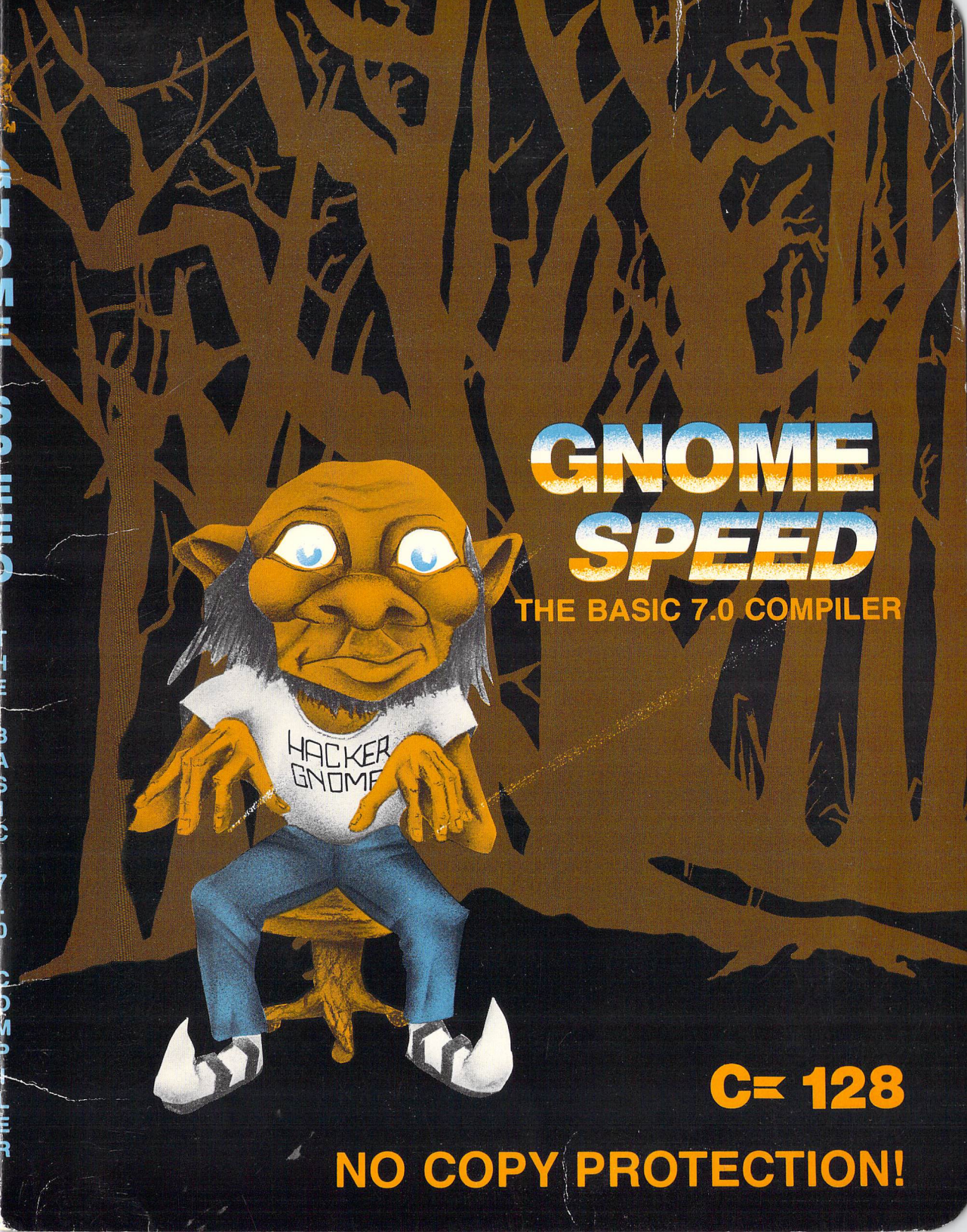
GNOME SPEED
THE BASIC 7.0 COMPILER
HACKER
GNOME
C= 128
NO COPY PROTECTION!

HACKER GNOME answers some commonly asked questions

GNOME SPEED COMPILES EVERY BASIC 7.0 COMMAND EXCEPT:

AUTO	DSAVE	HELP	SAVE
BOOT	DVERIFY	LIST	TRON
BSAVE	FN	MONITOR	TROFF
CONT	GO 64	NEW	VERIFY
DEF	HEADER	RUN	

Most of these would not be found in a final program. And for those like **HEADER,** that may be used, the BASIC 2.0 form of the command will compile. Many other BASIC commands and functions have been enhanced.

WHY IS A RUN-TIME MODULE NECESSARY? This Run-time program, which is easily copied onto your program disk, actually controls the execution of the P-code. Without this module, straight machine-code, which normally produces a very long program, would have to be generated.

