ACCEL3 COMPILER for TRS-80 BASIC (C) COPYRIGHT SOUTHERN SOFTWARE 1982

ACCEL3 is Southern Software's latest compiler for TRS-80 BASIC. ACCEL3 produces more compact code than ACCEL and ACCEL2, it compiles faster, its treatment of FOR-NEXT allows for badly-structured loops, it optimises more language constructs, and it supports SAVE, LOAD, RUN, CSAVE and CLOAD of compiled programs. It will compile the full DISK BASIC language, and will tolerate many non-Tandy language extensions (though not all).

ACCEL3 occupies less than 5632 bytes. This relatively low size is achieved by a technique of selective compilation. For instance I/O statements such as LPRINT or INPUT are not translated at all but remain in the compiled program in their source form, and are executed by the resident BASIC interpreter. Statements involving INTEGERs and flow-of-control statements (GOTO, GOSUB, RETURN) are, by contrast, translated to directly-executed Z80 machine-instructions. Other, more complex statements are translated into calls to ROM routines. ACCEL3 selects more statements for translation than ACCEL, notably those involving STRINGs and SINGLE and DOUBLE data-types. ACCEL3 also translates references to array elements (even when the array has dynamic bounds), and it translates more functions than ACCEL and ACCEL2.

ACCEL3 SUPPLIED ON TAPE.

If you have purchased ACCEL3 on diskette, skip the next four sections.

The tape supplied is self-relocating. This gives you the freedom to load the compiler anywhere in memory you please. It also provides the freedom to make mistakes, so please check all address arithmetic carefully. You need only perform the installation operation once, and then you can take your own back-up copies on tape or disk, for subsequent direct loading.

For both Model I and Model III you must load the tape supplied under Level2 (not DISK BASIC) using the SYSTEM command, and at the low cassette data rate. (On Model III specify L at bring-up). It will load itself at locations 18944 and up, and then, under your control, will relocate itself to any chosen location above this. The compiler will not run correctly unless it is loaded in PROTECTED memory. Depending on how much RAM you have, and on what other machine-language programs you want resident, decide where you want to locate it. The table overleaf gives addresses that are suitable if you want to load the compiler as high in memory as possible. For the main text, we will assume as an example that ACCEL3 is loaded on a 32K machine. In this case your answer to the initial MEMORY SIZE? question will be calculated as follows-

```
49152     (Upper limit of memory on a 32K machine)
-5632     (Size of ACCEL3 compiler)
-----
43520     = "SA", starting address of the compiler.
-----
```

1) On Video Genie (PMC-80) you don't get the MEMORY SIZE? prompt. However, on power up the machine gives you the opportunity to enter a number after the first READY?. This is exactly the same number referred to as "SA" in these examples.

2) If you are going to use TSAVE to make either a tape backup copy of ACCEL3, or to save the compiled program, then allow a further 512 bytes of protected memory, i.e. set SA=43008 instead.

southern software

PO Box 39, Eastleigh, Hants, England, SO5 5WQ

TABLE OF USEFUL MEMORY ADDRESSES.

The first table gives values of addresses you can use in order to locate ACCEL3 as high in your machine as possible, (assuming you have no other machine-code programs above it).

16K	32K	48K	
32767	49151	65535	Top-of-memory address
27136	43520	59904	Start of ACCEL3, and MEM SIZE
26624	43008	59392	MEM SIZE,(leaving 512 bytes for TSAVE)
27136-32767	43520-49151	59904-65535	BACKUP range to save ACCEL3
27136-28671	43520-45055	59904-61439	Run-time routines

This second table gives address ranges you can use when compiling programs to sell on a smaller machine. ACCEL3 has a run-time component of only 1536 bytes, and this table shows you how to ensure that only the minimum amount of ACCEL3 resides in the end-user's smaller memory.

16K	32K	Target machine size
31232-36863	47616-53247	ACCEL3 address range, on your machine
31232-32767	47616-49151	Run-time routines, in end-user's machine

LOADING THE TAPE SUPPLIED.

1) MEMORY SIZE? 43520 (enter) (or your value of "SA").
   RADIO SHACK LEVEL II BASIC
   READY
2) SYSTEM (enter)
3) *? ACCEL3 (enter)
4) Tape loads...
5) *? / (enter)
6) TARGET ADDR? (enter) (or any chosen protected address).
   READY

Notes.

1) The tape loading process at step 4 is a standard core-image tape load, and is subject to the usual variability on volume, head alignment, etc. A pair of asterisks will blink on the display. If no asterisks appear, or if a C is displayed, then there has been a loading error. Retry from step 2 with a different volume setting. (Southern Software tapes are recorded at a lower volume than is generally common, since this gives a wider tolerance to fluctuations). Two copies are supplied on the tape, in case one gets damaged. Damage is almost invariably due to either a tape kink (a tiny fold in the tape) or to recorded noise, caused by RESETTING the computer in the middle of a tape load. (Always stop the cassette player before hitting RESET on a bad load).

2) Step 6 lets you relocate the compiler anywhere in protected memory. If you just hit enter, then it relocates to SA, the answer to the MEMORY SIZE question. If you have other machine-code programs you want to hold concurrently in memory, then you may wish to respond with a value different from SA.

3) On Video Genie, for (enter) read NEW LINE key.

4) Some other products, e.g. the EXATRON stringy floppy, actually modify the PROTECTED MEMORY value. While not causing an error this can be very confusing, since it could cause ACCEL3 to relocate to a different address than you expect. If in doubt use an explicit target address at step 6.

southern software

PO Box 39, Eastleigh, Hants, England, SO5 5WQ

SAVING A BACK-UP.

You can now save your own, fixed-location back-up copy of ACCEL3. This can be on tape, disk, or wafer. It will be shorter than the original file, but more important, it will load under either Level 2 or DISK BASIC, and it will load directly at its final location, without corrupting location 18944 and up. The loading process can also automatically activate the compiler.

A) Backup on tape.

To prepare a core image tape, on Level 2 or DISK BASIC, you will need either the Southern Software utility TSAVE, or TRS TBUG. TSAVE is recommended, because it allows you to work in decimal, not HEX, and to check the saved tape. In this example the memory range you need to save is 43520 to 49151

Using TSAVE, respond as follows:

```
FILENAME? BACKUP (enter)        (or your own file name)
RANGE? 43520,49151 (enter)
RANGE? (enter)
START? 43520 (enter)            (start address, to activate compiler)
R (enter)
Tape records...
Reposition, and type C, to check tape.
```

This tape can be reloaded by:

```
SYSTEM (enter)
*? BACKUP (enter)               (or your own filename)
Tape loads...
*? / (enter)                    (ACCEL3 is now activated, see later)
READY
```

Note. On Model III, under DISK BASIC, the SYSTEM command does not function correctly. In effect you cannot load SYSTEM tapes while in DISK BASIC.

B) Backup on Disk.

Go into TRSDOS (or NEWDOS, etc.) from Level 2 by hitting RESET. (This will not destroy the stored image of the compiler).

Type DUMP ACCEL3/CIM (START=43520,END=49151,TRA=43520) i.e. SA to SA + length of ACCEL3.
(For Model III, these addresses must be converted to hex).

