 SP=0:L=1:R=N
1010 IF R<=L THEN 1070
1015 X=A%((L+R)/2):LC=L-1:RC=R+1
1020 LC=LC+1:IF A%(LC)<X THEN 1020
1030 RC=RC-1:IF A%(RC)>X
1040 IF LC<=RC THEN HI=A%(RC):
   A%(RC)=A%(LC):A%(LC)=HI:GOTO 1020
1050 IF RC-L<R-LC THEN SL%(SP)=LC:
   SR%(SP)=R:SP=SP+1:R=RC:GOTO 1010
1060 SL%(SP)=L:SR%(SP)=RC:SP=SP+1:L=LC:
   GOTO 1010
1070 IF SP>=1 THEN SP=SP-1:L=SL%(SP):
   R=SR%(SP):GOTO 1010
1080 RETURN
```

This program runs 36 times faster when compiled and sorts 1000 numbers in 10 seconds or in 5 seconds in the FAST mode.



Chapter 3: The optimizer

When writing programs to run at maximum speed, it is useful to know how the compiler works so that you can adapt time-critical parts of your program accordingly. Alternativley you can select optimization level 2 which tries to maximize execution speed. This is described in chapter 7. On the other hand, programming tricks which accelerate interpreter programs are unnecessary because the compiler doesn't require them or performs them automatically. You should recognize these cases because they can be used to save time.

3.1 The general operation of the compiler

After working with the BASIC interpreter you should notice the following in regard to the speed of programs:

- A GOTO/GOSUB works slower than a RETURN, although both accomplish a jump within a program.
- A loop which is programmed using IF . . . THEN executes slower than a FOR/NEXT loop.
- Formulas containing constants run slower than formulas with variables; this applies to numbers and strings.
- Access to variables is faster if the variable is encountered earlier in program execution.
- The speed of the GOTO and GOSUB for forward jumps is dependent on the jump distance. For backward jumps, speed is dependent on the distance of the jump destination from the start of the program.
- Integer variables are processed more slowly than are floating-point variables. Memory can be saved by using integers only in arrays.

- For large programs, the GOTO/GOSUB and processing of variables becomes much slower.
- The program is easier to read, but is much slower, when you insert spaces in the program.
- Structured programs are usually slower than spaghetti code.
- If large arrays are dimensioned at the start of the program, the interpreter pauses several seconds each time a new variable is used.

All of these characteristics and many others are not true for compiled programs. If the program is to be compiled you don't need to adapt it for the BASIC interpreter. There are also operations for compiled programs which execute faster than the BASIC interpreter and it is often useful to know how to use these.

This leads to the question: "Why are compiled programs faster than those which are not compiled?" The most important points are:

- The work of interpreting BASIC programs is shifted to interpreting a type of pseudo machine language (P-code), which can be done significantly faster. In addition, BASIC 128 has the ability to generate true machine code which eliminates the interpretation completely.
- Interpreting of formulas is performed by the compiler. The compiled program calculates formulas using a faster procedure.
- When processing a variable, it doesn't have to be found in a table, but can be accessed directly.
- It is no longer necessary to locate the destination of a GOTO/GOSUB jump by searching through the entire program for a specific line number--the jump takes place directly. Example: A GOTO in a machine language program takes 3 microseconds, while with longer BASIC interpreter programs it can take up to half a second.

- Constants must first be converted to binary form by the interpreter since they are stored in ASCII format in the program. The same applies for the line numbers of GOTO/GOSUB. By contrast, constants are stored in binary format in compiled programs.

This has some important consequences. The following example substantiates this:

```
10 FOR I=1 TO 1000:NEXT
20 PRINT"DONE WITH FOR LOOP"
30 I=1
40 I=I+1:IF I<=1000 THEN 40
50 PRINT "DONE WITH IF-THEN LOOP"
```

When you run this program notice that the first loop is considerably faster than the second. After compiling it, both loops will be about equally fast. The speed of compiled programs is not dependent on the number of commands used or the structure of the program, but only the operations performed themselves. Internally, both loops are executed in almost the same manner (the management time for the FOR/NEXT loop is somewhat higher because it is more flexible).

The compilation procedures mentioned so far are features of most BASIC compilers. BASIC 128, on the other hand, uses additional procedures to speed up a program. This included the optimization of formulas, the use of integer operations, fast string processing, and the selectable generation of machine code.

3.2 Optimizing formulas

When calculating complex formulas, temporary results are stored on a stack. The compiler transforms the formulas with the goal of achieving the fewest temporary results possible. This not only makes the compiled program shorter, it can also be executed faster. You no longer need to take fast interpretation into account when writing large formulas. The compiler even changes formulas in order to allow faster calculation. The multiplication of two floating-point numbers is faster than division, for instance. In many cases it is possible to replace floating-point division by multiplication (inverse). The compiler uses such procedures only when so doing will preserve the result of the formula, of course.

One thing which can make it easier to write mathematical programs is the ability of BASIC 128 to calculate formulas which contain many constants. The following example illustrates this:

```
10 S=1.414
```

The assignment is not very accurate, but will be processed faster by the interpreter than

```
10 S=SQR(2)
```

After compilation, both versions will be equally fast, but the second version is more accurate and easier to program. The compiler also calculates constant operations when they are contained in larger formulas. Hidden constant operations can often be discovered by transforming the formula and these can then be calculated.

One case in which the calculation of a constant before compilation is not possible is strings. Not all 256 possible character codes can be represented within the quotation marks. In spite of this, these codes are required quite often, such as when working with the disk drive.

The access of a memory location in the disk RAM might look like this:

```
PRINT#2, "M-R"+CHR$(0)+CHR(4)+CHR$(10)
```

The calculation of this string is not only very long, it is also slow. But in compiled programs, it occupies only the required 5 bytes and need not be computed because the compiler did this already.

The fast math functions of BASIC 128 result in a significant increase in speed. But this is not an optimization and is explained in section 4.4.

3.3 Processing strings

Have you ever had a program suddenly stop and appear to do nothing for several seconds and then the program continues normally? The reason for this peculiar "behavior" lies in the memory management of the BASIC interpreter. Each time a string is assigned to a variable, the previous contents of the variable are not erased, but remain in memory. When all of the memory is exhausted, a routine is then started to search for all of the unused strings in memory and remove them so that the program can continue (garbage collection). The compiler uses a faster routine to do this than does the interpreter. Even in extreme cases, the garbage collection of a compiled program will take no more than one second and the resulting program delay minimal. If this still takes too long for you, the garbage collection can be controlled by the program and can then be executed in the least time-critical sections of the program. This is done by using the FRE (1) function.

BASIC 128 performs additional optimizations in regard to faster execution of string operations. This is very effective especially for complex string formulas, such as:

```
A$=LEFT$(A$,X%-1)+B$+MID$(A$,X%)
```

The string B\$ is inserted into the string A\$ at the X%th place. The BASIC interpreter would first generate several partial strings and then append these. The compiled program is optimized so that it calculates only the resulting string; temporary results do not arise. The combined formula executes as quickly as a single function.

3.4 Integer optimization

In Commodore BASIC there are two different data types for representing numerical values: floating-point numbers (REAL) and integers. A floating-point variable can have any name, but only the first two characters are noted by the interpreter. A variable which has a "%" as the last character of its name can only accept integers. Here too the interpreter pays attention only to the first two characters of the name and the % sign. A variable which is designated in this manner can accept only the values -32768 to +32767, without any fractional part. Although such variables are normally seldom used, this data type is significantly more important than the floating-point type. A BASIC programmer normally doesn't notice this because the BASIC interpreter converts between both data types when necessary and even often when it isn't.

Below is a list of all commands, operations, and functions for whose execution the BASIC interpreter requires only integers and makes a corresponding conversion of the floating-point representation:

ON, GOTO, ON GOSUB, WAIT, LOAD, SAVE, VERIFY, POKE, CMD, SYS, OPEN, CLOSE, TAB, SPC, NOT, AND, OR, FRE, POS, PEEK, LEN, ASC, CHR\$, LEFT\$, RIGHT\$, MIDS\$, index of arrays\$, all graphic, sprite, and disk commands or functions, etc.

For the following operations the interpreter uses the floating-point representation and converts integers accordingly:

loop counters, INPUT, PRINT, READ, IF, DEF, +, -, *, /, >, <, = <=, >=, <>, SGN, INT, ABS, USR, SQR, RND, LOG, EXP, COS, SIN, TAN, ATN, STR\$, VAL.

For the following operations, it is not always necessary to convert to floating-point format, although the BASIC interpreter does it anyway:

loop counters, INPUT, PRINT, READ, +, -, *, /, >, <, =, <=, >=, <>, INT.

In addition, the BASIC interpreter performs all calculations in floating-point format. If the operation is an integer operation, the format is converted to floating-point format before the operation and converted back to integer again afterwards.

From these lists it becomes apparent that non-mathematical programs can get by almost entirely without floating-point variables. Even in mathematical programs, the use of floating-point variables can be reduced under certain circumstances. In any case, the majority of all data type conversions of the BASIC interpreter are unnecessary. In addition, the routines the interpreter used for conversion are slow and complicated. Furthermore, floating-point operations are far slower than integer operations. Since the BASIC interpreter always uses floating-point numbers, you never notice. A BASIC program could work significantly faster if the floating-point representation were used only when necessary.

A program compiled with BASIC 128 always makes maximum use of the integer representation. The compiler does require your cooperation, however:

- The compiler cannot tell when compiling a program which data types will be used in which variables during the run-time.
- The running program can determine this, but this is always slower than during the compilation of an optimized program.

For this reason, BASIC 128 assumes that all variables are floating-point variables, with the exception of the following cases:

- Variables which are designated with a percent sign, corresponding to the designation of the BASIC interpreter.
- Variables which are converted to integer variables by an appropriate compiler instruction.

- All variables in a program compiled with optimization level II (Chapter 7).

In Chapter 2 you saw that it really pays off to use integer variables. An average of 90% of all variables in a program can be replaced by integer variables. The BASIC interpreter does not allow integer variables to be used as loop counters in a FOR/NEXT loop. You can get around this problem with the help of a compiler directive. Variables which are used often in loops (such as I,J,K) should therefore be reserved as integer variables with an appropriate compiler directive and used only for this purpose. There are many more reasons to use integer variables:

- An integer variable occupies 2 bytes of memory space, while a floating-point variable occupies 5 bytes. This advantage of integer variables is especially important for arrays, a feature which is supported by the BASIC interpreter.
- Many simple integer operations belong to the instruction set of the 8502 microprocessor. They can therefore be executed in a few microseconds. These include:

I%+1, I, I%*2, I%+J%, I%-J%

All programming languages which can be implemented through a compiler make strong distinctions between integer and floating-point variables. If you want to move up to a more efficient programming language like Pascal, it is an advantage to be familiar with this difference. There are even programming languages which work only with integers (Forth, assembly language).

3.5 The machine code generator

Basically compilers can be divided into two different classes according to the type of code they produce. One often-used method is the generation of a pseudo code (P-code), which is then processed by an appropriate interpreter. The advantage of this code lies in its brevity. Programs compiled to P-code become 50-80% smaller than the original. This is especially important because the compiler adds a collection of routines (run-time module) to each program which the compiled program needs, but which also lengthens it. The disadvantage of P-code lies in the fact that it is not as fast as a machine language program. Although the P-code generated by BASIC 128 was developed especially for the Commodore 128, it is not as fast as machine code.

An alternative to generating P-code is compiling the program into machine language. This offers the powerful advantage that the program runs at maximum speed. But it is also takes up more memory than a P-code program. Normally only larger computers have compilers which can generate machine code, because computing time is costly for these computers and memory space it plentiful.

Many BASIC programs for the Commodore 128 do not make full use of the program memory. For this reason, BASIC 128 can generate either the shorter P-code or the longer machine code or even mix the two (Chapter 7). When compiling a program to machine code, it is no longer necessary to interpret the P-code.

The microprocessor of the Commodore 128 (8502) requires between 1 and 4 microseconds (FAST mode) for the execution of a single instruction. These instructions are primitive compared to the BASIC commands, so that several instructions are needed in order to execute the equivalent of one BASIC command. In many cases, entire subroutines have to be called and executed. A machine language program is significantly faster than a P-code program only when the compilation can be accomplished with few instructions, since in these cases the interpretation of the P-code slows the program considerably. BASIC commands can be converted to few machine language instructions only when it involves simple

commands. Simple commands are recognized by their high execution speed, such as all operations with integer variables and numbers. Most integer operations can be executed with few machine language instructions. In these cases the program is not only faster because the code is not interpreted, but also because special capabilities of machine language are used directly.

In summary it can be said that a machine language program is significantly faster than its P-code counterpart only if the P-code program itself runs fairly quickly. With programs which use complex floating-point operations (SIN, SQR, etc.), the generation of machine code does not result in any speed increase worth mentioning.

The question of how the optimizing options of BASIC 128 and the generation of machine code work can naturally be answered only by trying it out. Most programs can be speeded up through clever use of all possibilities. You have seen two short examples of this in Chapter 2.

The machine code generator is turned on via option "A" in the advanced development package. After starting the compiler, press <3> in order to access the advanced development package. With the <A> you can select the generation of 6502/6510/8502 code and then return to the main menu with <RETURN>. The program generated by the compiler will have a "M-" before the name instead of the "P-" for P-code. Otherwise the operation of the compiler is the same as for the use of the P-code generator.

3.6 Run-time optimizations

● In many programs it is difficult to designate all variables as integers for the compiler. This occurs when the program was not originally intended to be compiled or when the use of variables is very complex. The compiler then can not perform very many optimizations. So that the program runs with the greatest possible speed, these optimizations are performed by the run-time system of BASIC 128. The optimizations are performed while the program is running. This method of optimization is not as effective as the direct optimization by the compiler, but run-time optimizations work on every program, even ones not adapted to the compiler. Run-time optimizations are not normally performed by a compiler and are a specialty of BASIC 128.

●

●



Chapter 4: Details of the Compiler

4.1 The operation of the compiler

After selecting the option "1" from the main menu, enter the program name and press <RETURN>, the compiler starts to compile the program (Chapter 1). The compiler differentiates between two phases called pass 1 and pass 2.

Pass 1:

In pass 1 the program is interpreted, optimized, and the corresponding code created (P-code or machine language). The compiler outputs the current line number and each command separator (colon). If the compiler finds a compiler directive (REM@) it displays an "R" on the screen. Commands or functions which are not defined in Commodore BASIC (such as BASIC extensions) display the letter "E".

Pass 2:

In pass 2 the generated code is reprocessed and completed by the compiler, the run-time module is added and the DATA lines are inserted in the program. The compiler outputs the following messages during this process:

- Data code: The data lines of the compiled program are at this memory address in bank 0.
- Object code: The program lies in this range in bank 0.
- Strings: This area in bank 1 is completely free. The compiled program uses it for storing strings.
- Extensions: If the program is to use a BASIC extension, the compiler gives the number of uses.

- Errors: If errors occurred in the program, the compiler lists these in the appropriate line and outputs a list of all lines in which errors occurred after pass 2.
- Warnings: The compiler can output several types of warnings. The number of warnings is printed at the end of pass 2.

4.2 Error messages

The error messages possible in BASIC can be divided into two groups:

Program errors:

Errors which result in the inability of the compiler to compile a command are announced during compilation. The compiler then outputs the same error message as the BASIC interpreter. The following errors are detected by the compiler:

Syntax -

Same meaning as in BASIC

Redim'd Array -

Same meaning as in BASIC

Type mismatch -

Same meaning as in BASIC

Bad subscript -

An array access has an incorrect index number in contrast to the dimensioning.

Undef'd statement -

A GOTO/GOSUB command makes reference to a non-existent line. This error is discovered in pass 2.

Out of memory -

Program and variables no longer fit in memory. This normally happens only when the memory limits of the development package are changed.

Runtime -

The compiler tries to calculate formulas during compilation as much as possible (mainly for constants). If these formulas cannot be computed, the compiler outputs this message because it involves an error which would not normally occur until the program is running and therefore cannot be determined more exactly. A example of this is division by zero:

```
10 A=1/10
```

Warnings:

The compiler warns you about some faults and program errors. The following warnings can be output:

TRACE NOT FOUND	see section 4.8
TRACE NOT USED	see section 4.8
BEND WITHOUT BEGIN	see section 5.2
ILLEGAL BEGIN	see section 5.2
ILLEGAL ELSE	see section 5.2
ILLEGAL BEND	see section 5.2
ILLEGAL OVERLAY	see section 6.1
LOAD ONLY IN OVERLAY	see section 6.1
POINTER WITHOUT BLANK	see section 8.3

System error 1 to
system error 9:

These operating system messages result from turning the disk off or other obvious operation errors. It is also possible that a disk error message occur during compilation, such as for a full disk.

System error 10:

If the C-128 runs out of memory while trying to compile your program, this error message appears.

System error 11 to
system error 19:

If the number of errors is too great, this message is printed, such as when the program being compiled is not a BASIC program.

System error 20:

This message occurs if your BASIC 128 disk is damaged, if your disk drive is not working correctly, or if you do not start the compiler from the distribution disk.

In almost all cases the compiler can continue working after discovering an error and then look for more errors.

Note: A program in which errors are present can naturally only be executed with limitations. This applies mainly to the material following an error on a line. You should correct the error and recompile the program.

Run-time errors:

Many errors cannot be discovered by the compiler because they depend on running the program. These error messages have the same meaning as for the BASIC interpreter. The following should be noted:

Out of memory -

This message means either that the string range cannot accept all of the strings or that the stack for FOR, GOSUB, and levels of parentheses is full. The use of too many variables is detected by the compiler.

Bad subscript -

The index of an array access exceeds the array limits; an incorrect number of indices will be discovered by the compiler.

Formula too complex -

This message is printed by the interpreter if a string formula is nested too deeply, which normally does not occur. A compiled program never outputs this message since deeper nesting will be processed.

Illegal quantity -

This message is printed by the BASIC interpreter even if it actually be possible to execute the command, for example:

```
10 DRAW,100,100 TO -50,+50
```

The interpreter outputs an error message because the third parameter contains an invalid value. The following command is allowed:

```
10 DRAW,100,100 TO +50,+50
```

Since the interpreter recognizes the end coordinate as a coordinate relative to the starting point, similar to the MOVSPR instruction. Negative values also have meaning and therefore are allowed by the compiler.

When converting floating-point numbers to integer values or in calculations with integers, the result may exceed the range for an integer. You should select integer variables only if you are sure that range overflow will not occur on these calculations. For reasons of speed, the compiler does not make any range checks during integer calculations. If a temporary result exceeds the integer range but the final result is a valid integer, the result will be determined correctly by the integer operations. You can make use of this for the POKE command, for instance (Chapter 7).

The input command of BASIC 7.0 has some features which may appear when the input contains errors:

Entering a string instead of a number leads to the message "Redo from start" and the input command is re-issued. The compiled program behaves in precisely the same manner.

If the input is incomplete, the program requests the additional values with two question marks (compiler/interpreter).

If the input is missing, (pressing <RETURN> with no value entered) the input command is ignored and the variables are not changed. This is a peculiarity of C 128 and C 64 BASIC which is not present on the other Commodore computers. BASIC 128 was developed especially for the Commodore 128 and therefore supports this often-used option.

If too many values are entered, the interpreter outputs the message "Extra ignored". A compiled program ignores these values without printing a message.

Disk errors:

If an error occurs when working with the disk drive, the computer announces the error and ends the compilation. The meanings of the disk error messages are explained in the disk drive manual. Messages like "READ ERROR", "WRITE ERROR", "NO CHANNEL", and "WRITE FILE OPEN" indicate a damaged disk, which should be replaced.

When working with BASIC 128 and the BASIC interpreter in general, you may have to correct a BASIC program and resave it. This is usually done with the following command:

DSAVE "@:name"

Because of an error in the 1541 disk operating system, this command will sometimes cause loss of data on the diskette. This is naturally independent of whether BASIC 128 is used or not. It is recommended that the following commands be used instead:

SCRATCH "name": DSAVE "name"

If you interrupt the compiler with the RESET button or the on/off switch, it is recommended that you use the COLLECT command on the current work diskette in order to avoid disk errors.

4.3 The Line List

If errors occur during the execution of a program, an appropriate message is printed. The program can only output a memory address at which the error occurred instead of a line number because compiled programs are no longer organized by lines (exception: see 4.8). A listing containing the memory locations corresponding to the original line numbers can be printed with the Line List option in the advanced development package. The Line List is required to determine the location of the error causing statement. The "D" option of the advanced development package is used to enable generation of a Line List. You should always generate a line list for programs which have never been tried.

Proceed as follows to use this list:

- Note the memory address at which the error occurred.
- Load the line list with `DLOAD "Z-program_name"`
- List the list to the desired location with `LIST - memory_address .`
- On the right side of the list the corresponding lines which are assigned to the memory addresses on the left are displayed. The last line contains the error. Since errors do not always occur directly at the named memory address, it may be found at the end of the previous line or the start of the following line. This applies especially to programs compiled to machine language.
- Correct the program and recompile it.

4.4 Fast floating-point calculations

BASIC 128 uses floating-point routines for the functions TAN, ATN, SIN, COS, ^, EXP, LOG, SQR which run significantly faster than the routines of the BASIC interpreter. Up to now such routines were use mainly by the floating-point coprocessors of more efficient and more expensive 16 or 32-bit computers. With BASIC 128 these routines are also available for the Commodore 128.

The calculation accuracy of the compiler functions is slightly higher than that of the interpreter. The following table lists all functions and the average number of clock cycles required to execute them as well as the acceleration over the interpreter. All values refer only to the function itself and not to the program segments to which they may belong:

	Clock cycles	Speed-up Factor
TAN	11600	4.6
ATN	8700	4.9
SIN	16600	1.7
COS	16600	1.7
X^Y	15900	3.5
EXP	9400	2.9
LOG	8600	2.7
SQR	4900	10.8
SIN and COS together	17000	3.3

As a comparison:

A simple division requires about 3000 clock cycles. The C-128 executes 2 million clock cycles per second in the FAST mode.

The functions SIN and COS are the slowest by a considerable margin. But in many cases the SIN and COS of the same angle is required. In this case there is the option of reading the cosine of the angle with the USR function after calculating the sine, which is done in a much shorter time. Naturally, the USR function must not

have been used for another purpose. No other operations may be performed between `SIN` and `USR` (not even `PRINT`).

Example:

```
S=SIN(X) : C=USR(0) : T=S/C
```

Now `S` contains the sine, `C` the cosine, and `T` the tangent of `X`.

If such a program is to be tested with the interpreter before compiling it, `USR(0)` can be replaced with `USR(X)` because the compiler is not interested in the parameter of the `USR` function in this case. The following commands must be entered before starting the program so that the function of the interpreter corresponds to the function of the compiler:

```
POKE 4633,9 : POKE 4634,148
```

The floating-point routines occupy about 3K of memory space. The compiler inserts these routines in programs only if they are actually used. If, to save memory, you want to prevent the compiler from using its own routines, it can also use the same routines as the interpreter. To do this, press the `<E>` key in the advanced development package and then `<6>`. Now the arithmetic module is disabled. You get back with `<RETURN>`.

General information about the floating-point format of BASIC 7.0:

The interpreter and compiler have a precision of 32 bits which corresponds to a decimal accuracy of about 9.6 places. Temporary results are calculated with an accuracy of 40 bits. Numbers are stored in a binary floating-point format which allows the greatest-possible computation speed. This representation also has its disadvantages and you should be aware of these:

- Fractions which are easily representable in the decimal system (such as $1/10$) cannot be represented by an exact fraction in the binary system and therefore lead to calculation errors.

- An error arising from rounding off the temporary results cannot be compared with a decimal rounding since neither the 9th nor the 10th digit is rounded but somewhere in between. The last two points are of special disadvantage for financial programs.
- Numbers which are so small that they fall below the representable value range of the exponents are rounded to zero. This is a decisive error in many cases, especially when multiplying by a large number.
- When numbers become larger, the difference between a number and the next larger number also becomes larger. For example, 1 can no longer be correctly added to the number 1E10.
- A compiled program has slightly higher accuracy than an interpreted program.

Example:

```
PRINT 3^4
```

4.5 Array dimensioning

The boundaries of arrays must be known during the compilation for the following reasons:

- Arrays can also lie in bank 0, yielding additional memory capacity. This results in complicated memory management which only the compiler can perform.
- Access to arrays should be as fast as possible and should be direct. The compiler requires the array addresses and the memory bank to do this.

In some cases, the boundaries of an array are not known during compilation, such as with the following DIM command:

```
10 INPUT "DIM A(N)";
```

In this case the compiler expects the array name followed by a parenthesis and a question mark. After the maximum size of the array. In our example this could be the maximum value of X. With multiple dimensions, the compiler asks only for the unknown values. Separate the values by pressing the <RETURN> key between them.

Although the compiled program will usually have enough memory available to it, the question of the maximum array index is often also a question of the memory space required. In Chapter 8 you find a list of the various data types and their memory requirements.

When writing programs which work with many arrays, the problem of the interpreted program having less storage space (bank 1) than the compiled program (bank 1 and part of bank 0) occurs. In these cases it makes sense to test the program with smaller arrays and then compile it:

```
10 N=5000 DIM X%(N)
```

When compiling, the compiler asks for the maximum index of the array, allowing you to use a value larger than N.

4.6 Direct mode commands

Some BASIC commands cannot normally be used within a program or do not make sense there. The following are some of these commands:

```
LIST, SAVE, VERIFY, CONT, DSAVE, DVERIFY,  
AUTO, DELETE, RENUMBER.
```

The commands DSAVE, SAVE, DVERIFY, and VERIFY are correctly compiled by the compiler. It is not possible for a program to save itself, however, since a compiled program moves and changes itself in memory.

The remaining commands do not make sense in a compiled program and are ignored by the compiler. A compiled program can no longer be listed, for example, because it is not a BASIC program any more. The compiler outputs the message "DIRECT MODE ONLY" in these cases.

4.7 Integer loops

Integer variables may not normally be used as counters in FOR/NEXT loops, but this is possible with the BASIC 128 compiler. Integer loops are not only faster, they occupy less stack space and can therefore be nested more deeply. In addition, access to the loop variable within the loop is faster. The following example program uses an integer loop, but can still be executed by the interpreter:

```
5 REM@I=I, J  
10 FOR I=1 TO 100:PRINT I  
20 FOR J=1 TO 50: NEXT J, I
```

A STEP value can be specified for integer loops just as with normal loops, but extreme care should be taken, because the STEP value will be converted to an integer value. The value 0.5, for example, leads to the STEP value zero and therefore to an endless loop.

4.8 Special handling of certain BASIC 7.0 commands

When compiling a BASIC program into P-code or machine language, all information about the line and command structure of the program is lost. The run-time system can no longer determine which line is being executed and where a command begins and ends exactly. Some commands of BASIC 7.0 require precisely this information, however, and are therefore not normally compilable. BASIC 128 can compile these commands, however they require special handling. To handle these certain commands the BASIC 128 compiler uses either or both of the following:

LINE RECORD or COMMAND RECORD

The following commands require special handling in order to compile programs that use them:

1) **TRON:**

Requires a LINE RECORD and COMMAND RECORD in the range to be investigated.

2) **RESUME and RESUME NEXT:**

Requires a COMMAND RECORD in the range in which the error occurs.

3) **RESUME constant line number:**

record unnecessary.

4) **EL variable**

Requires a LINE RECORD for the range in which the error occurs. The EL variable can also be read after program termination through an error. In this event you don't need a line list.

5) COLLISION:

Requires a **COMMAND RECORD** in the range in which the command is to be effective.

6) Other commands with a variable line number, such as TRAP N*10:

Requires a **LINE RECORD** in the range in which the possible line numbers lie.

Exception: **RESTORE** never requires a command record.

There are several ways to have the compiler create a **COMMAND RECORD**. The simplest way consists of inserting the following compiler instruction in the first line of a program:

```
1 REM@TLC
```

This compiler directive avoids all problems which could occur with the commands mentioned above. This directive does make the program longer and slower. **BASIC 128** has the ability to control the structure record more flexibly. You can achieve the following settings with multiple presses of the <J> key in the advanced development package:

```
TRACE: OFF                ; no record, normal state
TRACE: LINE                ; only LINE RECORD
TRACE: COMMAND             ; only COMMAND RECORD
TRACE: LINE & COMMAND     ; both records
```

You can also use compiler directives to control the recording so that a record is made of only the parts of the program where it is required:

```
REM@TB or REM@TLC        ; both records
REM@TL                    ; only LINE RECORD
REM@TC                    ; only COMMAND RECORD
REM@TO                    ; turn record off
```

All compiler directives become effective immediately after their occurrence in the program.

Example:

```
10 TRAP 1000
20 FOR I=-5 TO 5
30 PRINT 1/I
40 NEXT
50 END
1000 PRINT "NOT DEFINED":RESUME NEXT
```

This program uses the command `RESUME NEXT`, for whose execution the run-time system needs a command record in line 30, because the error occurs in line 30 (division by zero). You can enable the command recording in the advanced development package, for example. Within a larger program, the following compiler directives would be more effective:

```
25 REM@TC
35 REM@TO
```

Note: A line record can replace a command record if there is only one command per line in the selected range.

The compiler warns you of an incorrect use of the structure record:

```
"TRACE NOT USED":
```

The record is not required (except perhaps for `TRON`) and should be turned off in the whole program.