WILL YOUR COMPILED PROGRAM BE SMALLER? Unless you write very compact, undocumented BASIC programs, your compiled version will be between 20 and 50 percent smaller. The actual reduction depends on the original program.

WHY WILL YOUR PROGRAM RUN FASTER? With a normal BASIC program, the BASIC interpreter reads every program line, each and every time it is executed. With a compiled program, this step is virtually eliminated. Also, searching for lines to GOTO or GOSUB, searching for variables and handling mathematical expressions is very slow in BASIC. **GNOME SPEED'S** P-code allows the run-time module to handle these tasks much more efficiently.

SM COMPILER 128

USER'S GUIDE

A BASIC COMPILER FOR THE COMMODORE 128

TABLE OF CONTENTS:

INTRODUCTION

This guide is intended to help you learn how to use the SM COMPILER 128. Even if you are an experienced programmer and are familar with other compilers, you should review this guide thoroughly. And always use it as a reference when writing a BASIC program that you intend to compile.

SYSTEM OVERVIEW

Your Commodore 128 computer is equipped to use the Programming language BASIC 7.Ø. An interpreter for this BASIC version is in the ROM (read only memory) of your C= 128. When you run a BASIC program, this BASIC Interpreter has to read each line of your program, and translate it into the internal machine format that your C= 128 computer can understand, before the C= 128 can execute it. The interpreter has to do this each time it executes a line of your program. This is often a slow and clumsy way of running a program!

SM COMPILER 128 eliminates this interpreting step during program execution, by translating your BASIC program into machine code and storing it on your disk as a compiled program. By converting your BASIC program into a machine code, the compiler does all of the work that it possibly can before the program is actually executed. This includes searching for variables and line numbers, checking for errors, etc. And mathematical expressions are converted into formats that will increase the speed of calculation. Unlike interpretive BASIC, when a compiled program needs a variable, it knows its exact location and does not have to perform the slow and tedious task of searching for it. Similarly, branch locations (i.e., GOTO xxx) are also known, rendering this slow task unneccessary during the execution of a compiled program.

Another advantage of compiling your program is the elimination of most if not all of your program coding errors. Syntactically erroneous code cannot be compiled and therefore the compiler will tell you where your errors are. You must correct them and compile again. In interpretive BASIC, your syntax is not checked until the program statment is executed. And in a program with many

branches, some of which are rarely used, you may not find these bugs during your testing.

SM COMPILER 128 will compile most BASIC programs written for the Commodore 128 computer. It is almost totally compatible with the BASIC interpreter, so that nearly all of the BASIC 7.Ø and BASIC 2.Ø commands/functions will compile. This includes both the GRAPHIC and SOUND commands, so that game programs should compile easily. It will compile programs upto 1999 lines in length. It will compile programs using upto 5ØØ distinct variables names (an array is counted as one variable name). It is easy to use and compiles your program relatively quickly. In short, it produces a reliable, executable, faster compiled program.

It is a three pass compiler, meaning that goes thru your program three times before it produces the final compiled version:

PASS 1: During this pass, the compiler translates each BASIC instruction into a P-code. Since the compiler does not know the length of the program and the addresses of the line numbers at this time, substitute parameters are located in the corresponding positions of the P-code. An address file is also created, which lists the locations of the substitute parameters of the P-code, that are to be replaced by their true value during Pass 2. It also sets up a list of the program line addressed and a Label list (of variable names). In addition, the length of any DATA text is calculated.

PASS 2: During this pass, the compiler uses the lists it created in the first pass. It checks to makes sure that all your GOTO's and GOSUB's, etc. reference valid line numbers. It also checks to make sure all your variables are setup and dimensioned properly. Invalid line numbers references and undimensioned arrays will be caught during this pass.

The machine code is then created on the basis of the P-code file, the address file, the list of line addresses and the Label list.

PASS 3: In the third pass, any DATA text will be chained to the converted program.

At the end of these three passes, assuming no errors were found, you will have a successfully compiled program called C/program name.

Now, there are some things that SM COMPILER 128 can't do. Certain BASIC commands and functions cause it problems. But most of these aren't normally used in a program. And sometimes, even though the BASIC 7.0 command doesn't compile, the BASIC 2.0 version of the command will compile. These incompatabilities and restrictions are discussed in the next chapter. However, with a little forethought and creative design, you will be able to compile almost any program.