This file can be reloaded under TRSDOS by typing LOAD ACCEL3/CIM. When you enter Disk BASIC after loading ACCEL3/CIM set MEMORY SIZE to 43520, or to your value of SA. See also later section on LOADING THE COMPILER FROM DISK.

C) Backup on Wafer.(EXATRON Stringy Floppy commands assumed).

The address to save is 43520, with length 5632, and with autostart 43520. Because 43520 is not representable as an INTEGER, you will have to type it in modulo 65536, i.e. as -22016.

So type @SAVEn,-22016,5632,-22016

This can be reloaded (under Level 2 or DISK BASIC) by @LOADn

Southern Software

PO Box 39, Eastleigh, Hants, England, SO5 5WQ

ACCEL3 SUPPLIED ON DISKETTE.

ACCEL3 is supplied on disk in relocatable format. You must use the disk appropriate to your system, i.e. Model I (including Video Genie), or Model III. If you have received the wrong one, return it immediately for a replacement. The final relocated program is the same on all machines, it is only the disk format that differs. Once installed, you can CONVERT the saved file from Model I to Model III, but you cannot CONVERT the product disk. If you have been supplied with a double-sided disk, then the Model I format is on the front (the labelled side), while the Model III format is on the back.

1)Put the ACCEL3 disk in drive 0.

2)Press the RESET button (i.e. BOOT the system).

3)The COPYRIGHT notice will appear. Press "ENTER" to install ACCEL3.

4)Answer the address relocation question. To load ACCEL3 at the highest suitable address on your system, simply hit ENTER. You need only bother with an explicit address if you have other machine-language routines you want to have loaded at the same time as ACCEL3, or if you are preparing programs compiled under ACCEL3 to run on a system with a smaller memory size than yours.

5)Note down the address at which the computer says ACCEL3 is loaded. Use this as MEMORY SIZE under BASIC.

6)Remove the ACCEL3 disk and place in drive 0 your own system disk, i.e. one containing TRSDOS or NEWDOS, etc.

7)Press ENTER. (Do not REBOOT!)

8)The system will now appear to REBOOT, i.e. the operating system will load. If you have an AUTO command in effect, this will be displayed on the screen, but it will NOT have been executed.

9)The relocated core-image file, named ACCEL3/CIM, will be DUMPed automatically on your system disk. The size will be slightly smaller than 5632, which is a published upper limit.

10)The DUMP command has an incompatible format on many operating systems, e.g. DOSPLUS. If DUMP fails at this point, or your operating system is incompatible with TRSDOS and DUMP fails to appear, simply type the DUMP command by hand, using the correct address formats. Alternatively, do the DUMP on to a TRSDOS disk, and then COPY or CONVERT the file to your DOSPLUS or NEWDOS80 disk, etc.

LOADING THE COMPILER FROM DISK.

To use ACCEL3 on future reboots, you can type LOAD ACCEL3/CIM under TRSDOS (or NEWDOS, etc.) and then enter DISK BASIC in the normal way, specifying MEMORY SIZE to protect ACCEL3. Once in DISK BASIC you will have to activate the compiler, by branching to its first location, as described later. Although the dumped file, ACCEL3/CIM, specified a TRAnsfer address, it's no good executing this branch under DOS (by using the file as a command). When you enter BASIC the compiler would get deactivated. Unfortunately, loading the compiler under DOS has the danger that the invocation of BASIC itself may corrupt high memory, destroying what you've just loaded. TRSDOS on Model I corrupts the top 64 bytes, while TRSDOS on Model III is even worse. The DO command, if used, endangers the top 500 bytes, but also there is a chance that if the timer interrupt fires while BASIC is setting up its stack, then even lower bytes may be corrupted. And of course other DOS's may have their own quirks.

1) If on TRSDOS, on Model I, always leave the top 64 bytes unused. LOAD the compiler under DOS, enter BASIC, and then activate it by branching to its first location.

2) If on TRSDOS, on Model III, do not risk loading under DOS. Instead enter BASIC (setting the correct protected memory address) and then load the compiler by CMD "I","ACCEL3/CIM" which loads and branches to the compiler's first location, thus activating it automatically.

Southern Software

PO Box 39, Eastleigh, Hants, England, SO5 5WQ

CREATING A SAMPLE PROGRAM.

Despite the triviality of the following example, it does illustrate most of the mechanics of compiling and saving a compiled program, and it should be followed through to completion.

Under Level 2 or DISK BASIC type

```
AUTO
10  'SAMPLE
20  DEFINT I-J
30  FOR I=1 TO 1000:NEXT
40  A$ = A$ + "X"
50  PRINT J; A$;
60  J = J + 1
70  IF J<5 THEN 30
80  STOP
```

List the program, check it, run it, and change it, if necessary. Once you have compiled it, you will no longer be able to EDIT it. So SAVE it on tape, or disk.

COMPILING THE PROGRAM.

You've got to this point in the scenario after first installing ACCEL3 from a Southern Software tape or disk, and you may have made your own backup copy on tape, disk, or wafer. If you are starting from scratch after a reboot, then, given a backup on tape, Model III disk, or wafer, you should get into BASIC first, with memory correctly protected, (either Level 2 BASIC or DISK BASIC). Now load your backup and the compiler will be automatically activated because:

1) / (enter) after tape load branches to the START address.
2) CMD "I","ACCEL3/CIM" branches to the TRAnsfer address.
3) @LOADn branches to the autostart address.

Alternatively you may be running directly from the product tape, i.e. you loaded the tape under Level 2, and you now want to compile the sample program. Or, you may have loaded ACCEL3/CIM from disk under DOS on Model I, and you have now entered DISK BASIC. In both these cases you must first "activate" the compiler by branching to its first location. Type

```
SYSTEM (enter)
*? /43520   (or your value of SA)
READY
```

The SYSTEM command does not work on Model III, under disk BASIC. An alternative is

```
DEFUSR=&Hstarting-address-in-hex   (or DEFUSR=43520-65536, i.e. SA-65536)
?USR(0)
```

Once ACCEL3 is activated you can execute its builtin commands which are BASIC keywords, preceded by a slash (/). (Under NEWDOS80 or DOSPLUS, precede the slash by a blank). To compile, type:

```
/FIX (enter)   (i.e. FIX program in machine-code)
ACCEL3 (C) COPYRIGHT SOUTHERN SOFTWARE 1982
116 98 141     (These three values are the changing program size)
READY
```


Use of the word FIX is intended to remind you that your BASIC program has now been irreversibly converted to machine-code. You cant EDIT it in any way, but you can LIST it. Shown in comparison with the original, it will look like this:

Before Compilation	Compiled by ACCEL3
10 'SAMPLE	10 :
20 DEFINT I-J	20 DEFINTI-J:
30 FOR I=1 TO 1000:NEXT	
40 A$ = A$ + "X"	
50 PRINT J; A$;	
60 J = J + 1	
70 IF J<5 THEN 30	
80 STOP	80 STOP

Notes.

1) Lines in the program that have been converted to machine-code do not appear in the listing. (The actual machine-code itself follows the dangling :, but is unprintable).

2) I and J were defined as INTEGERs in line 20, and as a result the machine-code compiled by ACCEL3 will be much faster than if they had been float variables (SINGLE or DOUBLE).

3) ACCEL3 compiles line 40, the STRING assignment, although ACCEL would not.

