GETTING THE MOST OUT OF TINY BASIC

TINY BASIC in the 6800 and 6502 was designed to be a small but powerful language for hobbyists. It allows the user to write and debug quite a variety of programs in a language more "natural" than hexadecimal absolute, and programs written in TINY are reasonably compact. Because the language is small it is not as convenient for some applications as perhaps a larger BASIC might be, but the enterprising programmer will find that there is very little that cannot be done from TINY with only occasional recourse to machine language. This is, in fact, as it should be: the high level language provides the framework for the whole program, and the individual esoteric functions done in machine language fill in the gaps.

For the remainder of this article we will assume one of the standard TINY BASIC programs which follow the memory allocations defined in Appendix D of the User Manual[1]. Specifically, memory locations 0020-0023 contain the boundaries of the user work space, and so on. If your system differs from this norm, you may have to make adjustments to Page 00 address locations referenced here, but everything else should be applicable. Because there are almost as many different starting addresses for the TINY BASIC code as there are versions, we will assume that the variable "S" contains the starting address. In other words, for the "R" version (Mikbug) S=256, the "K" and "S" versions S=512, for "T" (KIM-2 4K) S=8192, etc.

THE USR FUNCTION

Perhaps the least understood feature of TINY BASIC is the machine language subroutine call facility. Not only is it useful for calling your own machine language subroutines, but the two supplied routines let you get at nearly every hardware feature in your computer from a TINY BASIC program, including input and output directly to your peripherals.

First, how do subroutines work? In machine language a subroutine is called with a JSR instruction. This pushes the return address onto the stack and jumps to the subroutine whose address is in the JSR instruction. When the subroutine has finished its operation it executes the RTS instruction, which retrieves that return address from the stack, returning control of the computer to the program that called it. Depending on what function the subroutine is to perform, data may be passed to the subroutine by the calling program in one or more of the CPU registers, or results may be passed back from the subroutine to the main program in the same way. If the subroutine requires more data than will fit in the registers then memory is used, and the registers contain either addresses or more data. In some cases the subroutine has no need to pass data back and forth, so the contents of the registers may be ignored.

If the main program and the subroutine are both written in

TINY BASIC you simply use the GOSUB and RETURN commands to call and return from the subroutine. This is no problem. But suppose the main program is written in TINY and the subroutine is written in machine language? The GOSUB command in TINY is not implemented internally with a JSR instruction, so it cannot be used. This is rather the purpose of the USR function.

The USR function call may be written with up to three arguments. The first of these is always the address of the subroutine to be called. If you refer to USR(12345) it is the same as if you had written a machine language instruction JSR 12345; the computer saves its return address on the stack, and jumps to the subroutine at (decimal) address 12345. For those of you who worry about such things, TINY does not acutally make up a JSR with the specified address in it, but rather simulates the JSR operation with a sequence of instructions designed to have the same effect; the interpreter is clean ("pure code"), and does not modify itself.

So now we can get to the subroutine from a TINY BASIC program. Getting back is easy. The subroutine still simply executes a RTS instruction, and TINY BASIC resumes from where it left off.

If you want to pass data to the subroutine in the CPU registers, TINY allows you to do that also. This is the purpose of the second and third arguments of the USR function call. If you write a second argument in the call, this is evaluated and placed in the index register(s) of the CPU; if you write a third argument it goes into the accumulator(s). If there are results from the subroutine's operation, they may be returned in the accumulator(s) and TINY will use that as the value of the function. Thus writing the TINY BASIC statement

```
LET P = USR (12345,0,13)
```

is approximately equivalent to writing in machine language

```
LDX #0
LDAA #13
JSR 12345
STAA P
```

Now actually there are some discrepancies. The 6800 and the 6502 are 8-bit CPUs but TINY does everything in 16-bit numbers. So in the 6502 the second argument is actually split between the X and the Y registers (the 6800 has a 16-bit index, so there is no problem), and the third argument is split between the A and B registers in the 6800 (the 6502 has no register corresponding to B, so the most significant 8 bits are discarded); the returned value is expected to be 16 bits, so the most significant 8 bits are assumed to be in the B or Y register.