HARDWARE REQUIREMENTS

The SM COMPILER 128 requires:

- Commodore 128 computer
- Any 40 or 80 column monitor
- Commodore 1541 or 1571 disk drive
- Any suitably interfaced printer (optional)

REGISTRATION, SUPPORT AND UPDATES

Writing a compiler is tricky business. And although we have done our best to offer you the finest compiler we can, we are sure that there are things that you would like to see incorporated into or changed in the compiler program. We'd also like to know how SM COMPILER 128 is being used, so that we can improve and enhance the program. Therefore, we strongly urge you to fill in and return the enclosed registration card.

Completing the registration card will benefit both you and ourselves. We will learn who is using our compiler and why. In turn, you will get unlimited support from us. Just call or write us if you have any questions, suggestions, complaints or problems:

SM SOFTWARE, INC. (215) 682-4920
P.O. BOX 27
MERTZTOWN, PA 19539-0027

We will try to help you out in any way that we can. As a registered user, we will keep you informed of any updates and enhancements. When new versions are released, and they will be, you will be offered the new version for just $10.00 (to cover postage and handling).

We really believe that two-way communication is important with this type of product. So let us know what is going on. And don't hesitate to call us with any problems or concerns. We will be glad to help out.

BACKUP OF THE COMPILER

We struggled with the option of copy-protecting or not copy-protecting our compiler and probably much to the dissatisfaction of some of you, we decided to copy-protect. But to keep peace with at least most of you, we are offering a backup copy to registered users for a nominal fee.

We know that this copy-protection isn't going to keep all of you from making backups for your own or others' use, but we felt it could at least prevent some of the unauthorized copying. And maybe the rest of you will have a little compassion and understand that it took alot of work and sweat to produce this product. Well enough of the pep talk!

If you want a backup copy of SM COMPILER 128, just send us a check for $5.00 along with your registration card (or if you have already registered, then send us your registration number, as stamped on the program diskette) and we will send you a new program disk pronto.

EXAMPLE PROGRAM

There are three example programs included on the program diskette. After reviewing this manual, it is suggested that you review these programs. They should help you understand the compiler more thoroughly.

EXAMPLE: This is the BASIC source code. The program evaluates mathematical expressions. Numerical expressions, the operators +, -, *, /, and parenthesis are allowed as input.

C/EXAMPLE: This is the compiled version of the program above.

RUNEXAMPLE: This program first BLOADS the RUNTIME module, that is needed to execute a compiled program, and then DLOADS the compiled program C/EXAMPLE. This program is an example of the loader program, that is discussed in USING YOUR COMPILED PROGRAM section of this guide.

USING THE COMPILER

Since the compiler really does all the work, there is not much for you to do as far as actually using the compiler.

LOADING AND RUNNING

There are basically two ways to start the compiler. The first way is to first insert the diskette into your drive (be sure it is on). Then, if your computer is off turn it on, or if it is already on, then press the reset button (the square button on the right side of the computer, next to the on/off switch). The compiler will load and start automatically.

The second way is to power up, insert the diskette into the drive and press the <shift> and <run/stop> keys simultaneously. The compiler will load and start automatically.

SETTING UP YOUR BASIC PROGRAM

After the copyright notice has been displayed, you will be asked for the **FILE NAME?**. Insert the diskette containing your BASIC source program and respond to to the prompt with programname <return>. The compiler will then start working. Please be sure to insert your program diskette first. If you forget, then start the loading and running process from the beginning.

PLEASE NOTE: The compiler needs diskette space of about 2 1/2 times your program length. So if your program is 60 blocks long, then you need about 150 free blocks on your diskette, before you can compile. (This space is only needed during the compilation process. Your resulting program will normally be smaller then the uncompiled version.)

PRINT OPTIONS

Next you will be asked **PRINTER (1) ON (0) OFF?**. If you select 0, then error messages will only be displayed on your monitor. If you select 1, then error messages will also be listed on your printer.

FAST/SLOW MODE

Your Commodore 128 computer operates in two modes, one with a 1 mHz clock and the other with a 2 mHz clock (twice as fast). It is possible, during the compiling of your program, to swith the C= 128 into the fast mode by pressing the cursor-right key. Please note that the 40 column screen will be turned off while in the fast mode. To switch to the slow mode, press the cursor-left key. The mode change will go into effect when the compiler starts working on the next BASIC line. Either set of cursor keys will work. Since it makes sense to try to compile in the fast mode, the following rules will help you out:

- If you have an 80 column monitor always switch into the fast mode by pressing the cursor right key at the beginning of the compiling process.

- If you have a 40 column monitor and a printer then always opt to have the printer on (1) and switch into the fast mode at the beginning of the compiling process. This way you will see your error messages on the printer. However, periodically, switch back into the slow mode, to see what the screen has to say, and how far along you are. When the compiler is done, also switch back into the slow mode.