4) DEFINT, and STOP were not compiled, but the run-time environment is smart enough to ensure that the BASIC interpreter is passed control for these statements, and that its understanding of any variables they refer to is the same as that of the compiled code.

RUNNING THE COMPILED PROGRAM.

```
RUN (enter)
0 = 1 XX 2 XXX 3 XXXX 4 XXXXX (program runs)
BREAK IN 80
READY
```

A second RUN will rerun the program. GOTO 10 or GOTO 20 will reenter the program without resetting J to 0 or A$ to null. GOTO 30, or a reference to any of the lines that have disappeared will result in an UNDEFINED LINE NUMBER message. RUN it again, but hit BREAK to interrupt the program before completion. Note that this throws you into READY, without the BREAK IN N message. Type ?I;J;A$ to interrogate the current values of the variables. CONT will not work after BREAK. In a larger program the BREAK key may arbitrarily "take" in a compiled line, or in an interpreted line. In the latter case, CONT will work. In either case the variable values are correct. Type J=2, and then GOTO 10 to restart execution, with a modified value of J. Turn trace on by typing TRON, and rerun the program. Only the uncompiled lines are traced. Turn trace off again with TROFF.

Once you have compiled a program, you can no longer use the commands EDIT, AUTO, DELETE, NAME (renumber), or MERGE. This is because the machine-code in the compiled lines may contain bytes that are treated as control codes by the interpreter. So use of these commands may cause an infinite loop, or a machine reboot. To get the machine back to its normal, editable state, you must use NEW or CLOAD, or in DISK BASIC, LOAD or RUN "program-name". All of these destroy the compiled program.

southern software
PO Box 39, Eastleigh, Hants, England, SO5 5WQ

SAVING THE COMPILED PROGRAM.

A) On tape:

1)Type CSAVE "A" (or any other file name).

2)Rewind the tape, and check it with CLOAD?

3)Type CLOAD, and RUN to reload, now, or at a later date.

Notes.

1)To CSAVE or CLOAD a compiled program, ACCEL3 must be active. Otherwise the results will be unpredictable.

2)You can CSAVE or CLOAD an uncompiled program, with ACCEL3 active, without any restrictions.

3)You must have exactly the same environment in effect when you reload a compiled program, as when you saved it. ACCEL3 must be in the same place, you must be running under the same operating system (TRSDOS, NEWDOS, Level2, etc), and you must have specified the same number of disk I/O buffers.

B)On disk:

1)Type SAVE "PROG" (or any other FILESPEC).

2)Type LOAD "PROG" to reload.

3)Type RUN "PROG" to load and run.

Notes.

1)You must have ACCEL3 active to SAVE or LOAD a compiled program.

2)You must have exactly the same environment in effect when you reload a program, as when you saved it. ACCEL3 must be in the same place, you must be running under the same operating system (TRSDOS, NEWDOS, etc.), and you must have specified the same number of disk I/O buffers.

3)File error handling is done by the operating system, which may produce messages, e.g. FILE NOT FOUND.

4)The source file of a program, and the SAVEd compiled program are two very different things. It's easy to inadvertently SAVE a compiled program using the same name as the source. If you do this, your source is lost for ever. As a discipline, use "PROG/BAS" for the source file, and "PROG/ACC" for the compiled file.

MEMORY MAP.

1)ROM AND SYSTEM CODE. This includes the 12K ROM supplied with your machine, the display and keyboard memory-mapped I/O, system control blocks, and the Disk Operating System, if used. The upper address lies between 17000 (under Level2) and 28000 (under TRSDOS, depending on number of I/O buffers).

2)BASIC PROGRAM. The program is compiled in-place by ACCEL3. Depending on the number of comments and blanks, you may find that your program either expands during compilation, or contracts. An expansion is the norm.

3)SCALARS. This is a table of non-array variables, including the names of the variables, their types, and their values (except STRING values). For an uncompiled program, the SCALARS are destroyed by RUN or CLEAR, and then rebuilt incrementally, as used. But compilation builds this table permanently, and compiles references to it. This area effectively becomes a part of the program, and it is saved on tape or disk, when the program is saved.

4)ARRAYS. This is a table of array variables, and it is built incrementally both by the interpreter and the run-time routines in ACCEL3. But ACCEL3 remembers the address of each array, after the first reference to it.

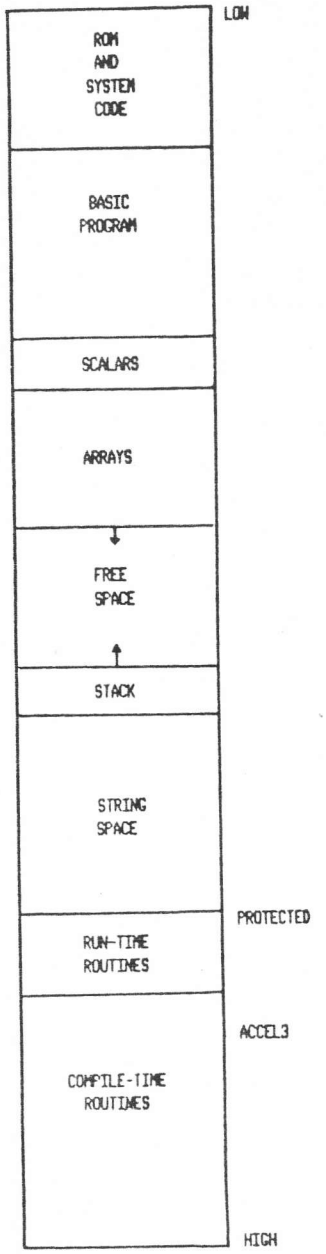
5)FREE SPACE. This is what's left between the top of the arrays, and the bottom of the stack. When they meet, OUT OF MEMORY results. Note that compiled code may fail to diagnose OUT OF MEMORY correctly.

6)STACK. This is used for expression temporary results, for calls within the run-time routines, for FOR-NEXT and for GOSUB-RETURN. The compiled code generally uses less stack than the interpreter, although neither uses a great amount.

7)STRING SPACE. This is where string values go. It's the same size (the value set by CLEAR N) in either a compiled or an interpreted program. However its use is not identical, and ACCEL3 will generally reduce the frequency at which garbage collection is necessary when strings of equal length are used.

8)RUN-TIME ROUTINES. These must be in protected memory, and they must be in the same place when you attempt to load and rerun a compiled program, since the program contains direct references to this code. If you sell or give away compiled programs to a third party, you must include these routines.

9)COMPILE-TIME ROUTINES. These routines convert the BASIC statements into machine-code. They are not necessary at run-time, and indeed must not be sold or given to any third party. If you compile a program on a 48K machine to run on a 16K or 32K machine, then you should arrange that these routines lie just above 32768 and 49152 respectively. This will maximise the space available to the running program.



southern software

PO Box 39, Eastleigh, Hants, England, SO5 5WQ

MORE ON COMPILER ACTIVATION.

The TRS-80 Level2 code in ROM provides a table of transfer addresses through which flow passes at certain key points in execution. ACCEL3 uses 3 of these to get control in the following situations:

1) At the beginning of execution of each program statement.
2) At the beginning of execution of each direct command.
3) After execution of RUN, NEW, CLEAR, LOAD, and END.

