UTILITY

LOAD 80

Write the fastest Basic program in town.

A Basic Compiler in Basic

Dr. Dimitri P. Bertsekas
M.I.T., Rm. 35-210
Cambridge, MA 02139

The TRS-80 is not the world's fastest number cruncher. For example, inverting a 15 by 15 matrix takes three and a half minutes. A compiler that translates a Basic program into machine language will make it run much faster.

A compiler to translate ordinary Basic programs is probably the most valuable kind of compiler for the TRS-80. If you have one, you can write and debug your program using the convenient Basic interpreter and then compile the final program to gain a speed advantage. Since compilation is a long process in an inherently slow micro you would rather not compile each time you make a small program correction.

I became interested in a Basic compiler soon after I obtained my computer. I also became interested in exploring machine language and it occurred to me that I could write a simple compiler in Basic without excessive effort. The "Tinycomp" article by D. Bohlke (*80 Micro*, May 1980) gave a Basic program that could translate a few Basic statements into pure machine code.

I intended to add a limited vocabulary of floating point operations to the Tinycomp program. However, I gradually implemented a complete set of single precision operations, including trigonometric, logarithmic, and exponential functions as well as parenthesized expressions. Integer arithmetic, loops and conditional branching came next. Then I introduced a function that exchanges variable values with the interpreter. I finally added Set, Reset, Point, PEEK, POKE and string handling capabilities that allow you to enjoy the spectacular speed advantages of machine language graphics.

The compiler produces machine code that uses subroutines available in ROM. The excellent booklet by J. Blattner and B. Mumford, "Inside Level II" describes these routines and proved invaluable in writing the program. Using ROM routines limits the size of the machine code produced. On the other hand, the ROM routines are slow in part because they include error-checking and handling routines.

The Key Box
Model I
48K
NEWDOS80

The compiler will accelerate your programs by varying amounts depending on their nature. Programs involving primarily integer arithmetic and graphics can be accelerated by 50 to 100 times. Programs involving primarily single precision floating point operations are accelerated by a more modest amount (3 to 20 times is a ballpark figure). Even so, the gain is considerable and it may just keep you from getting into the trouble (and expense) of running your program at the nearest computer center.

I want to warn you that this is *not* a comprehensive professional compiler. There is only one such compiler currently on the market and it costs $195. Less comprehensive compilers cost close to $90. My compiler is written in Basic and is therefore slow (it generates roughly 500 bytes of machine code per minute during compilation). On the other hand, it produces machine code that runs at least as fast as that produced by the commercial compilers (and in some cases faster); it is adequate for many purposes and it is free. Furthermore, it provides you with the challenge of modifying and improving it to suit your purposes better.

What the Program Does

The compiler in Program Listing 1 (lines 501–7200) can translate into pure machine code an ordinary Basic program written and debugged using the interpreter. You can execute the machine code from Basic via a USR call an unlimited number of times during any single program run. Each time, you can pass (using machine or Basic code) an unlimited number of integer or single precision variables from the Basic portion of the program to the USR and vice versa. The compiler translates a fairly complete subset of floating point operations involving single precision variables and one and two-dimensional arrays, together with branching operations and For...Next loops. It also translates a limited subset of integer arithmetic, string handling, and graphics statements. You can use Basic to perform all operations not supported by the compiler and pass control to the machine code at the appropriate times via USR calls. You can save the machine code on disk and load it into RAM when needed. Alternatively you can translate it into data statements, and merge it into a single Basic program with any ordinary noncompiled Basic statements.

I wrote the program on a 48K Model I TRS-80 with NEWDOS80. It works without modification on any 32K or 48K Model I TRSDOS 2.3 system. It works also in a 32K or 48K Level II system provided you modify the DEFUSR0 statement in line 1300. You can also use it to compile small programs in a 16K cassette system provided you make some minor modifications described later. I tried the compiler on a friend's Model III and was pleasantly surprised to find that all my test programs compiled and ran without problems. While I cannot be sure about this, it appears that the compiler can be used on a Model III.

To use the program properly you must learn how machine code stores variables and the nature and syntax of Basic statements that can be correctly compiled.

Variables and Storage

The compiler accepts only three variable types and stores each variable in fixed memory locations. The storage method is identical in all cases to that used by the Basic interpreter.

There are 26 possible integer variables denoted A%–Z%. Each integer is stored in two successive bytes in the memory area VT to (VT + 2*26) where VT is MS minus 2*26 and MS is the end of the storage area set by the user. Thus A% is stored in locations VT and VT + 1, B% is stored in VT + 2 and VT + 3 and so on (see line 1015).

There are 286 possible simple single precision (SP) variables denoted by a single letter A–Z, or by a letter followed by a single decimal number. Thus the possible simple SP variables are A, A0–A9, B, B0–B9, . . . Z, Z0–Z9. Each variable is stored in four successive bytes in the memory area starting at memory location VF (see line 1015). Thus A is stored in VF through VF + 3, B is stored in VF + 4 through VF + 7 and so on. Then A0 is stored following Z, B0 following A0, A1