- If you have a 40 column monitor and do not have a printer you can still use the fast mode. However, since the 40 column screen is lost during the fast mode, you need to switch into the slow mode periodically to see your error messages. Then switch back into the fast mode.

Sounds complicated, huh? Just try out the mode switching, and you'll get the hang of it.

COMPATIBILITY WITH THE BASIC INTERPRETER

SM COMPILER 128 will compile most BASIC programs. However, there are some design considerations and restrictions which need to be discussed. If you thoroughly familiarize yourself with this chapter, then you should not have any problems during the compiling process.

PROGRAM SIZE RESTRICTIONS

Programs upto 1999 BASIC lines in length and upto 500 different variable names (an array is counted as one variable name) will generally compile. However, programs which approach the maximum lines restriction, that have very long lines, may exceed the capacity of the compiler. This size restriction generally will not cause you any problems. (If you do run into a problem, and your program is extremely long, then please let us know.)

PROGRAM DESIGN CONSIDERATIONS

Even though the compiler has to check your program for coding errors, before it can compile it, this does not mean that you should write sloppy, untested code. Unlike the interpreter, where detected errors can be corrected immediately and the program re-executed, correcting a compiled program is more complicated. You have to correct the source code (your BASIC program) and then recompile the entire program. Therefore, it pays to write a well-structured, tested program.

Lines that contain only REMark statements are bypassed by the compiler, and are not compiled. Therefore, you cannot GOTO or GOSUB any line that contains only a REM statement. If the program does branch to a REM line, then you will get an UNDEF'D STATEMENT error, during Pass 2 of the compiling process.

All arrays, even those containing 10 or less elements, MUST be explicitly DIMensioned. If not DIM'd, then you will get an UNDEF'D ARRAY error in Pass 2.

And tricks of the trade, such as starting a remark statement with an apostophe or making remarks, without

REM, after a GOSUB, GOTO or RETURN will cause you problems. These types of tricks cannot be compiled by the compiler, so you will get an SYNTAX error message.

BASIC COMMANDS/FUNCTIONS THAT WILL NOT COMPILE

The following, lists BASIC commands or functions that SM COMPILER 128 will not compile. If found during the compiling process, you generally will get a SYNTAX ERROR. However, some of the commands/functions such as FN will cause the compiler to bail out. Most of these are not normally included in a program anyway. And for some of the disk handling commands, even though the BASIC 7.Ø form will not work, the BASIC 2.Ø command may. If any of these restrictions really cause you a heartache, let us know. We will try to help you out.

AUTO	
BOOT	
BSAVE	
CONT	
DEF	will be included in the next version
DSAVE	
DVERIFY	
FN	will be included in the next version
GO 64	
HEADER	use OPEN1,8,15:PRINT#15,"NØ:disk,id"
HELP	
LIST	
MONITOR	
NEW	
RUN	
SAVE	
TRON	
TROFF	
VERIFY	

BASIC COMMAND/FUNCTION RESTRICTIONS AND ENHANCEMENTS

The following, lists BASIC commands or functions that may behave differently after they are compiled. Although, they can be compiled, you should become thoroughly familar with the way they will execute in their new, compiled form.

BOX Input of polar coordinates and relative coordinates is also allowed.

CIRCLE See BOX.

CLR Arrays are erased, but can be redimensioned; single precision variables (i.e. A, B$, Z%) are reduced; strings have a null value; numeric variables have a value of zero (Ø).

DIM Arrays must be explicitly dimensioned. After a CLR, if variables are re-dimensioned, they must have the same dimensions.

DLOAD Variables are erased and cannot be passed between programs.

DRAW See BOX.

ELSE See IF.

END Variables are erased and there is no access to them from the direct mode of the interpreter.

GET Only one variable is possible. Not GET#1,A$,B$; only GET#1,A$:GET#1,B$.

GSHAPE See BOX.

IF THEN ELSE Will work correctly even outside of a BEGIN-BEND. When the IF-THEN-ELSE sequence is compiled an ELSE statement is always matched with the preceeding IF. (In the BASIC Interpreter, if an IF expression is not true, the interpreter searches for the next ELSE clause in the current line and executes the statement accordingly, without regard to possible IF statements on the THEN branch.)

EXAMPLE

```
1ØØ IF A=B THEN IF A=C THEN PRINT "B=C":
                       ELSE PRINT "A<>C"
```

The interpreter returns "A<>C", if A is not equal to B or if A is not equal to C. A compiled program returns "A<>C" only if A is equal to B and at the same time A is not equal to C.

Clear as mud, huh? We suggest you play around with the example above, in both the interpretive and compiled modes, until you get the hang of the difference.

