Radio Shack may have trouble with this one.

Model II Compiler Basic

Model II Compiler Basic
Tandy/Radio Shack
Fort Worth, TX
$199

Larry Clark
Perimeter Data Systems
4449 Oak Trail Drive
Atlanta, GA 30338

As a professional programmer engaged in building large applications systems, I eagerly awaited Radio Shack's Compiler Basic for the Model II. I anticipated a significant performance advantage over the interpretive Basic that comes with the system. In addition, I felt that the ability to distribute programs in object form would provide increased protection against software piracy.

I knew that my existing programs would require modification. Radio Shack clearly indicates that the language differs from interpretive Basic, and suggests that it is primarily suitable for new application development. To achieve higher performance, I expected that the compiler language would require some additional specification. But I was utterly unprepared for what I encountered: gratuitous changes for the sake of change, callous disregard for human factors, and woefully disappointing performance.

The Development System

Programs are normally written and debugged using the Development System. This is a complete environment that closely resembles interpretive Basic—programs can be written, saved, executed, modified, etc., without returning to TRSDOS. There is also a stand-alone editor called BEdit, which allows more powerful editing operations, and a stand-alone run-time package that can execute previously compiled programs.

The editing commands in the Development System are annoyingly different from the familiar Edit command of interpretive Basic. In fact, there is no Edit command; its function is performed by Change, which is better suited to global changes than to intra-line corrections. The BEdit program uses another scheme (including an Edit command), but differs from both of the above mentioned, and from Edit-80, which it closely resembles. This makes it incredibly difficult to switch from the Development editor to BEdit. The difficulty is compounded if the user also works in interpretive Basic and/or Edit-80 (e.g., for Assembly-code or Fortran work).

If you have existing programs in interpretive Basic, forget trying to use them as a starting point. The file format used for storing source programs by Compiler Basic is different from either of the formats used by interpretive Basic. I managed to write a conversion program, but the language differences are so significant that I'd have been better off starting over.

Data Types

The compiler supports only two types of numeric data, integer and real. Integers occupy two bytes and perform as in interpretive Basic. Reals, on the other hand, are stored in floating packed decimal, and occupy eight bytes each. Each real number carries up to 14 significant digits. The use of a decimal base would seem a good choice for accounting work, since it eliminates the possibility of roundoff errors.

Strings have both maximum and current lengths. The default maximum is 255 (as with interpretive Basic). However, if the user knows that the string will never be that long, he can assign a shorter maximum to save space. Unlike interpretive Basic, strings always require enough storage to hold their maximum length. This avoids the interpreter's string compaction delays, but may prevent programs with large string arrays from fitting into memory.

Variable names may be arbitrarily long, and the first six characters are significant. Thus, Joe and Joseph are different variables, although Joseph and Josephine are the same. A variable's type may be set implicitly (Real is the default), or specified by appending a type tag ($, %, or #). The first mention of a variable is binding—if the first line of a program refers to A$, the compiler treats every reference to A as a string, regardless of the presence of the $ tag, and the variables A% and A# are not available.

Arrays of all data types are available, but may not have more than two dimensions. Although relatively few programs need more than this, those that do become extremely awkward. The limitation seems arbitrary.

Needless Differences

Some of the other differences seem pointless. Their primary effect is to frustrate users who are accustomed to interpretive Basic.

- Quotation marks are not used around filespecs on commands such as Save and Kill.
- Load will load only an object file, not a source file. You must use Old for source files.
- LLIST <range> is replaced by List <range> {PRT}. The braces are mandatory.
- In concatenating strings, the plus sign has been replaced by an ampersand.
- In debugging a program, you cannot use Print <var> to see its value; you must Display <var> or DI <var>).
- RENUM is not recognized, but Renumber and RE are.
- Print @(6,3),<list> must be written as Print CRT(6,3);<list>.
- To determine the cursor position,

"A close reading. . . reveals that Compiler Basic is not a true compiler."

ROW(x) and POS(x) have been replaced by CRTX and CRTY. (This would be fine if they hadn't reversed the normal usage of the coordinates.)
- The End statement marks the end of complication, not the end of execution.
- The INSTR function has been relabeled POS, and accepts only the two-argument form.
- The MID$ function has been renamed SEG$.

The above list is far from complete, but indicates the nature of the changes. If the compiler were to be used in a vacuum, the choice of syntax wouldn't matter. But every Model II is already supplied with interpretive Basic, and most potential users of Compiler Basic are probably already using it. (If they wanted to start from scratch, they would be better off using a language other than Basic.) So these changes simply make life needlessly difficult.

Input/Output

The file I/O commands are also markedly changed, but in this case there is some justification. Compiler Basic provides a rich variety of I/O options, including fixed or variable-length records; sequential, direct, or indexed access; and stream, formatted, or binary formats. However, most of the interesting combinations seem to waste an inordinate amount of space.