```
"TRACE NOT FOUND":
```

No record directive was inserted in the program, although the program contains one or more commands which require a record. `TRON` is not checked.

4.9 Loading and saving the compiler settings

All compiler settings which can be set with the advanced development package and its sub-menus can be saved in a disk file and loaded again. This is done with option "I" in the advanced development package. Press the <S> key for save or <L> for load and enter the filename. To save a file which can also be loaded with option "2" from the main menu, the name "B128" must be given. To distinguish it from other files, BASIC 128 puts "E-" in front of each file of compiler settings, but you should not type in this "E-".

4.10 Restarting the compiler and compiled programs

After a compiled program has been started, it moves itself to the memory location for which it was generated by the compiler. After the program is done, it is still in memory, but can no longer be started with RUN and must be reloaded. There is a way to restore a program without loading it. This is the function of the program "START" on the BASIC 128 diskette. Simply insert the disk with the program "START" and enter RUN "START". The compiled program currently in memory will be started. Naturally, you can also copy the commands in the program "START" and enter these directly.

The compiler can also be loaded with the program "START" The following conditions must be valid:

- The BASIC 128 distribution disk is in the drive.
- The compiler has already been used and correctly ended (not with RESET or the on/off switch).
- After using the compiler, no compiled program was started, else the compiled program will be started.

- Before restarting the program, a BASIC program may be loaded, changed, and saved again. This program may not significantly exceed a length of 15K (60 disk blocks), or the compiler will be overwritten. When using the graphics area the program may have a length of only 6K.

If these conditions are not met, the compiler must be started as described in Chapter 1.

4.11 The FAST mode

The `FAST` command of the C-128 doubles the speed of the computer. Naturally this command also doubles the speed of a compiled program so that the speed ratio between the compiler and interpreter remains intact. If a compiled program is started with the 80-column screen active, the program automatically switches to the `FAST` mode. This can be retracted with `SLOW` command.

The compiler itself can operate in the `FAST` mode which is automatic when using the 80-column screen. When using the 40-column display, the compiler must be switched to the `FAST` mode with the "F" option in the advanced development package. In this case the compiler enables the screen only for important messages. The compilation speed is not quite double by the `FAST` mode because the compiler speed is largely dependent on the speed of the disk drive used.

4.12 Interrupting compiled programs

Interrupting compiled programs with the STOP key is only rarely possible, such as when using some input/output commands. It is not possible to continue with CONT. The TRAP command can be used to prevent a program from being interrupted. Note that the TRAP command is inactive within an interrupting routine.

Interrupting a program with the STOP key can be prevented with the POKE command, given below. This also disables the RS-232 interface:

```
POKE 792,51: POKE 793,255
```

Chapter 5: Special commands of BASIC 7.0

5.1 Error handling

The `TRAP` command of BASIC 7.0 is used to trap errors. This is not supported by BASIC 128 in all cases. For example, some errors discovered by the compiler are taken care of so that they do not enter into the program flow. In addition, some error messages no longer occur after compilation so that parts of the error handling routines become superfluous (see also section 4.2).

Programs in which the commands `RESUME` and `RESUME NEXT` (not `RESUME line number`) are used must be compiled with *command records* enabled (see section 4.8).

5.2 IF...THEN...ELSE, BEGIN...BEND

The interpreter does not always handle the structured instructions `ELSE`, `BEGIN`, and `BEND` in the way the programmer and the program structure intended. For reasons of compatibility to the interpreter, the compiler is forced to behave in the same manner. In such cases the compiler outputs the following warnings:

`BEND WITHOUT BEGIN:`

A `BEND` does not refer to a `BEGIN` and is therefore superfluous or erroneous.

`ILLEGAL BEGIN:`

In the same line before the `BEGIN` there is a `THEN` or an `ELSE` which no `BEGIN` follows. The two structures are then not nested but intersecting, which is a gross program error in almost all cases.

ILLEGAL ELSE:

An ELSE refers to two or more previous IF . . . THEN instructions. In this case the part after the ELSE is always executed if one of the two IF conditionals is false. This behavior is almost never the intention of the programmer, however. The correct nesting can be achieved with BEGIN . . . BEND.

ILLEGAL BEND:

A BEND does not show its effects until the end of the line or the next ELSE. A BEND which is not just before a line end of an ELSE is incorrectly placed.

5.3 Undocumented commands and command options

BASIC 7.0 contains some commands and options for commands which are not mentioned in the Commodore 128 manual. In addition, some commands do not behave as indicated in the manual. In these cases BASIC 128 behaves like the BASIC interpreter. Despite intensive research and investigation, there may still be some undocumented possibilities in BASIC 7.0 which the compiler does not support.

The most important undocumented cases:

- Almost all graphics commands can work with relative coordinates (see also section 4.2), as is possible with MOVSPR, for instance.
- Graphic coordinates can be specified in distance and angle by using a semicolon instead of a comma.
- MID\$ may be used on the left of the assignment character.
- SYS has the same parameters as the RREG command (see also section 8.3)

Some commands of BASIC 7.0 do not behave correctly or according to the intentions of the programmer. Where this is necessary to maintain compatibility, the compiler follows the

incorrect behavior of the interpreter (see also 5.2), though this cannot be guaranteed in all cases.

5.4 BASIC extensions

If a program containing commands which do not come from BASIC 7.0 is to be compiled, the compiler can be informed of this via the option "H" of the development package. Additional compiler settings are also necessary. The manual for your BASIC extension may contain the necessary information. But you still may have to try to compile each extension command to check for proper operation.

The run-time system of BASIC 128 makes the following procedures available to adapt a BASIC extension to the compiler:

Functions:

An extended function token calls a run-time system routine at \$ACE in bank 0. The second byte of the token is found in the accumulator. The integer value of the function parameter is found in memory locations \$47, \$48; the result is also expected there.

Commands:

If an extended command token occurs in a program, the run-time system passed control to a routine which starts at address \$AC6 in bank 0. The BASIC extension can fetch command parameters with the usual interpreter routines since these are appropriately redirected by the compiler. In addition, the CHRGET or CHRGOT routine can be used.

The BASIC extension is responsible for supplying memory locations \$ACE and \$AC6 with appropriate routines.



Chapter 6: Overlays and Run-time module

6.1 Compiling overlay packages

In Commodore 128 BASIC 7.0 you can load and start a program with the `LOAD` and `DLOAD` commands. In these cases, all previous variable contents remain intact. This version of the `LOAD` command is always executed when the command is used within a program and results in a warm overlay. The command `RUN "name"`, by contrast, uses a cold overlay and causes all of the variables to be erased.

Cold overlays can be compiled as individual programs, but the run-time module may not be disabled because programs without run-time modules cannot be started with `RUN`. The command `RUN "name"` can also start an uncompiled program from a compiled program.

Compiled programs which use the warm overlay mechanism are not compiled in the usual manner. To compile a program from the overlay package the compiler requires a symbol table in which all of the variables used in the program package and the memory layout is listed. Overlay pass 1 serves to create this symbol table (not to be confused with compiler pass 1).

Before the execution of overlay pass 1, the file `"S-OVERLAY"` must be deleted from your work disk, provided it is present.

Overlay pass 1:

Compile all of the programs in the program package individually. Before starting the compilation with `<1>` or `<RETURN>`, press `<4>` (overlay) and then `<1>` (pass 1). All options of the advanced development package (menu option 3) can also be used. The compiler does not create a finished program and for this reason runs faster. It generates just the table of variables. In addition, the compiler saves all compiler settings and reloads them for overlay pass 2. It is important that the first program compiled also be the

starting program of the overlay package because this program will be the only one which can be started from the direct mode. Furthermore, the start program can only be started with `RUN "name"` and not with `DLOAD "name"` from another program. If necessary, an appropriate program must be written for the overlay package, such as a menu program.

After the symbol table is complete, the programs in the overlay package can be compiled. Overlay pass 2 does this.

Overlay pass 2:

Compile all of the programs in the package, but first press the keys `<4>` (overlay) and `<2>` (pass 2). The compiler then loads the corresponding overlay table and the compiler settings. After all programs have been compiled with overlay pass 2, the package can be started.

The following must be noted when compiling overlay packages:

- The compiled programs must be renamed to the appropriate names because the compiler puts "P-" or "M-" in front of the name of the compiled program.
- If larger overlay packages are to be compiled, the individual programs must be divided among several diskettes because the compiled programs need disk space. The compiler loads the overlay table before the input of the program name. The table can therefore be loaded from a disk other than the disk containing the program to be compiled. It is important only that in pass 1 the last-generated table is loaded (last compiler pass). In overlay pass 2 the last table generated from overlay pass 1 is always loaded.

A command can be sent to the disk drive with the menu option "M". In this manner you can delete an already-compiled program (naturally only programs of which copies exist) in order to create space on the disk.

- All arrays must be dimensioned in the start program, at least if the compiler outputs a corresponding warning.
- The compiler compiles the commands `DLOAD` and `LOAD` automatically into the corresponding `BLOAD` commands. Although no programs in the package contain a run-time module except the first program, you need not take the instructions in section 6.2 into account.
- Within the overlay package the start program may be loaded only with `RUN "name"` and the other programs only with `DLOAD` or `LOAD`.
- The compilation of a warm overlay package requires a bit of practice in operating the compiler. Overlay packages are usually only written by experienced programmers.

The compiler can output the following warnings for errors in the compilation:

`ILLEGAL OVERLAY :`

The overlay package was not compiled correctly. The absence of this message is not an absolute guarantee that the compiler was used correctly.

`LOAD ONLY IN OVERLAY :`

When using the commands `DLOAD` and `LOAD` the program must also be compiled as an overlay.

6.2 The run-time module

The run-time module contains all routines necessary for the execution of the compiled program. Every compiled program contains these routines. If disk space must be saved, you can disable the generation of the run-time module (menu option "G"). When loading several programs sequentially, only the first need have a run-time module. All additional programs without run-time modules must be loaded absolutely with `BLOAD "name", B0`. You will get an individual run-time module if you compile the following program:

```
10 REM
```

The run-time module and program can now be loaded individually. A program without run-time module can be run only if a run-time module is already in memory. Starting a program without a run-time module is done with the program "START" (see section 4.10).

The compiler can generate two types of run-time module. The second type is generated only if you directly disable the arithmetic module with the compiler setting. Disabling the arithmetic module automatically through the compiler has no effect on the module. The two types of run-time module may not be mixed.

Chapter 7: Speed

7.1 The optimization levels

The compiler has two levels of optimization which can be selected via option "O" of the advanced development package. The selected optimizer is displayed under option 1 of the main menu.

Optimizer I:

When selecting this optimization level, all possible optimizations and program changes are performed and guaranteed not to change the operation of the program. Optimization level 1 is therefore completely compatible with the BASIC interpreter. Calculations with integer values are executed as integer operations only when it is determined that an integer makes sense as the result. This is the case for most BASIC operations, however.

Optimizer II:

This optimization level has several important differences from level 1 and the BASIC interpreter:

- All variables, with the exception of string variables are classified as integer variables, that is, the compiler assumes that a % sign is behind each variable. The compiled program then works faster and the compiler does distinguish between variables like I and I% -- they just have the same type. Variable arrays, on the other hand, are classified with the correct data type.
- The division of two integer variables or values is always done with floating-point division in optimization level 1. Level 2, on the other hand performs such a division as an integer operation and ignores the remainder. In certain cases this leads to a difference from the interpreter. Usually this is not the case because no floating-point result is expected from

integer operations. An integer division runs significantly faster than a floating-point division, especially division by 2.

- The function `INT` is not treated as the greatest-integer function, but as a conversion to the data type "integer". Further processing with the corresponding value in a formula will be performed with integer operations. The difference here is that the function `INT` is no longer usable on values outside the integer range (-32768 to 32767).

Optimization level 2 has various applications. It is intended for the following type of programs:

- Programs for which high speed is more important than compatibility to the interpreter.
- Programs for which it is clear from the start that all variables will have only integer values.
- Programs in which the use of integer variables was not taken into account and the compiler is to do this. This is also done in level 1 by the run-time optimizations in a somewhat less effective form.

The following example is a typical application of optimization level 2:

```
10 A=INT (RND (1) *1000)
```

The variable `A` serves only to accept an integer number despite the fact that a floating-point calculation is used.

Programs for which optimization level 2 is used usually require some floating-point variables. Since optimization level 2 does not recognize these, it is then necessary to specify these variables to the compiler in a compiler directive.

Instruction:

REM@R=variable, variable, ...

The variables listed are classified as floating-point variables in optimization level 2.

Example:

```
10 FOR I=1 TO 1000
20 A=SQR(I) : PRINT A;
30 NEXT
```

If this program is compiled with optimizer II, the following line should be added:

```
5 REM@R=A
```

The optimization level can also be changed within the program:

```
REM@O1           ;enables level 1
REM@O2           ;enables level 2
```

Switching the optimization level affects only the data type of variables if these variables are not used before the switch.

Optimization level 2 has no effect on the data type of arrays. Arrays of integers should always be designated with a "%" sign because this saves a good deal of space when using the BASIC interpreter. This usually makes it possible to test such a program with the interpreter. After compiling, the memory space for arrays is significantly larger.

NOTE! Use optimization level 2 only for programs which you have developed yourself and for those for which you know the operation of the program and the use of data types. This also applies to the other additional possibilities of the compiler over and above the interpreter.

7.2 P-code (speed code) and machine language

Very large programs are often not compilable into machine language because a program translated into machine language is always longer than the original. In larger programs there are also time-critical program segments, of course, and it may be necessary to compile these into machine language. For this reason BASIC 128 has the ability to change the code generator during the compilation with two compiler directives:

1) REM@M

This directive tells the compiler to compile all subsequent program lines into machine language. When these program lines are encountered, the corresponding machine language program is started.

2) REM@P

After this directive the lines following it are compiled in P-code again. When these lines are encountered, the P-code interpreter is started and the commands following are interpreted.

These two compiler directives should be used only if you understand their effects completely. The following restrictions apply when using these directives:

- The microprocessor of the Commodore 128 can execute only machine language and not P-code.
- The P-code interpreter can interpret only P-code and not machine language.

If the code is changed without encountering a corresponding switch point (compiler directive), the behavior of the program is no longer predictable and in no event will it work correctly. As a consequence of this, no program jumps within a program which end in a program segment with different code will succeed. The commands GOTO, GOSUB, RETURN, IF, ELSE, LOOP, EXIT, NEXT, and RESUME perform program jumps.

In addition, the COLLISION instruction executes a jump to an unpredictable location. By limiting the structure record (section 4.8), you can limit the corresponding range.

The TRAP instruction executes a jump from the erroneous location to the error-handling routine. The code type at the location where the error occurred is unimportant--the TRAP command must just have the same code type as the error-handling routine. Furthermore, the RESUME instruction represents a jump whose destination code must be taken into account, which is very easy for "RESUME line number". In addition, RESUME NEXT and RESUME are usually only performed for known program locations.

Switching the code generator in structured programs can be done without danger at the start and end of a block. A program block is designated such that no jumps are made out of it and no jumps are made into it. It is started at the beginning and is exited at the end. Larger programs are usually block-structured so that switching to machine code for parts of them is possible. Programs which are neither block structured nor possess some other recognizable structure should not switch the code generator within the program. Such programs usually do not run correctly because of the lack of structure or are very small.

It is especially simple and useful to switch the code generator for subroutines. Such a subroutine could have the following general form:

```
1000  REM@M
. . .
1980  REM@P
1990  RETURN
```

Lines 1001 to 1979 are compiled into machine code in this example. The subroutine may be started only with GOSUB 1000 and then exited only by executing lines 1980 and 1990 (with GOTO 1980 if necessary). This is the case with structured programs. If another subroutine is called within the subroutine, the code generator must be switched back first. In our example this would look as follows:

```
1500 REM@P  
1510 GOSUB5000  
1520 REM@M
```

It doesn't matter what code type the subroutine called is, as long as the first line (with the compiler instruction) is compiled into P-code. Time-critical subroutines usually call no other subroutines.

7.3 The integer value range for the POKE command

Some other possibilities of the Commodore 128 can be used in BASIC only with the help of the commands PEEK and POKE. These commands are usually used to pass data to assembly language programs and should therefore be executed as quickly as possible. The use of integer variables in combination with the POKE command is particularly useful because memory addresses are represented by whole numbers. There are exactly 65536 different memory addresses in the Commodore 128 (0-65535) and an integer variable can have one of 65536 different values (-32768 to 32767). Unfortunately, the number ranges are not identical. Floating-point values and variables must be used for memory addresses above 32767. In connection with the POKE command and some other commands, the compiler has the ability to perform integer operations for values above 32767, though this involves sacrificing the negative numbers. The user of the compiler notices this through higher program speed.

```
5  REM@I=I
10  FOR I=1024 TO 2023
20  POKE I,65
30  POKE I+54272,0
40  NEXT
```

The program becomes faster with the following change:

```
6  OF%=30000
30  POKE I+OF%+24272,0
```

This change causes only integer values to be processed and a floating-point addition results in two integer additions. Such tricks should be used only in extremely time-critical program locations.



Chapter 8: Memory addresses and machine language

You should read this chapter if you want your BASIC and assembly-language programs to work together or if you use POKE commands in your program.

8.1 Memory layout

Pressing the <E> key from the advanced development package moves you to another menu, with whose help you can manipulate the memory usage of the compiler arbitrarily. Press the appropriate digit <1> through <5> and then enter the address.

The memory layout of compiled programs results from the values printed by the compiler and the preset and not-yet changed addresses in the memory menu. The values for the start of the run-time module and the arithmetic module are found in the program "START" on the BASIC 128 disk:

1) 0 to start of bank 0

System memory

2) Start of bank 0 to data code start

Storage for individual variables and arrays

3) Data code start to top of bank 0:

Program code

4) Run-time module or arithmetic module start to memory end

Run-time system

5) \$0400 to start of bank 1:

Stack for string descriptors in bank 1

6) Start of bank 1 to string end:

Free memory area for strings in bank 1

7) String end to top of bank 1:

Memory for arrays and individual variables in bank 1

8) Top of bank to end of memory:

Run-time system in bank 1

An area of memory can be reserved by moving the top values down and those for the start values up. The compiler will no longer use these areas and they can be used to store assembly language programs or new character sets, for instance. Be sure to note that neither the run-time system nor the arithmetic module is movable. A memory area between the program and the run-time system or arithmetic module can be created by lowering the setting of the code start below that specified by the compiler.

The compiled program stores the variables in the following form:

Integer variables:

2 bytes:

low byte, high byte

Floating-point variables:

5 bytes:

first byte exponent, second through fifth bytes mantissa
or

first byte 0, third and fourth bytes integer representation

The USR function can be used to pass a floating-point variable in the interpreter format to an assembly language program.

String variables:

3 bytes:

Length, address of the string low byte, address high byte

Strings:

2 bytes trailer plus 1 byte per character

An assembly language program can change the contents of variables. The addresses of the variables can be obtained from the symbol table. For strings, neither the location of the string nor its length can be changed by an assembly-language program. This can be done only by the compiled program or by uncompiled programs of the interpreter.

The graphics storage from \$1C00 to \$4000 is automatically reserved by the compiler if a command which uses this area appears in the program. In this case the value for the start of bank 0 may not be changed or only changed to a value over 16383.

8.2 Memory addresses

When using POKE and PEEK commands, it should be noted that a compiled program uses some memory locations differently than the interpreter. The following applies for the usability of POKE commands:

- There is no change to banks 2-15
- All memory locations from \$0000 to \$1300 in bank 0 used by the operating system or by commands of the interpreter retain their function (keyboard buffer, pressed key, I/O vectors, screen memory, graphic values, etc.). Many previously unused memory locations in this area are used by the run-time system.

Changes occur only in the following memory locations:

The reserved area of \$1300 to \$1C00 is used by the compiled program. If this area is to be used in another manner, it can be released with a compiler instruction:

REM@S address

The whole buffer can be freed with the following command:

REM@S7168

When using graphics the following command will free the graphics screen:

REM@S16384

Memory locations which the BASIC interpreter uses for interpreting the program partially lose their meaning. This is a natural result of compiling. Only the memory locations which can normally be used meaningfully with POKE commands remain. These include:

- floating-point registers
- random number
- CHRGET routine and other memory access routines
- status word ST
- RND value
- pointer to the program start (43-44)
- pointer to the program end (45-46)
- pointer to the string stack (53-54,57-60)
- pointer to the next DATA element (67-68)
- description of DS\$
- hires flag
- mode flag

The stack pointer \$7D, \$7E is used in a different manner by the compiler than by the interpreter. It is possible to read memory location \$7D and save it again if necessary in order to check the FOR and GOSUB nesting. Memory location \$82 must also be corrected. Memory location \$7E may not be affected.

Example:

```
10 A%=PEEK(125):REM save stack state
20 FOR I=1 TO 1000
30 POKE 125,A%:POKE 130,A%:
      REM restore stack state
40 NEXT:REM results in NEXT WITHOUT FOR
```

When using addresses (2 bytes), it must be noted that it is possible that other address than when using the interpreter will be stored there. This applies especially for the memory management (45-58) and for the error handling (768-769).

The differences between the interpreter and compiler in the layout of memory addresses is so small that it has practically no repercussions.

8.3 Special commands

SYS:

The **SYS** command of BASIC 7.0 has the same parameter list as the **RREG** command. For this reason this command has only limited use for BASIC extensions as opposed to the **SYS** command of BASIC 2.0. This also applies for the compiler.

BANK:

When starting the program, the command **BANK 15** is automatically executed because this involves the standard setting of the interpreter. If another bank is to be used, the corresponding switch must be made within a program.

STOP:

This command causes the end of a compiled program. As with the output of error messages, the command **CONT** cannot be used.

POINTER :

The **POINTER** function of the interpreter always specifies an address in bank 1. The compiler places some of the variables in bank 0 as well. For this reason the compiler has the ability to read the memory bank with **PEEK (2)**. A program which is to run correctly with the interpreter as well as the compiler could look as follows:

```
10 POKE 2,1:
    REM Bank for the interpreter
20 A=POINTER(X)
30 BANK PEEK(2):
    REM Bank for compiler and interpreter
40 E=PEEK(A):REM read value
```

The compiler usually outputs the warning "POINTER WITHOUT BANK" if the **POINTER** function is used incorrectly.

The compiler always puts a variable at a set address. It cannot occur that the **POINTER** return several different addresses during the course of a program, as can sometimes happen with the interpreter.

If a data field must be placed in bank 1 for reasons of compatibility, this can be communicated to the compiler directly after the **DIM** command:

REM@B

8.4 Symbol tables

BASIC 128 has the ability to load and save symbol tables. A symbol contains all of the variables in a program and the addresses at which they will be stored. The symbol tables themselves are stored in the form in which they are internally processed by the compiler.

Saving a symbol table:

To save a symbol table the compiler needs only a name for the table. This is done with menu option "C". The name "OVERLAY" is reserved for overlay symbol tables. The symbol table is saved after compilation.

Loading a symbol table:

The name of a symbol table to be loaded can be specified with menu option "B". The table is loaded before compilation. All of the variables and memory addresses listed in the symbol table are accepted. This is useful if several programs access common variables, in order, for example, to use the same assembly-language programs. The compiler makes use of this when compiling overlay packages.

Processing a symbol table:

A program called SYMBOL is located on the BASIC 128 distribution disk. After loading and starting the program it asks for the name of the symbol table. After entering the name you can select between two options.

By pressing the <1> key the program will be told to convert the symbol table into the format used by the Abacus ASSEMBLER/MONITOR. The name of the converted symbol table is the same as that created by the compiler. The old table is not erased however because the compiler places "S-" in front of the

name of the table. This need not be taken into account. The ASSEMBLER/MONITOR can load symbol tables with the the pseudo-op ".LST" (see the ASSEMBLER/MONITOR manual). The names of the variables can be used in an assembly language program as if they were defined in the program. This makes it possible for an assembled program to access variables in a compiled program. The program SYMBOL transforms the names of the variables in order to distinguish them from each other. In the following example, "na" stands for the first two bytes of the variable name:

Individual variables:

```
na      -na
na%     -naIN
na$     -naST
```

Arrays:

```
na      -ARna
na%     -ARnaIN
na$     -ARnaST
```

The representation is not dependent on the optimization level. It is derived directly from the variable name. The following fragment of an assembly-language program exchanges two BASIC integer variables:

```
100  LST  8, 2, "name, S, R"
110  ...
1000 LDA  AIN
1010 LDX  BIN
1020 STX  AIN
1030 STA  BIN
1040 LDA  AIN+1
1050 LDX  BIN+1
1060 STX  AIN+1
1070 STA  BIN+1
1080 ...
```

This corresponds approximately to the following BASIC program:

```
100 H%=A% : A%=B% : B%=H%
110 ...
```

The ASSEMBLER/MONITOR runs in the C-64 mode of the Commodore 128, but the assembled programs can be run in the C-128 mode and used in a compiled program.

If you do not have an assembler, you can still process the symbol table. After starting the program SYMBOL, press the <2> key to output the listing. The program asks for the number of the device on which the listing is to be outputted. The following device addresses have meaning:

0 = device 0 = screen
4 = device 4 = printer
8 = device 8 = sequential file on the disk

The program then outputs the names of the variables followed by the memory address and the memory bank.

The program SYMBOL may not be used on the symbol file OVERLAY. To transform an OVERLAY file, let the compiler load it and then save it to another file by compiling the following program in the non-overlay mode with the appropriate compiler settings.

10 REM

The program SYMBOL is just a suggestion for processing symbol tables and can be extended as desired, since it is not compiled. Possibilities include a symbol editor or adaptation to another assembler.



Chapter 9: The features of BASIC 128 and their applications in overview

This chapter offers only a short overview of the compiler and its use. Most of the compiler capabilities are described in greater detail in the appropriate chapters of this manual.

Starting the compiler:

The <1> key or <RETURN> starts the compiler when you press it in the main menu. You can always exit a menu with <RETURN>, provided the compiler is not waiting for you to end some input with <RETURN>.

Loading compiler parameters:

You can use the <2> key in the main menu in order to load all of the compiler settings from the main menu and then automatically move to menu option "3". Via this option you can load only files which are stored under the name "B128" (see option "I").

Setting compiler parameters:

When starting the compiler with the <1> or <RETURN> key, the compiler has a list of values and instructions which it needs for compilation. To display this list on the screen, press <3>. You can now change this list and then press <RETURN> to get back into the main menu. The individual options on the list are designated with letters and can be selected with the appropriate keys:

Menu Options:

Option "A" - Code generator

The compiler can create either P-code, machine language (6502/6510/8502), or no code at all. Machine language programs receive "-M" before the name, other programs "P-". The compiler preset is the P-code generator.

Option "B" - Load the symbol table:

Via this option the compiler can load a variable list which was saved when compiling another program. This has the result that both programs use the same addresses for the corresponding variables (overlay packages and similar applications).

Option "C" - Saving the symbol table:

Saved symbol tables can be loaded when compiling other programs or used by an assembler for the label addresses. This is useful when a BASIC program calls machine language subroutines with the SYS command.

Option "D" - Generates a line list:

After a compiling a program, the compiler can save a line list of the program. This list can be loaded with `DLOAD "Z-program_name"` and listed with `LIST`. Each BASIC line (right side) is assigned to a memory address (left side) (for error messages or for starting a program segment with `SYS`).

Option "E" - Memory division:

With this menu option you can change the memory division of the compiler. A sub-menu with the following menu options appears:

"1" - start of the variable storage in bank 0

"2" - end of the program and data storage in bank 0

"3" - start of the data storage in bank 1, lower boundary for strings

"4" - end of the data storage in bank 1

Memory space, such as for a machine language program, can be reserved by changing the values of these four memory settings.

"5" - maximum start value for the compiler-generated code

"6" - This option can be used to turn the fast floating-point compiler routines off. This makes the program about 3K shorter. The compiler switches the routines off automatically if they are not required.

Option "F" - Compiler mode:

If the 40-column screen is active, the compiler runs in the `SLOW` mode of the C-128 in order to make screen display possible. You can also run the compiler in the `FAST` mode, however; in this case the compiler switches the screen on only to output important messages.

Option "G" - Generating a run-time module:

You can disable the automatic inclusion of the run-time module (about 9K) and then load module and program from disk individually. This saves disk space and loading time for overlay packages.

Option "H" - Setting the BASIC extension:

If the compiler is to compile a program which contains the commands of a BASIC extension, this must be selected via this option and the memory division of the compiler must be set so that memory is free for the extension. See the manual for your extensions to see if and to what extent is also extends to the compiler.

Option "I" - Load and save all compiler settings

Via this menu option you load and save the compiler settings. The compiler always places "E-" before the name of a file. This need not be taken into account for the name, however.

Option "J" - Line and command recording:

To compile some commands in BASIC 7.0 (such as RESUME NEXT, TRON, COLLISION) the compiler must insert the line and/or command structure into the compiled program because these commands will otherwise be executed incorrectly. This menu option should be used only for small programs; larger programs should use the compiler instructions instead.

Option "K" - Error trap line:

This menu option allows the automatic insertion of a TRAP command in the program.

Option "L" - Overlay pass

This option sets the overylay pass. Normally this is done via option "4" of the main menu, however.

Option "M" - Disk command channel:

With the help of this option you can send commands to the disk drive. It suffices to enter the first letter of the desired DOS commands. The compiler then asks for the command parameters. In addition to the commands `HEADER`, `SCRATCH`, `RENAME`, `COLLECT`, and `DIRECTORY`, there is the command `OTHERS`, with which arbitrary strings can be sent to the disk drive via the command channel. If `HEADER` is used with an `ID`, both letters of the `ID` should be separated from the diskette name with a comma.

Option "N" - Directory:

This menu option reads the directory of the disk in the disk drive.

Option "O" - Optimization level:*Optimization level 1:*

This level is completely compatible to the BASIC interpreter V7.0. All program optimizations performed do not affect the course or behavior of the program in any way but serve only to achieve higher speed. In order to use the speed of integer calculations in level 1, integer variables must be designated appropriately. In Commodore BASIC this is done by appending a percent sign (%) to the variable name.

Optimization level 2:

Level 2 differs from level 1 in that all variables are viewed as integer variables, even those not viewed as such by the BASIC interpreter. If some variables are to contain floating-point values despite this, they must be designated with a corresponding compiler instruction. In addition, the level 2 optimizations involve integer division and integer conversion (`INT`) which are not compatible with the interpreter in all cases.

Compiling overlay programs:

Programs which are to be loaded and started with DLOAD or LOAD should usually also use the same variables or the same contents (warm overlay). When loading with RUN, all variables are erased and such programs can be compiled normally (cold overlay). Compilation of overlay packages, which use the command LOAD or DLOAD, is done in two passes:

Pass 1:

Compile all of the programs in the overlay package. Immediately after activating the compiler, press <4> (overlay) and then <1> (pass 1). All compiler settings must be made in pass 1. The compiler automatically disables the run-time module except when compiling the first program in the package, because no program without a run-time module can be started from the direct mode. The first program in the overlay package must therefore be a start program, which may be started only with RUN. No program is created in pass 1, just a table; for this reason this pass is done relatively soon.

Pass 2:

Compile all of the programs again, whereby you use <4> and <2> (pass 2) this time. The compiler then generates the desired program.

If the disk space does not suffice, you can also delete the original programs after compilation (option "M" in the sub-menu).

In order to start an overlay package, the names of the compiled programs must be changed so that the programs can load each other.

Compiler directives:

It is often necessary to change the compiler options during compilation. This is done with the help of special REM statements within the program. These directives are designated with an at-sign (@).

Switching to generation of machine code:

Format:

REM@M

Effect:

The compiler immediately begins to create machine code. In addition, an instruction is inserted into the program which informs the P-code interpreter at run-time that a machine language program follows.

Switching back to P-code:

Format:

REM@P

Effect:

The compiler generates P-code again. In addition a command is inserted into the program which activates the P-code interpreter.

Switching error handling on and off:*Format:***REM@E line number***Effect:*

This option corresponds to the TRAP command of the interpreter, but is not active until after the compilation.

Declaration of integer variables:*Format:***REM@I=variable, variable...***Effect:*

All variables listed (floating-point variables with % sign) are classified as integer variables by the compiler, which leads to faster program execution. This allows you in addition to use integer variables within FOR/NEXT loops, which the BASIC interpreter normally does not allow. This command makes sense only in connection with optimization level 1.

Declaring floating-point variables*Format:***REM@R=variable, variable...***Effect:*

All variables listed as classified as floating-point variables, which is necessary only when using optimization level 2.

Switching optimization level:*Format:***REM@O level number***Effect:*

The optimization level is changed. This affects only the data type of variables which have not yet been referenced by this point in the program.

Releasing reserved memory area:*Format:***REM@S address***Effect:*

The compiler automatically uses the unused memory area from \$1300 to \$1C00 for data storage. This can be disabled with this command.

*Example:***REM@S7168**

This directive causes the compiler to place variables at \$1C00 on up.

Selecting bank 1:*Format:***REM@B***Effect:*

The array dimensioned immediately before is placed in bank 1 independent of available memory. Normally the compiler places arrays in bank 0 until no more space is available there.

Switching to the FAST mode:*Format:***REM@F***Effect:*

The compiler is switched to the **FAST** mode. This directive is useful when a program is often compiled in the 40-column mode.

Selecting program structure recording:*Format:*