INPUT #	INPUT#X,TI$ is not allowed.
INPUT	INPUT TI$ is not allowed.
INSTR	The optional starting position parameter is required.
LOCATE	See BOX.
LOAD	See DLOAD.
OPEN	In the case of OPEN using a cassette, the secondary address must be given.
MOVSPR	MOVSPR x,distance,angle is also possible.
PAINT	See BOX.
SOUND	Cannot specify waveform or pulse width. Will be in next version.
SSHAPE	See BOX.
STOP	A compiled program does not return a BREAK IN xxx (as is the case with the interpreter). It causes a BREAK ERROR IN xxx and execution of the program cannot be resumed with a CONT.
SYS	Input of parameters is possible. For example, SYS adr,AC,XR,YR,SR.
USR	Due to a somewhat different numeric representation than with the interpreter, this function should be used with caution.

PROGRAM OVERLAYS

In a program overlay, using LOAD or DLOAD, the current variables will be erased and not passed on to the follow-up program. Since the follow-up program has no information as to the address of the variables in the preceeding program, it cannot access them. If the follow-up program is NOT compiled, then it is neccessary to have a CLR command at the beginning of the program. This is neccessary because when the variables of a compiled program are erased, the storage space for single precsion variables is not cleared. This may cause the interpreter to produce nonsense variable values.

The passing of variable values from one program to the next, must be done with the use of PEEK and POKE. If you are not certain how to do this, then give us a call.

ERROR MESSAGES

As previously discussed, the compiler cannot compile statements containing program coding errors. This is the most common type of error message you will run into. However there a a couple of bugs in Commodore's DOS and OS that need to be discussed. And although we have tried to make the compiler as complete as possible, we will also discuss what to do if it bails out.

If these discussions on errors do not solve your problem, please do not hesitate to contact us (215) 682-4920. We'll try to help you. All we ask is that you have sent in your registration card, have tried to solve the problem yourself, have the exact wording of the error messages that were displayed and that you have a listing of your BASIC program source code.

ERRORS IN YOUR BASIC PROGRAM

Although the compiler will inform you of any coding errors in your BASIC program , this does not mean that you should write an untested, sloppy program. Unlike interpretive BASIC, where an error can be corrected immediately and the program run again, correcting a compiled program is more time-consuming. First, the compiler will display the error(s) on the screen, then continue compiling the rest of the program, displaying more error messages, if any. Next you have to load your uncompiled program and correct the errors. Then you have to compile the corrected program again. Please note that the second time thru the compiling process, the compiler may find other coding errors. This is caused by the fact that the original errors may have prevented the compiler from checking the syntax of some related instruction.

Also, the compiler cannot check logic errors. You may have intended the program to do one thing, but erroneously coded it to do another. As long as the code is syntactically correct, the compiler cannot and will not pick up the erroneous logic. Please note that this is true whether or not your program is compiled. An error in your logic will not be picked up by the BASIC interpreter either.

So it pays you to write clean, organized code and test your program before you compile it. However, none of us are perfect so you're bound to get a couple of error messages displayed during compilation. These messages are the same as those that are displayed by the BASIC interpreter and are listed in APPENDIX A of your Commodore 128 System Guide.

The error will be displayed on your monitor screen and listed to your printer. When the compiler is finished, simply load your BASIC program and make the corrections. Then try to compile your program again. Usually by the second or third try, your program should compile cleanly and you will be all set.

PLEASE NOTE: As listed in the COMPATIBILITY WITH THE BASIC INTERPRETER section of this guide, a few of the BASIC commands cannot be compiled or have certain restrictions. If the compiler encounters one of these commands it will normally display the SYNTAX error message. In some instances (i.e. FN) the compiler will bail out. If you get a SYNTAX ERROR, then check your BASIC line as indicated in the error message. If the compiler dies, then check both the last and previous to the last lines that were compiled. You will either need to eliminate the command or follow the restrictions as noted.

IMPORTANT! IMPORTANT!: If the compiler bails out it may be either graceful or not so graceful. If you get a message on the top of your screen like ILLEGAL STRUCTURE ERROR (5055). Can't continue compiling, the compiler terminated gracefully. (Note that the line number in parentheses is in the compiler, and not in your program). If the compiler just bails out with no warning, then it didn't properly close the disk files it was working with. Therefore, files may be left open on your disk. If this is the case then enter DCLOSE <return> before you do anything. This will close the files. If you forget to do this, then load the diskette directory (DIRECTORY) and if any of the program files are open you will see an asterisk next to the file type. You need to collect (COLLECT) the diskette before you run the compiler again. If you are unsure of what we are talking about review the VALIDATE command (BASIC 2.0) or the COLLECT command (BASIC 7.0) in your Commodore Disk Drive User's Guide.