When you branch to the starting address of ACCEL3 it enables these traps by putting its own addresses in the transfer slots. Because ACCEL3 then gets control on each command or statement, it is able to support new commands of its own, which it chooses to distinguish with a / prefix. Many other products use a similar technique. ACCEL3 attempts to coexist with these products by preserving the original transfer address, during activation, and branching to it, when it has finished its own work. Other products you may want to use in conjunction with ACCEL3 may not be so kind, but may simply overwrite the original transfer address with their own, thus "disabling" any other product playing the same trick. If you encounter this problem, solve it by activating ACCEL3 last.

Inadvertent reactivation is ignored by the compiler. However, once a switch has been enabled, it would be a disaster if the compiler were destroyed in memory while the switch was still active. So ACCEL3 supports a command, /RESTORE, which will reset the transfer values to their original values, i.e. will deactivate itself. Use this before you overwrite ACCEL3 with another program. Otherwise you can leave it active indefinitely. You can switch back into TRSDOS, and then re-enter BASIC without destroying ACCEL3. However you will have to reactivate it, even if you use BASIC ↑ to preserve a compiled or uncompiled program. You can also load the core-image of ACCEL3 from within BASIC, if you are running under NEWDOS or Model III TRSDOS, provided it loads into protected memory.

At run-time ACCEL3 determines whether or not the resident program is compiled by looking for a first line consisting only of a single colon (:). So no source program may start in this way. When ACCEL3 gets control at the beginning of a BASIC statement, the decision to execute in-line code, rather than to leave a statement to the interpreter is based on detection of a colon, followed by line-end. Once ACCEL3 has made the switch to in-line code, this code runs uninterpreted through one or more statements or lines, until the next uncompiled statement is encountered. INTEGER operations, GOTO, GOSUB, and RETURN are therefore uninterruptable, except by reboot. However, non-integer assign, NEXT, array referencing, SET, RESET, POINT, and PRINT, all contain a "fast" test for the BREAK key. This throws execution back to READY, and the program is not CONTinuable. This trap is not affected by BREAK disable, and if you want to suppress it, then you should POKE byte SA+7 with a RETURN instruction (X'C9', decimal 201), where SA is the Starting Address of ACCEL3.

CHAINING PROGRAMS FROM DISK.

ACCEL3 allows you to chain programs together, i.e. to proceed through a sequence of routines, each invoking the next from disk, and being overlaid by it. (It does not support the NEWDOS80 option which preserves variable values aross RUN). The chained programs may be either compiled or interpreted, or a mixture. You will need to debug these segments in an arbitrary order, compiling each one after it is checked out, and you will not want to change the chaining program, when the program it chains to is compiled.

The best way to achieve this is as follows. Adopt the convention that source programs are named e.g. PROG1/BAS, while the compiled version of the same program is PROG1/ACC. Since you want your final set-up to run compiled, use RUN "PROG1/ACC" etc. anywhere a chaining statement appears in a program. While debugging, simply store a double copy of each source program, one as BAS and one as ACC. So initially the whole system runs uncompiled. Now, when PROG1 is debugged, save its compiled version over the top of PROG1/ACC. Your total system will run as a mixture of compiled and uncompiled routines, while you gradually check out and compile the various sections.

SELLING COMPILED PROGRAMS.

One of the major attractions of a BASIC compiler is that it enables you to write BASIC programs for sale which, with care and tuning, can be comparable in performance with machine-code programs. Secondly, and no less important, a compiled program is very difficult to steal. It can be copied, of course, since any file can be copied byte for byte, but it cannot be modified, except by the owner of the original source BASIC. And of course you dont have to release this when you sell a compiled program.

Although tape is an unpopular medium, it has a number of very significant advantages. Cassettes are very cheap, and therefore expendable or replaceable. They are small and light to post, and will survive violent handling, unlike diskettes which need a lot of protection. Finally the TRS-80 built in SYSTEM command is part of ROM, and therefore consistent on all machines, and it is powerful enough to load any number of core-image segments directly into RAM, without restriction.

So if you can ship on tape, do so. To produce a self-contained tape you have to save the three address ranges

a) Control storage, (including program size, memory size, etc).
b) The program itself, including its dictionary of scalar (non-array) variables.
c) The ACCEL3 run-time routines which interface the running program to interpretive BASIC.

To save these ranges you will need Southern Software's TSAVE utility (TBUG is not satisfactory). Relocate TSAVE in a separate area of protected memory, or better, prepare an absolute-address copy which will load on top of the compile-time routines, after compilation is complete. Invoke TSAVE and give the following responses-

```
FILENAME? MYPROG
RANGE? 16512,16863        (save control storage)
RANGE? 16548↑,16635↑      (save the compiled program)
RANGE? 43520,45055        (save the run-time routines, i.e. SA to SA+1536)
RANGE? (enter)
START? 6681               (dummy start address)
R                         (record)
```

Notes.

1)Locations 16512 to 16863 contain control information such as program start and end addresses, dictionary size, MEMORY SIZE, etc. So when the tape is reloaded MEMORY SIZE is automatically set to what it was when the tape was saved. Also, ACCEL3 is automatically activated.

2)16548↑ to 16635↑ means save the range defined by the values contained in these locations. This includes the program itself, and its dictionary of scalar variables, but not the array variables.

3)To run the compiled program you must have the ACCEL3 run-time routines available, and in the same place as when the program was compiled. These routines constitute the first 1536 bytes of ACCEL3. So the values in this third range depend on where you originally decided to load the compiler.

4)If you want the final program to run on a smaller machine than yours, then you should arrange that the 1536 bytes of run-time routines fall just within that smaller memory. See the earlier table.

5)On Video Genie, use the ESCAPE key for upward arrow.

Whether on tape or disk, do NOT save the whole of ACCEL3, or you will be regarded as infringing the copyright. Also, you must give an acknowledgement in your program documentation that it was compiled by Southern Software's ACCEL, ACCEL2, or ACCEL3.

southern software

PO Box 39, Eastleigh, Hants, England, SO5 5WQ

On disk the situation is not so simple. The DUMP routine provided under TRSDOS (or NEWDOS) will only save a single contiguous core image, and it cannot save any range below HEX'7000'. These two restrictions make it more difficult to sell a compiled program as a single file on disk. What you must do instead is to SAVE the compiled program as described earlier, as a single file, "PROG/ACC" say, and also to DUMP on the sale diskette the core-image of the run-time component of ACCEL3, as a separate file, LOADER/CIM, say. (Again, do not save the whole compiler). This core image is the first 1536 bytes of wherever you have located ACCEL3.

As an example suppose you want to sell a program "PROG/ACC" to run on a 16K machine (although you have a 32K machine). The full sequence is as follows:

1) The required location for ACCEL3 is 32768-1536 = 31232. Under Level2 set this as MEMORY SIZE, load the original self-relocating version of ACCEL3, and locate it at 31232.

2) Return to TRSDOS and DUMP this version of ACCEL3 as a core-image file for your own use.

3) Enter DISK BASIC setting the NUMBER OF FILES to whatever will be required by PROG/ACC, and the MEMORY SIZE to 31232 again.

4) LOAD the source for PROG/BAS, compile it, and then SAVE the compiled program as PROG/ACC as described earlier, but onto a new master disk.

5) Return to TRSDOS and DUMP the run-time component of ACCEL3 (just the first 1536 bytes) on this new master disk, as a file called LOADER/CIM, say. I.e. DUMP LOADER/CIM (START=31232,END=32767). Use Hex addresses on Model III.