```

REM@O           ;turns the record off
REM@L           ;turns the line record on
REM@C           ;turns the command record on
REM@B or REM@LC ;turns both records on

```

Each of these directives may be preceded by a **T** if the **TRON** command is used. The run-time system needs a record of the original command or line structure in order to execute some **BASIC 7.0** commands. This makes the program longer and slower. For this reason the structure recording can be limited to individual program segments with these commands.

Using the optimization options of BASIC 128:

Although compilation speeds up your program significantly, the speed can be increased even more if you pay attention to which operations will be executed particularly quickly after compilation while you are writing the program. Usually it suffices to pay attention to the following rules to achieve the highest speed:

- Things you have learned to increase the speed of individual operations in an interpreted program probably don't apply to compiled programs any more.
- Operations with integer variables run faster than those with floating-point variables. According to experience, the larger share of all variables in most programs can be replaced by integer variables (values between -32768 and +32767). Integer variables can be indicated to the compiler and interpreter by appending a "%" sign to the variable name. In addition, you can inform the compiler of this with the help of REM instructions or with optimization level 2.

Integer values are used exclusively in the following functions and applications, for instance:

loop counter, array indices, POKE, PEEK, ON, WAIT, file parameters for OPEN, SYS, TAB, SPC, FRE, number parameters for string functions, ASC, CHR\$, logical operations, all graphics commands, often with comparisons and *,/,+,-, and with almost all other BASIC commands.

In each of these cases it does not make sense to use floating-point variables, and doing so will slow down the program. This is not detectable when using the BASIC interpreter because it performs all calculations with floating-point numbers.

- String operations are performed differently by compiled programs than by the interpreter. This results in higher execution speed for complex string formulas.

- The compiler takes work away from the program, such as finding jump destinations for GOTO, GOSUB, the interpretation of commands and formulas, syntax checking, decimal/floating-point/integer conversion, locating variables, transforming and optimizing formulas, calculating operations with constant values and strings, memory management, etc.
- The complex arithmetic functions of BASIC 7.0 are significantly accelerated by BASIC 128 (SIN, COS, TAN, ATN, EXP, LOG, SQR, ^). It is not necessary to avoid these functions in time-critical program segments.

Chapter 10: Differences between BASIC 128 Compiler and BASIC 64 Compiler

BASIC 128 is a revision of the BASIC 64 compiler for the Commodore 64. BASIC 64 already contained many efficient capabilities, but it was possible to build even more important features into BASIC 128. The following contains an overview of the most important new features together with a reference to the corresponding chapter of this manual:

- BASIC 128 is completely compatible with BASIC 7.0.
- The average speed of compiled programs can be considerably increased (chapter 2).
- BASIC 128 performs run-time optimizations in addition to the compiler optimizations (section 3.6).
- The floating-point functions TAN, SIN, SQR, etc. are executed considerably faster than they are by the interpreter (section 4.4).
- The code generator generates shorter P-code and faster machine language than BASIC 64.
- The entire 128 K RAM of the Commodore 128 is used by the compiler. In addition to bank 1, the entire memory of bank 0 not required for program code is available for variables and variable arrays (section 4.5).
- A record of the program structure is necessary to compile certain BASIC 7.0 commands. BASIC 128 offers several flexible options for doing this.
- The settings for memory management are more comprehensive than those for BASIC 64 (section 8.1).
- Compiler settings can be saved in files and reloaded again later (section 4.9).

- The compiler can be started without reloading (section 4.10).
- The procedure for compiling overlay packages has been improved and no longer needs to be used for cold overlays (section 6.1).
- BASIC extensions don't have great significance for the Commodore 128 because BASIC 7.0 is quite capable. For this reason, extensions are supported by BASIC 128 only when they are adapted for the compiler (section 5.3).
- Accessing the DOS on the disk is very easy and there is an option for reading the directory (section 9.1).
- The compiler outputs warnings in addition to the error messages of the interpreter (section 4.2).
- A STEP value is allowed for integer loops (section 4.7).

Chapter 11: Additional applications

11.1 Input/Output

Peripheral devices like the disk drive and printer work just as slowly after compilation as before. Therefore when saving and loading data it is advisable to make full use of the available speed of the disk. The operating system sends a control code before and after each command that addresses the disk. The transfer of this command requires time. In order to avoid this, the largest possible set of data should be transferred with each command:

```

90  REM @I=I
100 FOR I=1 TO 100
110 PRINT#1, CHR$( I) ;
120 NEXT

```

This program fragment could be worded like this:

```

90  REM@I=I
100 FOR I=1 TO 100
110 PRINT#1, CHR$( I) ; CHR$( I+1) ;
120 I=I+1 : NEXT

```

The command GET# works particularly slowly because it reads in only single bytes. For this reason, a line-end indicator should be chosen after each 80 character, provided the type of data allows this.

```

90  REM@I=I
100 A$="":FOR I=1 TO 80
110 GET#1, B$:A$=B$+B$
120 NEXT

```

This could be simplified to the following lines given the appropriate data format:

```

100 INPUT#1, A$

```

This saves the repeated transmission of control codes as well as the concatenation of a string.

Data output on the screen is usually fast enough. When constructing screen masks or similar devices, the screen may flicker during the reconstruction of the mask. Here the same effect as when writing programs occurs. If the cursor is moved beyond the right edge of the screen, all additional lines are move down. As soon as a program clears the screen in order to display new or changed data, this effect can occur. The construction of the screen representation then hesitates. This can be prevented by simply writing over the old screen, causing no new lines to be inserted.

11.2 High-resolution graphics on the 80-column screen

Drawing 640x200 point graphics on the 80-column screen takes quite a long time with the BASIC interpreter. High-resolution graphics show off the advantages of the compiler against the interpreter. The following routines use almost exclusively integer operations:

```

5   REM@O2
10  BANK 15:WR=52684:RE=52698:FOR I=0TO7:
    MA%(I)=2^(7-I):NEXT
20  SYS WR,128,25:SYSWR,7*16+1,26
30  SYS WR,0,18:SYS WR,0,19
35  FOR I=0TO64:SYSWR,0,31:SYS WR,0,30:NEXT
40  GOTO 10000
100 REM DRAW X,Y,X2,Y2
105 X1=X:Y1=Y:DX=ABS(X2-X1):DY=ABS(Y2-Y1)
110 IF DX=0 THEN SX=0:ELSE SX=INT((X2-X1)/DX)
120 IF DY=0 THEN SY=0:ELSE SY=INT((Y2-Y1)/DY)
130 IF DY>DX THEN BEGIN
140 XS=0:DH=DY:DL=DX
150 BEND:ELSE BEGIN
160 XS=-1:DH=DX:DL=DY
170 BEND
180 X=X1:Y=Y1:C=INT(DH/2):CP=DX+DY:GOSUB 1000
190 DO:C=C-DL:IF C<0 THEN BEGIN

```

```
200 C=C+DH:Y=Y+SY:X=X+SX:CP=CP-2
210 BEND:ELSE BEGIN
220 IF XS THEN X=X+SX:ELSE Y=Y+SY
230 CP=CP-1:BEND
240 GOSUB 1000
250 LOOP UNTIL CP<=0
260 RETURN
1000 REM PLOT X,Y
1001 REM@M
1005 AD=INT(X)/8+INT(Y)*80:H=AD/256:L=AD AND
    255
1010 SYS WR,H,18:SYS WR,L,19:SYS RE,0,31:
    RREG BY
1020 SYSWR,H,18:SYS WR,L,19:SYS WR,BY ORMA%(X
    AND 7),31
1030 REM@P
1040 RETURN
2000 REM CIRCLE X,Y,R
2001 REM@01
2010 S=R*R:X1=X:Y1=Y
2030 FORX2=0TOCOS(.786)*R+1
2040 Y2=SQR(ABS(X2*X2-S))
2050 X=X2+X1:Y=Y2+Y1:GOSUB 1000
2060 Y=Y1-Y2:GOSUB 1000
2070 X=Y2+X1:Y=X2+Y1:GOSUB 1000
2080 X=X1-Y2: GOSUB 1000
2085 X2=-X2: IF X2 < 0 THEN 2050
2090 NEXT:RETURN
10000 REM DEMO
10005 FORA=50TO600STEP30
10010 X=A:Y=10:X2=600-A:Y2=190:GOSUB100
10020 NEXT
10030 FORW=50TO80STEP10
10040 X=200:Y=90:R=W:GOSUB2000
10050 NEXT
10060 GOTO 10060