ERRORS IN THE DOS AND OS

As we said nobody is perfect, and well that includes Commodore. The DOS (Disk Operating System) and the OS (Operating System) are the programs that run your computer and disk drive. They are transparent to the normal user and will normally not cause you any problems. But they do have a couple bugs in them, that will infrequently cause problems to the compiler. We know of two that have shown their ugly faces and if you happen across any more let us know.

BREAK IN nnnn or WARNING THERE IS A BUG IN DOS: One of these type of errors will occur when your program is an exact multiple of 254 bytes long, such as 254 or 508 or 2540. The easiest way to fix this problem, if it occurs, is to pad your program with a dummy statement so that it is not an exact multiple of 254. For example make the last line of your program read **63999 GOTO 63999**. Do not use a REMark statement, because these are not compiled.

DEVICE NOT PRESENT ERROR: We're not sure where this bug is but we think it is somewhere in the DOS or OS. Unfortunately, since we do not know where it is we cannot tell you how to fix it. However, it occurs very infrequently and intermittently. Somehow, the computer loses communication with the disk drive. If this error occurs while you are compiling a program, don't get discouraged. Just try the process again, because it not triggered by the program itself, and is not likely to occur again.

PLEASE NOTE: These errors normally cause an abnormal termination to the SM COMPILER 128 program. Therefore, files may be left open on your disk. See IMPORTANT! IMPORTANT! above to fix this situation.

ERRORS CAUSED BY INCOMPATIBILITY

There are some cases where the compiler will just not be able to compile a portion of your code, and it will bail out. These types of error messages include **ILLEGAL STRUCTURE DETECTED** or **OVERFLOW ERROR**. Possible causes of such errors could be statements that are too complicated, numeric values that lie outside the legal range of the interpreter, improperly constructed loops or incompleted

IF-THEN-ELSE clauses. In such cases the cause of the error will be found in the last or next to last BASIC program line number, that is displayed on the screen. Review your program code and correct the problem.

PLEASE NOTE: These errors normally cause an abnormal termination to the SM COMPILER 128 program. Therefore, files may be left open on your disk. See IMPORTANT! IMPORTANT! above to fix this situation.

ERRORS IN THE COMPILER

Well, last but not least is ourselves. We've tried to make the compiler error-free, but there is a possibility that we goofed. If this is the case we will try to fix the problem pronto, but you have to let us know. Usually, if there is an error in the compiler a message such as **SYNTAX ERROR (5055) Can't continue compiling** will be displayed on the top of the screen. READY and the cursor will also appear. Your BASIC program has triggered an error in our compiler that we did not find during testing. Since the compiler displays the line numbers as it compiles, you will know about where the error was triggered. You could fool around with your code a bit and see whether you can get around the error. If you can you are in luck.

PLEASE NOTE: These errors normally cause an abnormal termination to the SM COMPILER 128 program. Therefore, files may be left open on your disk. See IMPORTANT! IMPORTANT! above to fix this situation.

USING YOUR COMPILED PROGRAM

Once you have compiled your BASIC program successfully, you will have a runtime program called C/program name. Your source code (uncompiled version) has remained untouched and will also be on the program disk. It is strongly recommended that you copy both versions to a backup disk and put it in safe-keeping. Now you are ready to use your new program.

PREPARING YOUR PROGRAM DISK

When you have successfully compiled your program, your diskette will contain not only your original BASIC program, but also the compiled version and two other work files. **FOR EXAMPLE:**

BILLYGOAT	Your BASIC program
A/BILLYGOAT	Work file
P/BILLYGOAT	Work file
C/BILLYGOAT	The compiled BASIC program

The only program you need is C/BILLYGOAT. However, BILLYGOAT is your source code, and you will need it if you want to make any changes to the program. So back it up and put it in safekeeping. A/BILLYGOAT and P/BILLYGOAT are no longer of any use, so they may be scratched off of your diskette. If you are planning on reselling the program and you want to protect your source code then also scratch BILLYGOAT off of the disk.**BUT REMEMBER . . . copy it to a backup diskette first!!**

You also need to copy a compiler runtime module (included on the **SM COMPILER 128** program diskette to your program, as this runtime module is neccessary to run your compiled program. This runtime module is called **RUNTIME....5BH1M.** The easiest way to copy it onto your program diskette is with the COPY utility supplied on your disk drive's TEST/DEMO diskette. This runtime module must be loaded into the computer, prior to running your compiled program.

The command for loading this runtime module is:

POKE 4627,208:BLOAD"RUNTIME....5BH1M",P53248,B0

Now this could be kind of bothersome to do, everytime you want to run your program, especially if others are going to be using it. So we suggest you set up a loader program, that will load both the runtime module and your compiled program.