6) This master disk will now contain two files PROG/ACC and LOADER/CIM, not the full core-image of ACCEL3. TRS BACKUP is now a convenient way of making copies of this disk for sale.

Your operating instructions must now include the following directions to the end-user. (Alternatively, you can automate the procedure with the use of Southern Software's Command-List processor, EXEC).

1) From TRSDOS load the run-time routines by LOAD LOADER/CIM.

2) Enter DISK BASIC setting NUMBER OF FILES to N (the number you used earlier) and MEMORY SIZE to 31232.

3) Activate the loader by SYSTEM (enter) and #? /31232 (or DEFUSR under Model III).

4) Run the compiled program by RUN "PROG/ACC".

Cautions:

You may want to provide different instructions for Model III users, e.g. to load and activate LOADER/CIM from within BASIC using CMD"I". Also, depending on which operating system your end user will have, you may need to leave e.g. 64 bytes free at the top of memory, to avoid overwriting.

EXECUTION PERFORMANCE.

The aim of using a compiler is to improve execution speed. But the compiler cannot do better than the machine on which the program runs. The Z80 CPU chip is remarkably cheap, reliable, and fast, but it lacks many common operations (such as multiply and divide). These have to be executed via calls to ROM routines which provide the required function (e.g. multiply by successive additions), and this is of course relatively slow. The complex table at the end of this section is a guide to what features can be improved by compilation, and by how much. It remains one of the programmer's tasks (unfortunately), to match the requirements of the problem to the capabilities of the underlying computing system. The extra effort needed to optimise performance could be thought of as a form of machine-code programming. It can produce results comparable in performance with real assembler language coding, but it is incomparably easier, because debugging is in BASIC, using PRINT statements, TRACE, etc.

The result of compilation is a program which is a mixture of BASIC statements and directly executing Z80 machine-code instructions. The Z80 can execute branches and subroutine calls, and can perform logic and arithmetic (excluding multiply and divide) on INTEGERs, but not on SINGLE or DOUBLE precision floating-point numbers. Nor can it directly manipulate the internal form of BASIC strings, although it can move strips of bytes from one variable to another quite efficiently. (The difficulty with strings is that their lengths vary dynamically). So ACCEL3 translates many statements to sequences of calls to routines in ROM, or to its own run-time component.

In addition to the actual execution of the program operations, there is the "resolution" of the variable names and line-numbers. Here a compiler comes into its own. The BASIC interpreter resolves each name by a sequential search through its dictionary (table of variables), every time the variable is referenced during execution. In contrast the compiler allocates storage for the variable once during compilation, and then replaces each compiled reference by a direct machine address, rather than a dictionary search. Similarly each reference to a line number in GOTO or GOSUB translates to a simple branch address, whereas the BASIC interpreter has to search the program sequentially from the top to find the target line.

One effect of BASIC's two forms of sequential search is that the running time of a program depends on how large it is. The more variables you have in your program, then the longer the average time taken to find each one, and the more lines in your program, the longer it takes to execute each GOTO or GOSUB. The speed of the compiled code, on the other hand, is independent of program size and number of variables. This means that it is quite impossible ever to make a firm statement about relative performance, since you cannot say how long a statement such as A = B + C will take under the interpreter. It depends on context. Similar arguments apply to program size before and after compilation. Programs may contain REMarks and blanks. BASIC names can be any length. After compilation all these uncertainties disappear - the REMarks and blanks are removed (from translated code) and the variable and line references are all two-byte addresses.

So the table that follows is in one sense very pessimistic. The timings were all taken on the smallest program in which they could be measured, i.e. a simple FOR-loop. There were no blanks or remarks in the source, and the names were all two bytes long. The performance improvement measured for GOTO, for example, is 216 to 1. In a large program this would be even greater. But the catch is that this figure may be irrelevant. Because the directly executing operations are so fast, they scarcely contribute to the execution total at all, and performance becomes dominated by those operations that are not compiled, e.g. READ, by the out-of-line subroutines, e.g. Multiply, or by I/O.

Apart from indicating performance gains the table also summarises those language features that ACCEL3 will optimise, rather then leave to BASIC. SAVE, CSAVE, CLOAD, and RESTORE are also translated, but for expediency rather than as a performance improvement.

southern software

PO Box 39, Eastleigh, Hants, England, SO5 5WQ

SPEED/SPACE Performance Table.

Speed Improvement(Ratio)				Operation	Space Degradation(Bytes)			
INT	SNG	DBL	STR		INT	SNG	DBL	STR
170	28	20	7.3	Assignment (LET)	-4	0	0	0
3.5	3.6	3.6	3.5	Array Reference (1-dim)	13	13	13	13
3.0	3.0	3.0	3.0	Array Reference (2-dim)	12	12	12	12
35	1.0	1.6		AND, OR	4	7	7	
23	2.0	1.6	6.6	Compare (=)	3	10	10	3
57	1.0	1.4	3.6	Add, Concatenate (+)	1	6	6	2
40	1.0	1.3		Subtract (-)	4	6	6	
1.5	1.5	1.1		Multiply (*)	6	6	6	
1.00	1.17	1.02		Divide (/)	6	6	6	
77	70	84	9.3	Constant Reference	0	6	4	4
7.1	1.9			FOR-NEXT	6	23		
111	6.0	4.0		POKE	-1	5	5	
10	4.5	3.6		SET, RESET	-1	5	5	
47	4.6	3.0	8.1	IF THEN ELSE	3	9	9	3
33	4.3	3.5		ON expression GOTO	-2	0	0	
50	6.0	5.1		ON expression GOSUB	0	3	3	
1.2	1.01	1.03	1.2	PRINT simple-variable	-1	-1	-1	-1
61	5.0	3.7		OUT	5	11	11	
				Flow of Control				
	216			GOTO		-7		
	74			GOSUB/RETURN		-10		
				Functions				
inf	inf	inf	inf	VARPTR	-3	-3	-3	-3
5.2	1.9	1.7		POINT	3	9	9	
38	2.3	2.0		INP	5	8	8	
149	2.3	2.0		PEEK	0	3	3	
				String Functions				
			53	ASC				5
			258	LEN				0
			4.8	LEFT$				1
			4.7	RIGHT$				1
			6.4	MID$				2
			25	CHR$				0
			36	CVI				0
			16	MKI$				0
			7.1	CVS				0
			25	MKS$				0
			5.4	CVD				0
			16	MKD$				0

Disclaimers-

1) Absolutely no commitment is implied by these figures. They are subject to all sorts of variability. E.g the time to reference a constant depends on the actual value of the constant.

2) Speed ratios for STRING operations depend on the lengths of the strings, whether the string is a program constant (a literal), whether the receiving string is the same length as the source string, etc. In measurement 4-byte strings were used.

3) Use of "inf" (infinity) in the table means that the ratio could not be measured meaningfully. I.e. the reference to VARPTR(X) in a BASIC program takes longer than the reference to X alone. But in a compiled program, VARPTR(X) is actually faster than the reference to X.

4) Negative numbers in the space table mean that the compiled code occupied less space than the original. These numbers are based on the assumption that one statement per line is used. GOTO5000 occupies 10 bytes in BASIC (5 for the line overhead, 1 for the GOTO keyword, and 4 for the line number). In compiled code this becomes a single 3-byte instruction.