Direct access files automatically include two bytes of overhead per record—one for the record length and a second that's unused. (The record length indicates how much of the record is actually used, even though the physical records must be fixed in length.)

ISAM files also have two bytes of overhead per record, but round the storage to the next larger multiple of 32 bytes. Thus, a file with a record length between one and 30 bytes requires 32 bytes per record.

Numbers in formatted files are written in ASCII. Thus, an integer may take six bytes, including sign.

Real numbers within binary files carry a length byte in addition to the internal representation. This means that nine bytes may be (and therefore *are*) required to store a number.

The ISAM format was apparently dictated by considerations of (limited) compatibility with the Cobol compiler. Considering how much these two languages have in common, I wonder whether (or why) anybody would even want to try to marry them.

Program Segmentation

Perhaps the most welcome feature of Compiler Basic is that it allows true subprograms, which can be called by name (not by GOSUB <line>) and passed arguments. Except where declared as Common or as formal parameters, the variables in the subprogram have nothing to do with those in the calling program. "At last!", I thought, "I can actually write programs with a reasonable degree of modularity and structure." Wrong!

A close reading of the manual reveals that Compiler Basic is not a true compiler. It does not convert source programs into machine code, but rather, into an intermediate pseudo-code, which still must be interpreted at run time. (This is also true of most implementations of Pascal.) The pseudo-code is more compact than true object code would be.

Subprograms can be compiled either separately or in conjunction with the main program. If they are compiled separately, they can be loaded together before beginning execution, but the result can only be executed, not saved. The pseudo-code cannot be linked with true object modules. It is still possible to invoke machine code, but the technique is similar to the USR calls of interpretive Basic—the code must be preloaded into high memory that has been reserved, and the entry point addresses must be coded into the program.

The inability to combine independently compiled modules into a single executable whole makes the concept of separate compilation virtually useless. In designing large systems for beginners, I want to give them simple directions, such as "Enter the command Do Daily and follow the prompts." I certainly don't want to tell them to enter RSBasic, load a half-dozen modules, then say Run. Worse yet, if they have only the run-time system (sold separately for just such applications), they can't load multiple modules even if I wanted them to. So without a linkage mechanism, all necessary subprograms must be compiled at once.

This is easier said than done. The Development System requires room in memory for the entire source program it is compiling, plus the compiler itself. (I suspect it may also keep the object code in memory, since I was able to compile an oversize program by reducing some dimensions.) Consequently, the largest program that can be compiled in one shot is limited, and I reached the limit on my first attempt.

Chaining

So we can't combine modules that were compiled separately, and we can't compile a very big collection of modules at once. What's left? Chaining.

The program chaining facility is a mild improvement over what's available in interpretive Basic. It allows you to save the contents of certain variables in Common, where they can be retrieved by programs later in the chain. Except for this feature, it is no different from the Run program statement in interpretive Basic.

I thought there might be some smaller applications that could benefit from the performance improvements that come from compilation. To see how much improvement the compiler produced, I wrote a simple benchmark to find out.

```
Interpretive Basic       Compiler Basic

10 DEFINT A-Z            10 INTEGER
20 PRINT TIME$           20 PRINT TIME$
30 FOR I = 1 TO 1000     30 FOR I = 1 TO 1000
40 X = (I*3 + 6)/7       40 X5(I*3 + 6)/7
50 NEXT I                50 NEXT I
60 PRINT TIME$           60 PRINT TIME$
70 END                   70 END
```

This seemed like a fair test, since both systems would be doing arithmetic using the same internal number format. Upon running these test cases, I was amazed by the relative performance of the two systems: Interpretive Basic, 13 seconds; Compiler Basic, 22 seconds (plus 14 seconds compilation).

Where I expected at least a threefold improvement (probably more like tenfold) in performance, I actually achieved a degradation of about 70 percent. Either Microsoft does things awfully well, or Ryan and McFarland (the authors of Compiler Basic) are doing something terribly wrong!

Conclusion

I have never bought a program with such high expectations, and wound up feeling so totally ripped off. It is inconceivable to me that a compiled program—or even a partially compiled one—can perform so poorly.

I am equally appalled by the unnecessary incompatibilities between this and the standard version of Basic for the Model II. It shows an utter lack of concern for the people who will inevitably use both.

If the compiler is intended for developing applications programs, the authors should recognize that program linkage is a necessity, and that disk space may be a precious resource. In any case, there is little rationale for needless overhead bytes.

After my first experience with the compiler, I put it on the shelf and dismissed it as useless. Several months later I brought it back out for another try, hoping against hope that I had missed something crucial. It appears that I hadn't.

There may be a few limited applications where Compiler Basic might be useful. If you know of one, please contact me. I'm willing to sell my copy at a sizable discount.■