FOR EXAMPLE let's use the BILLYGOAT program above. You could write the following program:

```
10 REM BILLYGOAT LOADER PROGRAM
20 REM PROGRAM NAME RUNBILLYGOAT
30 POKE 4627,208:BLOAD"RUNTIME....5BH1M",P53248,B0
40 DLOAD"C/BILLYGOAT"
50 END
```

Save this short program on your program diskette. Then whenever you want to run your program simply type in:

DLOAD"RUNBILLYGOAT",D0 <return>
RUN <return>

The EXAMPLE program, included on the SM COMPILER 128 diskette, is setup like this. Take a look at these programs, to get a better idea of how to do this.

LOADING YOUR PROGRAM

Your compiled program will load in the same manner as the uncompiled version. Just type **"DLOAD C/program name" <return>.**

LISTING YOUR PROGRAM

Now, I am sure that you are anxious to see what this program looks like, so type in **LIST <return>.** Presto! Only one line that looks something like:

1985 BANK 0:SYS 53248:MH-COMP 1.0

Since all of your program is in machine code, there is not much to list. This line simply tells the computer where to go to start the compiled program. This is why it is important to keep your source code. You can't change anything in this compiled version. If need be, you have to change the source code and compile again.

RUNNING YOUR PROGRAM

Just like loading your program, your compiled version will run in the same manner. Simply type in RUN <return>. (Make sure you have bloaded the runtime module first, as described above.)

You've taken care of all the logic errors and the compiler has taken care of all the syntax errors, so you shouldn't have any problems . . .RIGHT? Well almost right. There are cases where you will run into errors, usually when working with variables and parameters. These errors will include such problems as ILLEGAL QUANTITY, BAD SUBSCRIPT, STRING TOO LONG or DIVISION BY ZERO, FILE NOT OPEN. What has happened is that you have manipulated a variable to a value that, when used within the context of your program, causes an error.

FOR EXAMPLE:

> Your program includes a line 5055 B = A/C). Somewhere in the program C is set to zero (Ø). You will get a DIVISION BY ZERO IN 5055 error.

You will have to review your source code, to determine where the erroneous logic is, correct the source code (uncompiled version) and compile again.

APPENDIX A

OPTIMIZING EXECUTION SPEED

One of the more common reasons for compiling a program, is to increase the speed of execution. Speed is especially important when working with calculations. The runtime module, which is used to execute your compiled program, differentiates between floating point numbers and integers, not only in variables, but also in calculations. (The BASIC interpreter generally only carries out its calculations with numbers in floating point form.) In order to optimize the speed of your compiled program, the following factors should be considered.

PLEASE NOTE: It is not necessary to understand or use the information in this chapter in order to use the compiler. Simply compiling your program will make it run faster and smoother. You may not be concerned with the extra speed you would gain by implementing the following rules. However, if you are writing a program, and want to optimize the compiler's capabilities, then you should thoroughly review this chapter.

In a program compiled by **SM COMPILER 128**, the addition, subtraction and multiplication operations are performed in INTEGER form, if both operands are in integer form. The result of the operation also will be in integer form, unless outside of the legal range (-65535 to +65535), where the result will then be transformed into floating point form. If at least one of the operands is in Floating point form, then the other will be converted into floating point form before the operation is performed. The result also will be in floating point form.

Two floating point numbers are neccessary for division. Therefore speed will be optimized if both the dividend and the divisor are also in floating point form.

Functions such as **CHR$** or **MID$**, require their parameters to be in integer form. If a floating point form is given, then it must be first converted into integer form by the Runtime Module, before the function is performed.

In general, numbers will be converted in to the form that is needed to perform the operation or function. However, this conversion is only neccessary when the number is not in the form that is needed. And this conversion process wastes time in a compiled program! Therefore, it is important, when writing a BASIC program, to make certain that the values are in their proper form. The following, lists operations and functions with the optimal input types and the resulting output types. PLEASE NOTE: These forms need not neccessarily exist before the operation is carried out; the RUN-TIME module will transform the value into the proper form that is needed at any specific moment. However, as noted above, this transformation requires time.