5) When a program is compiled there is a one-time overhead of about 30 bytes. So small programs will appear pessimistic, compared with the table. Also scalar variables become "part" of the program when it is compiled, so use of ?MEM immediately after compilation will give apparently pessimistic results.

6) ACCEL3 is very different from ACCEL and ACCEL2 in its treatment of both lines and expressions. Coupled with the difficulty of taking meaningful measurements, this has resulted in some large differences in the quoted speed ratios. Generally, the large ratios should not be read literally, except to note that they are "large". Significant differences between ACCEL3 and ACCEL2 are that FOR-NEXT and 1-dim array references are slower, while IF THEN ELSE, OUT, INP, SINGLE and DOUBLE assignment and the string conversion functions (CVS etc.) are faster.

PROGRAMMING EXAMPLES.

These examples illustrate specific advantages that can be achieved by compilation. The first program allows you to produce musical notes from a BASIC program via the tape output port. In its uncompiled form the program runs so slowly that the waveform generated sounds like a series of blips, much like a Geiger-counter. Compiled, a top note of two octaves above middle C (1024 cycles) is easily achieved. In fact the program uncompiled runs for 16 mins 34 secs, while compiled it takes only 10 seconds.

The second example is much more worthwhile. Every business application involves some degree of validation of the keyed input. This validation has to meet two conflicting requirements. First, it must diagnose any detectable errors immediately, and request the operator to rekey. Second, although this validation code may be quite complex, it must not be so slow that it causes the loss of any operator keystrokes. Apart from introducing errors, this has the effect of causing the operator to stumble, and lose confidence. So in this second example we are not looking for start-to-end speed-ups as the result of compilation. Rather, we are looking for better human factors.

These sample programs, and possibly other sample test-cases, may or may not be included on the tape or disk you received from Southern Software. On tape they will be standard CLOAD files on the reverse (unlabelled) side of the tape. On disk the initial boot-up menu may give an action code to install them on your system disk. If other samples are included, then their running instructions will appear as comments at head of the program.

southern software

PO Box 39, Eastleigh, Hants, England, SO5 5WQ

A) SINGLE-NOTE MUSIC MAKER.

The tape output is port number 255. You can drive this from a BASIC program by the statement OUT 255,X where X is a value sent to the port. If the least significant bit of X is 0 then the tape signal latch goes low. If it is 1 then it goes high. So by driving it alternately high and low you can generate a square wave. The frequency of this wave will control the pitch you hear, if you put the signal through an audio amplifier. The length of the note is decided by how long you make the loop. The volume cannot be altered.

Notes-

1)The tape output signal is on the larger grey jack.

2)A square-wave makes a nasty "electronic" sound. You can improve the sweetness by putting it through a circuit with a poor response to high frequency, e.g. 2000 c/s maximum.

3)The delay values which control the pitch do not give a uniform table. This is because the inner loop has a non-linear overhead which itself depends on the frequency.

4)On the Model III the timer is not disabled by CMD"T". The timer interrupts produce a crackle, which can be eliminated by calling a USR routine to disable interrupts. This routine is two bytes long, X'F3' (disable) and X'C9' (return).

The Program-

```
10 'SINGLE NOTE MUSIC MAKER, USING TAPE OUTPUT PORT
20 DEFINT A-Z
30 DIM P(100),L(100)
40 'PITCH OF NOTES, FOR 3 OCTAVES, 128 TO 1024 C/S APPROX.
50 'C   C#  D   D#  E   F   F#  G   G#  A   A#  B   C
60 '284,268,250,236,223,208,196,183,172,161,152,144,134
70 '134,126,118,111,104, 97, 90, 85, 78, 73, 68, 63, 59
80 ' 59, 55, 51, 47, 43, 39, 36, 33, 31, 29, 26, 24, 21
90 READ N 'NUMBER OF NOTES TO PLAY
100 DATA 23
110 'TAKE A PAIR OF SPARKLING EYES
120 'PITCH OF NOTE, FOR THIS TUNE
130 DATA 134,104,85,73,63,59,59,63,73,85,85,73,104,118,97,97,104,118,134,134
140 'LENGTH OF NOTE PLAYED, IN QUAVERS. NEGATIVE MEANS REST
150 DATA 2,1,2,1,2,1,4,1,1,2,1,2,1,4,1,1,2,1,1,1,1,4,-2
160 FOR I=1 TO N:READ P(I):NEXT 'PITCHES
170 FOR I=1 TO N:READ L(I):NEXT 'LENGTHS
180 FOR CC=1 TO 2 'PLAY 2 PHRASES
190 FOR C=1 TO N 'PLAY N NOTES
200 LL=0:LM=L(C) 'LENGTH OF NOTE, IN QUAVERS
210 R=1:IF LM<0 THEN R=0:LM=-LM 'REST?
220 I=0:K=P(C):L=K:M=0
230 OUT 255,M 'OUTPUT SIGNAL, ODD=HIGH, EVEN=LOW
240 I=I+10:IF I<L THEN 240 'DELAY, TO PRODUCE PITCH
250 L=L+K:M=R-M 'SWITCH WAVEFORM SIGNAL
260 IF I<10000 THEN 230 'PLAY ONE QUAVER. (CHANGE THIS CONSTANT TO ALTER SPEED OF MUSIC)
270 LL=LL+1:IF LL<LM THEN 220 'ANOTHER QUAVER WITHIN THIS NOTE?
280 NEXT C 'NEXT NOTE
290 NEXT CC 'NEXT PHRASE
300 END
```


B) INPUT VALIDATION.

This program collects a Work Order specification for the hypothetical ACME freight company. The first input field is a customer account number, validated for length, for numerics only, and against a Modulus-11 check. When this program runs under interpretive BASIC, a very fast typist can exit from this field and lose the first character of the next field by keying the second character before the validation completes successfully.

The second field is a two-character US State code. The program checks this input against a table of the 50 possible values. This is a very common form of validation. Other examples are commodity type codes, insurance classifications, tax codings, etc. In this case, although an early code like CAlifornia causes no problems, the table is sufficiently long that a search for e.g. WYoming causes a visible delay, and 2 or 3 keystrokes can easily be lost from the next field. In effect the operator must stop and wait for validation to complete before keying the next field. Compilation solves this important human problem, although of course it makes little difference to the overall throughput.