READY.
```


This program clears the memory of the 80-column video chip and thereby also the 80-column character set. The 80-column character set can be restored with `SYS 52748`.

The program has the following graphic routines:

`GOSUB 100` ;draws a line from `X,Y` to `X2,Y2`

`GOSUB 1000` ;draws a point with coordinates `X,Y`

`GOSUB 2000` ;draws a circle with center `X,Y` and radius `R`

You can also replace the example program at line 10000 with your own graphics programs, of course. The program makes full use of the capabilities of BASIC 128. After removing the compiler instructions in lines 1001 and 1030, it can even be compiled completely into machine code. This results in fast routines that make use of the graphics capabilities of the Commodore 128 which can be added to any other program.

The higher speed of compiled programs offers you possibilities with the programming language BASIC that you would have never though possible before.

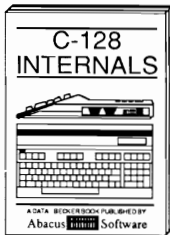
INDEX

advanced devolpment package	3, 66
arrays	30, 62
BANK	59, 74
BEGIN...END	39
COLLISION	34, 51
Command Record	33, 68
constants	11
data code	21
disk errors	26
direct mode commmands	32
directives	4, 50, 71
DIRECTORY	69
EL variable	33
errors	22, 24, 39
error handling	39, 72
extensions	21, 41, 68
FAST	37, 67, 74
floating-point calculations	28, 48, 56, 72
formulas	12
functions	41
Hi-res graphics	80
IF...THEN...ELSE	39
input	79
integers	9, 14, 32, 53, 56, 72
Line List	27, 66
Line Record	33, 68
LOAD	1, 36, 43
loops	6, 9, 16, 32
machine code	17, 50, 66, 71
memory layout	55, 57, 67, 73

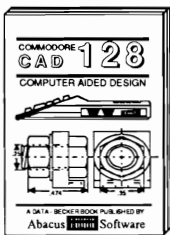
object code	21
optimizer	9, 14, 19, 48, 73, 75
Optimizer I	47, 69
Optimizer II	47, 69
options	66
output	79
overlays	43, 68, 70
pass 1	21, 43
pass 2	21, 44
P-code	3, 17, 50, 66, 71
POINTER	60
REM@	4
RESET	36
RESUME	33,34, 51
RUN	1, 3, 43
run-time	19, 23, 24, 43, 46, 67
speed	47
strings	13, 21, 57
STOP	59
symbol table	61, 66
SYS	59
system errors	24
TRAP	35
TRON	33, 34, 35
variables	15
warnings	22, 23

HOT OFF THE PRESS

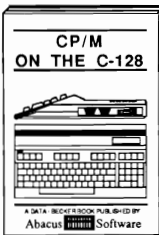
NEW INFORMATION FOR YOUR C-128



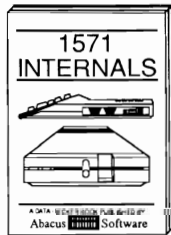
A detailed guide presenting the 128's operating system, explanations of graphics chips, a concise description of the MMU, well documented ROM listings, more. \$19.95



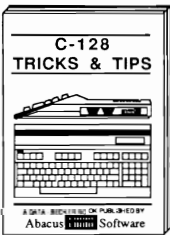
Computer Aided Design on your C-128 or 64. Design a CAD system using programs provided. Create 3D objects. With 128-Hardcopy and 128-Merge program listings. \$19.95



An essential guide to using CP/M on your 128. Simple explanations of operating system and its memory usage, CP/M utility programs, submit files, and other subjects. \$19.95



A guide for novice and advanced users. Sequential and relative files, direct access commands, directory usage, important DOS routines, compressed DOS listings. \$19.95



This book is chock full of information which no 128 user should be without. It covers memory usage, hires graphics in 80 columns, windowing, memory locations. \$19.95

...AND TRUSTED INFORMATION ON THE 64!



ANATOMY OF C-64 Insider's guide to the 64 internals. Graphics, sound, I/O, kernel, memory maps, more. Complete commented ROM listings. 300pp \$19.95

ANATOMY OF 1541 DRIVE Best handbook on floppy explains all. Many examples and utilities. Fully commented 1541 ROM listings. 320pp \$19.95

MACHINE LANGUAGE C-64 Learn 6510 code write fast programs. Many samples and listings for complete assembler, monitor, & simulator. 200pp \$14.95

GRAPHICS BOOK C-64 - best reference covers basic and advanced graphics. Sprites, animation, hires, Multicolor, lightpen, 3D graphics, IRO, CAD, projections, curves, more. 350pp \$19.95

TRICKS & TIPS FOR C-64 Collection of easy-to-use techniques: advanced graphics, improved data input, enhanced BASIC, CP/M, more. 275pp \$19.95

1541 REPAIR & MAINTENANCE Handbook describes the disk drive hardware. Includes schematics and techniques to keep 1541 running. 200pp \$19.95

ADVANCED MACHINE LANGUAGE Not covered elsewhere - video controller, interrupts, timers, clocks, I/O, real time, extended BASIC, more. 210pp \$14.95

PRINTER BOOK C-64/VIC-20 Understand Commodore, Epson-compatible printers and 1520 plotter. Packed: utilities; graphics dump; 3D-plot; and commented MPS801 ROM listings, more. 330pp \$19.95

SCIENCE/ENGINEERING ON C-64 In depth intro to computers in science. Topics: chemistry, physics, biology, astronomy, electronics, others. 350pp \$19.95

CASSETTE BOOK C-64/VIC-20 Comprehensive guide; many sample programs. High speed operating system fast file loading and saving. 225pp \$14.95

IDEAS FOR USE ON C-64 Themes: auto expenses, calculator, recipe file, stock lists, diet planner, window advertising, others. Includes listings. 200pp \$12.95

COMPILER BOOK C-64/C-128 All you need to know about compilers: how they work; designing and writing your own; generating machine code. With working example compiler. 300pp \$19.95

Adventure Gamewriter's Handbook Step-by-step guide to designing and writing your own adventure games. With automated adventure game generator. 200pp \$14.95

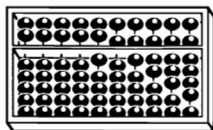
PEEKs & POKEs FOR THE C-64 Includes in-depth explanations of PEEK, POKE, USR, and other BASIC commands. Learn the "inside" tricks to get the most out of your 64. 200pp \$14.95

Optional Diskettes for books For your convenience, the programs contained in each of our books are available on diskette to save you time entering them from your keyboard. Specify name of book when ordering. \$14.95 each

Call now for the name of your nearest dealer. To order by credit card call 616/241-5510. Other software and books are available - ask for free catalog. Add \$4.00 for shipping per order. Foreign orders add \$8.00 per book. Dealer inquires welcome - 1200+ dealers nationwide.

Abacus Software
 P.O. Box 7211 Grand Rapids, MI 49510 - Telex 709-101 - Phone 616/241-5510

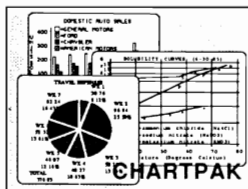




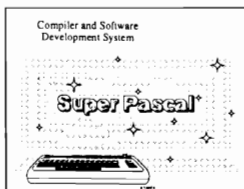
NEW



'128 SOFTWARE



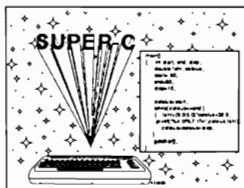
CHARTPAK
Make professional quality pie, bar and line charts, and graphics from your data. Includes statistical functions. 3x the resolution of '64 version. 500+ data points Outputs to most printers. \$39.95



SUPER Pascal
Complete J&W development system. With enhanced editor, compiler, built-in assembler, tool-kit, graphics, 220 page handbook, and plenty more. \$59.95



BASIC 128 Compiler
Versatile compiler instantly turns BASIC into lightning fast 6510 machine code and/or compact speedcode. Variable passing overlays, integer arithmetic, and more. \$59.95



SUPER C
Complete K&R compiler and development system. Editor, compiler, linker, I/O library and extensive 200 page handbook. Creates fast 6510 machine code. \$79.95

...AND OUR OTHER FANTASTIC 64 SOFTWARE

Technical Analysis System
A sophisticated charting and technical analysis system for business investors. By charting and analyzing the past history of a stock, TAS can help pinpoint trends and patterns and predict a stock's future. TAS lets you enter trading data from the keyboard or directly from online financial services. \$69.95

Cadpak
A deluxe graphics design and drawing package. Use with or without a lightpen to create highly detailed designs with dimensioning, scaling, text, rotation, object libraries, hardcopy and much more. \$39.95

Xper
Capture your information on XPER's knowledge base and let this first expert system for Commodore computers help you make important decisions. Large capacity. Complete with full editing and reporting. \$69.95

PowerPlan
One of the most powerful spreadsheets for your Commodore computer. It includes menu or keyboard selections, online help screens, field protection, windowing, trig functions and more. PowerGraph is also included to create integrated graphs and charts for your spreadsheet data. \$39.95

Personal Portfolio Manager
Complete portfolio management system for the individual or professional investor. Allows investors to easily manage their portfolios, obtain up-to-the minute quotes, news, and perform selected analysis. \$39.95

Versions of the above are also available:
Super C \$4 \$79.95
Super Pascal \$4 \$59.95
BASIC \$4 \$39.95
Chartpak \$4 \$39.95

Abacus Software

P.O. Box 7211 Grand Rapids, MI 49510 Telex 709-101 Phone 616/241-5510
Call now for the name of your nearest dealer. To order by credit card, MC, AMEX or VISA, call 616/241-5510. Other software and books are available - Call and ask for your free catalog. Add \$4.00 for shipping per order. Foreign orders add \$12.00 per item. Dealer inquiries welcome - 1200+ dealers nationwide.







842

Product ID

REGISTRATION CARD

165793

Registration # _____ Product: _____

Name _____

Address _____

City _____ State _____ Zip _____

Purchase Information:

Dealer _____

Address _____

City _____ State _____ Zip _____


Returning this registration card entitles you to phone support for the above product. You may also obtain a backup copy of the diskette for a handling charge of \$10.00. This card and a check, money order or credit card number must accompany this request. Purchase orders are not acceptable.

BACKUP COPY? No, do not send a backup, but register my purchase
 Yes, send a backup copy, payment is enclosed

Credit card# _____

Expiration Date ____ / ____ / ____



You Can Count On 

**Abacus
Software**

**P.O. Box 7219
Grand Rapids, MI 49510**

BASIC-128

The *complete* BASIC compiler and development system

BASIC-128 is no ordinary BASIC compiler. It compiles your standard Commodore BASIC programs into either superfast machine code or very compact p-code. In fact, you can mix the two in during a single compilation.

BASIC-128 lets you compile a series of programs using the overlay features. It even lets you make use of any language extensions.

If you need to make your BASIC programs run *lightning fast*, or want to protect your precious programming techniques, get **BASIC-128**.

For Commodore-128 and 1541 or 1571 disk drive.