(I = INTEGER, F = FLOATING POINT, S = STRING)

OPERATION/FUNCTION	RESULT
I + I	I (/ F)
F + F	F
I - I	I (/ F)
F - F	F
I * I	I (/ F)
F * F	F
F / F	F
I = <> . . . I	I
F = <> . . . F	I
I AND I	I
I OR I	I
NOT I	I
SGN(F or I)	I
INT(I or F)	I (/ F)
FRE(I or F)	I
POS(I or F)	I
PEEK(I)	I
LEN(S)	I
STR$(F)	S
VAL(S)	F
ASC(S)	I
CHR$(I)	I
LEFT$(S , I)	S
RIGHT$(S, I)	S
MID$(S , I , (, I))	S
SIN(F)	F
LOG(F)	F

The following, lists operations and functions that generally result in values of integer form

=	PEEK
<>	LEN
<	ASC
>	JOY
<= >=	RDOT
AND	POT
OR	BUMP
NOT	PEN
SGN	RSPPOS
FRE	RSPRITE
POS	RSPCOLOR
RWINDOW	RSPCOLOR
POINTER	

Similarly, BASIC commands require certain types of input values. **FOR EXAMPLE:** OPEN Integer, Integer, Integer, String. It is therfore possible to optimize BASIC commands, by inputting the proper variable types.

When assigning values that exist internally in integer form to variables of the floating point type, a conversion to the floating point type <u>is not</u> performed. These variables can also store integer type values, so that when used later, they will appear as integer types.

For numeric constants, the rule is that if the constant is written with a decimal point or with exponential notation, the number will be of floating point type. Otherwise, the number will be of integer type. **FOR EXAMPLE:** 26 is Integer type, 26. is floating point and 26.ØØ is floating point.

To sum it up, if you want to optimize your BASIC program, then make sure that the numbers or variables are in the form that is needed, prior to the operation, function or statement. This will save time, by eliminating the need for the RUNTIME Module to do the conversion itself.

APPENDIX B

WARRANTY POLICIES

A BACK UP copy of the SM COMPILER 128 is available to the registered owner for $5.00. When requesting a backup, please include the registration number, that is stamped on your original diskette's label.

Any NEW OR UPDATED VERSION of SM COMPILER 128 (should one be released) is available to the registered owner for $10.00. When requesting an updated version, please include the registration number that is stamped on your current version diskette's label.

Please include your registration number and a check for the proper amount and mail your request to:

SM SOFTWARE, INC.
P.O. BOX 27
MERTZTOWN, PA 19539-0027

APPENDIX C

LIABILITY DISCLAIMER

This manual and the software described in this manual are sold on an "as is" basis without warranty as to their performance. SM SOFTWARE, INC. does not warrant this program for any purpose nor does SM SOFTWARE, INC. warrant the accuracy, quality or freedom from errors of this manual and the software described in this manual.

SM SOFTWARE, INC., their distributors, agents and retailers can assume no responsibility for any consequential, incidental or other liability arising from the use, the inability to use or the attempted use of this program nor for any loss of anticipated profits or benefits inccured.

Some states do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation may not apply to you.

The user of this product shall be entitled to use this product for his or her own use, but shall not be entitled to sell or transfer reproductions of the software or manual to other parties in any way.

WHY COMPILE?

For those of you related to the HACKER GNOME, you may already know the answer to this question. But, if you aren't even his second cousin, twice removed, you may be wondering what GNOME SPEED can do! This superb compiler will transform your BASIC program into a P-code that is as sophisticated as any program written in machine code. However, unlike machine code, your program will be compacted up to 50 percent. And since GNOME SPEED proof-reads your program and notifies you of coding errors, your compiled program will be error-free. Imagine! Programs that run up to 10 times faster, are up to 50 percent shorter and are error-free!

HOW DOES GNOME SPEED WORK?

Simply load GNOME SPEED and specify your BASIC program name. Then the compiler takes over . . . converting your BASIC program into a super-fast, super-compact P-code. During this compiling process, GNOME SPEED does as much as possible to eliminate the tedious and time-consuming steps that are normally performed by the BASIC interpreter during program execution. This includes eliminating such tasks as searching for line numbers and variables, and converting mathematical formulas into more efficient formats. You can also include special directives in your BASIC program that will direct GNOME SPEED to produce various testing and referencing aids such as variable lists, cross-reference tables, BASIC line references and special error checking. With these directives, you can produce both testing and final versions of your program. Your compiled program can be loaded, run, copied, renamed, etc., just like any BASIC program. And if you are sales-minded, all your efforts and techniques can remain yours, since only the compiled version, not your original BASIC source code, need be on the disk.

MAJOR FEATURES OF GNOME SPEED:

- Reduces program size by up to 50% and increases execution speed up to 10 times
- Programs can be as large as 1999 lines, 8000 jumps and 500 distinct variable names
- Compiles virtually all BASIC 7.0 / 2.0 commands/functions
- Proof-reads your program for coding errors, with option to list to the printer
- Special directives to produce various testing and referencing aids

REQUIRED HARDWARE

- Commodore 128 computer
- 40 or 80 column monitor
- Commodore 1541 or 1571 disk drive

SM SOFTWARE, INC.
P.O. Box 27
Mertztown, Pa. 19539-0027
(215) 682-4920