The Program-

```
10 'DATA VALIDATION EXAMPLE - FREIGHT ROUTING
20 CLEAR 1000
30 DEFSTR A-H,S-Z:DEFINT I-M
40 DIM SC(50) 'STATE CODE
50 GOSUB 330 'INITIALISE
60 CLS:PRINT @10,"ACME FREIGHT COMPANY - WORK ORDER"
70 PRINT @138,"ENTER CUSTOMER NUMBER: ";CHR$(30);
90 INPUT CUSTNO
100 IF LEN(CUSTNO)<>5 THEN PRINT @970,"CUSTOMER NUMBER NOT 5 DIGITS";CHR$(30);:GOTO 70
110 MODSUM=0
120 FOR I=1 TO 5
130 CN=MID$(CUSTNO,I,1)
140 IF CN<"0" OR CN>"9" THEN PRINT @970,"NON-NUMERIC DATA IN CUSTOMER NUMBER";CHR$(30);:GOTO 70
150 MODSUM=MODSUM+ASC(CN)-48 'COMPUTE MODULUS-11 CHECK
160 NEXT
170 IF 11*INT(MODSUM/11)<>MODSUM THEN PRINT @970,"CUSTOMER NUMBER FAILED MODULUS CHECK";CHR$(30);:GOTO 70
180 PRINT @970,CHR$(30); 'CLEAR ERROR MESSAGE, IF ANY
190 PRINT @266,"ENTER DESTINATION (STATE): ";CHR$(30);
210 INPUT STATE
220 FOR I=1 TO 50
230 IF STATE=SC(I) THEN GOTO 260 'FOUND IT
240 NEXT
250 PRINT @970,"INVALID STATE CODE";CHR$(30);:GOTO 190
260 PRINT @970,CHR$(30); 'CLEAR ERROR MESSAGE, IF ANY
270 PRINT @394,"ENTER GOODS CLASSIFICATION: ";
280 INPUT GOODS
290 PRINT @522,"END OF TEST CASE. HIT ENTER TO RERUN. ";
300 INPUT X:GOTO 60
330 'INITIALISATION
340 FOR I=1 TO 50
350 READ SC(I) 'READ IN STATE CODES
360 NEXT
370 RETURN
380   DATA AL,AR,AS,CA,CO,CN,DW,DC,FL,GA,HA,ID,IL,IN,IO,KA,KY,LA,ME,MD,MA,MI,MT,MN,MR,MT,NB,NV,NH,NJ,NM,NY,NC,ND,
OH,OK,OR,PA,RI,SC,SD,TN,TX,UT,VT,VA,WA,WV,WI,WY
```


southern software

PO Box 39, Eastleigh, Hants, England, SO5 5WQ

PERFORMANCE HINTS.

Nothing the compiler can do will speed up I/O devices - disk, tape, printer, or keyboard. But for processing limited by computation the following are good rules!

1) Always use INTEGER data types whenever possible, since these are the only data elements the CPU can manipulate directly. You can qualify variable names with % to make them INTEGERs, but better is to get into the habit of coding e.g. DEFINT I-P at the head of each program.

2) Because FOR-NEXT processing has to be "defensive", in terms of handling badly-behaved loops, it transpires that a programmed loop (e.g. I=I+1:IF I<100 THEN GOTO n) is very much faster. So it may be worth using such a technique on critical inner loops.

3) Avoid continually processing DATA with READ statements. Rather, READ the data values once into an array and process from that. This avoids the very considerable overhead of converting the DATA constants from character to numeric on every use.

4) There is a well-known execution "hiccup" caused by string space "garbage collection", (recovery of free space). ACCEL3 does not affect the actual garbage collection process, but it does attempt to minimise its frequency of occurrence, by avoiding string space allocation if possible. In particular, if string sizes match in assignment, then a spectacular improvement may result.

5) Keystroke polling. The key overrun example earlier showed how it was possible for ACCEL3 to substantially improve the keying characteristics of a program, by reducing the processing time between INPUT statements. However there is one situation where the compiled program may appear to behave worse. Suppose you have a real-time simulation, such as a game like Space Invaders, where your program continually updates the screen and periodically polls the keyboard, using the INKEY$ function. If INKEY$ is null, you loop round and perform the next update. If this update is both long and fully compiled, then it is possible that the player may depress and release a key in between the INKEY$ polls. In this case the keystroke is lost. Interpretive BASIC reduces the chance of this by polling the keyboard at the beginning of every statement (whether or not it asks for input). The cost of this poll is high - in a graphics test case, putting the poll into compiled code actually slowed down the program by a factor of 3. So it is omitted from compiled code, but included in uncompiled statements. (In any case, it's not a perfect solution. Interpreted BASIC may also lose keystrokes). If you have a compiled program that you believe suffers from this problem, then precede some of the compiled statements in the update loop with a colon (:), to force the poll to take place more often.

COMMON PITFALLS.

1) Many programs have loops that are simply there to delay the process, e.g. to make a "ball" moving on the screen go more slowly. Either lengthen these loops when the program is compiled, or use a SINGLE variable FOR-LOOP containing a very slow operation, like DOUBLE divide, which will swamp the compile speed-up.

2) 100 GOTO 100 is a common way of terminating a program to avoid the READY message corrupting the screen. This loop cannot be interrupted by the BREAK key, and will need RESET. Instead use e.g. 100 :GOTO 100

3) If you choose different MEMORY SIZE settings from the example given in the text, or if you position the compiler elsewhere in memory, then be sure the address arithmetic is correct. This is very error-prone. Work it out on paper first, and type it in from the written copy.

4) When you have compiled a program, do not use the editing commands, since they will produce completely unpredictable results. Always reset the machine state with NEW, LOAD, or CLOAD.

5) It is common practice to use DATA statements as a source of variable data. I.e. after running the program once you EDIT new values into the DATA statements for the next run. This isn't possible once the program is compiled. Instead you have to modify the source and recompile.

NOEXPR OPTION.

ACCEL3 supports a compile-time option which minimises the level of optimisation. Code the single line-

REM NOEXPR

in front of the section where optimisation is to be minimised. You can turn optimisation back on with!

REM EXPR

There are a number of reasons for using REM NOEXPR in front of part of a program!

1)If that section contains extended non-Tandy language, then this may be a way of having it execute successfully.

2)The section may make use of ON ERROR GOTO. This may fail in an optimised section because either the error may not be correctly diagnosed, or, if it is, the error line number may not be up-to-date, so RESUME will not work.

3)To minimise code expansion. Since code expansion is not great with ACCEL3, this use is less important than it was with ACCEL2. However, array references in particular give quite a lot of compiled code, and in a non-performance-critical section you may prefer to have these interpreted.

4)If the compiled program fails. This might be due to integer overflow, for example. Preceding the program with REM NOEXPR may either make it easier to trace by running with trace on (TRON), or may eliminate it, in which case it can be identified by limiting optimisation to a section at a time.

REM NOEXPR inhibits compilation of all statements except GOTO, GOSUB, RETURN, FOR, NEXT, ON expr, and IF THEN ELSE (although the statements after THEN and ELSE are uncompiled). Also SAVE, CSAVE, and RESTORE are always compiled. All of the above have to be compiled for the program to retain its integrity. However this does mean that if you have an uncompilable non-Tandy language extension, you cant ever use it in an ON, IF, or FOR expression. Also, if such expressions fail through INTEGER overflow, you wont eliminate the problem with REM NOEXPR.

It is also inevitable that some non-Tandy language extensions will always fail. For instance GOSUB X, where X is a variable containing a dynamically varying line number, would not be understood by ACCEL3, and could not work. The extensions that function correctly rely on the fact that ACCEL3 will pass a string of unrecognised bytes to the interpreter. Since ACCEL3 maintains the run-time variable dictionary exactly as the interpreter expects, such statements or expressions will work, if that's all they depend on. But ACCEL3 does not maintain either the LINE structure of the program, nor the run-time stack, in a compatible form.

If ACCEL3 finds an unrecognised function reference within an expression it will pass just that reference to the interpreter. However it assumes that the data type resulting from that reference is SINGLE. This may be wrong, and in this case make sure the reference is protected with REM NOEXPR.

COMPILE-TIME MESSAGES.

These are messages you may get when compiling a program with ACCEL3.

OM OUT OF MEMORY. Compiler could not complete.
FC ILLEGAL FUNCTION. Disallowed statement, e.g. DELETE.
UL UNDEFINED LINE. Bad line number referenced in GOTO, GOSUB, RUN, etc.
SN SYNTAX ERROR. Compiler cant parse the line.
TM TYPE MISMATCH. Statement implies STRING/numeric data mismatch.
MO MISSING OPERAND. Check the syntax.
ST STRING FORMULA TOO COMPLEX. ACCEL3's restriction is tighter than the interpreter. Break the statement down.

southern software

PO Box 39, Eastleigh, Hants, England, SO5 5WQ

The result of any error found at compile-time will be to leave the program in an indeterminate state. Dont even attempt to LIST it. Note down the error line number, and reload the original.

During compilation 3 numbers are displayed. These are put out chiefly as an aid to see how compilation is progressing. The first is the size (in bytes) of the original BASIC program. The next 2 are the sizes of the program after the two compiler passes over the program.

PASS 1 builds the variable dictionary, and modifies some of the source statements, e.g. DATA statements are moved to the back. It removes REMarks and redundant blanks, so the program size will usually go down.

PASS 2 actually compiles the code, and is the one that expands the text.

RESTRICTIONS.

Experience of users of ACCEL and ACCEL2 has shown that some programs working under BASIC may fail in execution, or even in compilation. These failures were almost always due to the program infringing one or more of the restrictions below, rather than as a result of a compiler bug. So if you encounter a problem, believe that it is as a result of a restriction, and identify the problem by tracing the program, inserting diagnostic PRINT statements, or by breaking the program down into segments.

Certainly users who create a program from scratch, compiling occasionally as they go to check progress rarely suffer any real constraints. ACCEL3 relies on your program obeying certain BASIC rules which are not checked by the interpreter, and indeed are not necessarily documented in the TANDY manual. An example is the "strength reduction" applied by the interpreter. If two INTEGER variables are added, their result may exceed the maximum INTEGER size (32767). In this case the interpreter turns the temporary result into a float (SINGLE) variable. No compiler can afford to produce code to check for this contingency, and indeed it is certain that the original BASIC language designers would have regarded this as an error. Unfortunately the effect here is that compiled and uncompiled programs can give differing results.

1) No redefinition of meaning of names.

The names in your program must mean the same whether the program is read globally as the compiler sees it, or executed dynamically, as the interpreter sees it. The ambiguity applies only to names that take the DEFined data type by default. Names like IX or S$(3) are always consistent. An example of a disallowed name is I=1:DEFINT I:I=1. The interpreter will treat the first I as SINGLE, and the second as INTEGER. The compiler will treat both as INTEGER, i.e. it sees DEFINT as applying to the whole program.

You are unlikely ever to do this sort of thing deliberately, but it can come about, e.g. if CLEAR is used other than at the top of the program. CLEAR resets variables types to default (SINGLE), and may therefore cause a variable to change from INTEGER to SINGLE without your meaning it to. A common error is-

```
10 DEFINT I-N
20 CLEAR 1000
```

2) Current line-number is not maintained.

Lines which start with statements that have been compiled to machine-code do not update the current line-number. Therefore BASIC diagnostic messages may be misleading. TRON will give an incomplete trace.

3) Error behaviour is not necessarily consistent.

Out-of-range arguments to string functions (e.g. MID$ offset and length) are rounded modulo 256. Values out-of-range in ON statements are treated as zero, not errors. Out-of-memory may not be diagnosed at run-time, and may cause a wild branch, or a reboot. Your program may contain errors which BASIC does not diagnose, but which the compiler will reject, for instance bad syntax in an ELSE clause which is never executed. Some error diagnosis will be imprecise, e.g. RETURN WITHOUT GOSUB is diagnosed as NEXT WITHOUT FOR. (Both are symptoms of an empty stack). IF (A=B) 100 is treated by the interpreter as IF (A=B) THEN GOTO 100. ACCEL3 cannot handle this, although it will accept IF expr THEN 100, IF expr GOTO 100, or IF expr PRINT A, etc.

INTEGER OVERFLOW is not necessarily diagnosed. It is rarely caused by addition or subtraction, but may come about through multiplication, which IS diagnosed, but possibly with the wrong line-number. E.g. A = PEEK(I) + 256 * PEEK(I+1) is typically used to calculate a STRING address, and will overflow if the address is in the upper half of memory, i.e. PEEK(I+1) is greater than 127. Correct the problem by forcing one of the arguments to be SINGLE, e.g. 256.0 * PEEK(I+1).

In general, programmed error handling (i.e. the use of ON ERROR) is suspect. This is firstly because the error you are trying to trap may not be caught by the compiled code at all. But secondly, even if the error is trapped, the current line number may be out-of-date, i.e. it is the last uncompiled line. So RESUME may cause a loop. Actually, this problem is not as severe as it sounds, because in practice ON ERROR GOTO is almost always used in conjunction with I/O statements to detect FILE NOT FOUND, DISK FULL, INPUT BEYOND END OF FILE, etc. Since I/O is not compiled, the error trap will work.

4) A first program line of a single colon is disallowed.

5) Compiled programs may not be EDITED.

When the machine holds a compiled program you may not use the commands EDIT, AUTO, DELETE, MERGE, and NAME, and obviously these must not appear in a program you try to compile. (This gives an ILLEGAL FUNCTION diagnostic). In addition GOSUB should not be used as keyboard (i.e. direct) command.

6) Operational differences.

You cant arbitrarily GOTO or RUN any line of the compiled program, only those lines that haven't been optimised. (To force a line N to retain its BASIC line number, simply put RUN N or RESUME N somewhere harmless in the program). BREAK may "take" in an interpreted line, in which case CONTinue may work. Or BREAK may be detected by the ACCEL3 library, in which case control goes to READY. Or BREAK may not take at all, e.g. in a tight GOTO loop. Then you have to reboot.

7) Saving and Loading compiled programs.

Compiled programs contain address references to both variables and to code. These will only work if the program is reloaded (from tape or disk) at exactly the same address. (The run-time library must be at the same address as well). In effect always use the same environment as when the program was saved. SAVE and CSAVE can only be used in direct mode and may not appear in a compiled program. SAVE and CSAVE of a compiled program are only supported for a literal file name, e.g. SAVE "PROG", SAVE expr will not work. Owing to an incompatibility between NEWDOS80 and TRSDOS, a compiled program has to be specially reformatted before SAVE or CSAVE, giving a significant delay on a large program. Also SAVE and CSAVE cause all variables to be cleared, and after LOAD and CLOAD you cannot LIST the program or execute GOTO to a line, until some other operation has been performed.

8) Complexity of STRING expressions.

ACCEL3 is more restrictive than the interpreter on how complex STRING expressions can be. This is diagnosed at compile-time, and if it occurs break the statement down into separate statements.

9) Keyboard Poll.

Compiled code does not poll the keyboard. This may cause cause different operator characteristics, for instance a delay in accepting a keystroke, or failure to pause a scrolled display. You can force the poll by inserting a colon at the front of a line.

southern software

PO Box 39, Eastleigh, Hants, England, SO5 5WQ