It is important to realize that the three arguments in the USR function are expressions. That is, any valid combination of (decimal) numbers, variables, or function calls joined together by arithmetic operators can be used in any argument. If the variable C=6800 or C=6502 (depending on which CPU you have), the following is a perfectly valid statement in TINY BASIC:

```
13 P=P+0*USR(S+24,USR(S+20,46+C/6800),13)
```

When this line is executed, the inner USR call occurs first, jumping to the "PEEK" subroutine address to look at the contents of either memory location 002E or 002F (depending on whether C<6800 or not); this byte is returned as its value, and is passed immediately as the second argument of the outer call, which stores a carriage return in the memory location addressed by that byte. We are not interested in any result data from the store operation, so the result is multiplied by 0 (giving zero) and added to some variable (in this case P), which leaves that variable unchanged.

What kinds of things can we use the USR function for? As we saw in the example above, we can use it with the two built-in subroutines to "peek" or "poke" at any memory location. In particular this gives us the ability to directly access the input and output devices in the memory space.

DIRECT INPUT & OUTPUT

Suppose you have a PIA at memory address 8006-8007 (the B side of the PIA used by Mikbug, but any PIA will do): We want to read a 4-bit BCD digiswitch in through the low four bits, and output to a 7-segment decoded display through the high four bits. For simplicity we will read in the switch setting, add one, and output it to the display, then repeat. This program will do it:

```
100 REM SET UP PIA DATA DIRECTION
110 B=32768+6
120 X=USR(S+24,B+1,0)+USR(S+24,B,240)+USR(S+24,B+1,4)
130 REM THE FIRST USR SETS THE CONTROL REGISTER
135 REM   TO POINT TO DATA DIRECTION REGISTER
140 REM THE SECOND STORES HEX F0 IN IT
150 REM THE THIRD SETS THE CONTROL REGISTER
155 REM   TO POINT TO PERIPHERAL DATA
160 REM X IS GARBAGE
200 REM INPUT A NUMBER
210 D=USR(S+20,B)
220 REM REMOVE TRASH AND ADD ONE
230 D=D-D/16*16+1
240 REM OUTPUT IT
250 X=USR(S+24,B,D*16)
260 GOTO 200
```

You can also use the USR function for direct access to the character input and output routines, although for input you need to be careful that the characters do not come faster than your TINY BASIC program can take them. The following program inputs characters, converts lower case letters to capitals, then outputs the results:

```
10 REM READ ONE CHARACTER
20 A=USR(S+6)
30 REMOVE PARITY FOR TESTING
40 A=A-A/128*128
50 REM IF L.C., MAKE CAPS
60 IF A>96 IF A<123 THEN A=A-32
70 REM OUTPUT IT
80 A=USR(S+9,A,A)
90 GO TO 10
```

Because of the possible timing limitations of direct character input, it may be preferable to use the buffered line input controlled by the INPUT statement of TINY. Obviously for input of numbers and expressions there is no question, but for arbitrary text input it is also useful, with a little help from the USR function. The only requirement is that the first non-blank characters be a number or (capital) letter. Then the command,

```
300 INPUT X
```

where we do not care about the value in X, will read in a line into the line buffer, affording the operator (that's you) the line editing facilities (backspace and cancel), and put what TINY thinks is the first number of the line into the variable X. Now, remembering that the line buffer is in 0030-0078 (approximately; the ending address varies with the length of the line), we can use the USR function and the PEEK routine (S+20) to examine individual characters at our leisure. To read the next line it is essential to convince the line scanner in TINY that it has reached the end of this line. Location 002E-002F normally contains the current pointer into the input line; if it points to a carriage return the next INPUT statement will read a new line, so all that is needed is to store a carriage return (decimal 13) in the buffer memory location pointed to by this address (see line 13 above).

STRINGS

As we have seen, character input is not such a difficult proposition with a little help from the USR function. (Character output was always easy in the PRINT statement). What about storing and manipulating strings of characters? For small strings, we can use the memory space in 0000-001F and 00C8-00FF, processing them one character at a time with the USR function. Or, if we are careful, we can fill up the beginning of the TINY BASIC program with long REM statements, and use them to hold character strings (this allows them to be initialized when the program is typed in). For example:

```
2 REMTHIS IS A 50-CHARACTER DATA STRING FOR USE IN TINY
3 REM0         1         2         3         4         5
4 REM12345678901234567890123456789012345678901234567890
5 REM...IT TAKES 56 BYTES IN MEMORY:  2 FOR THE LINE #,
6 REM.....3 FOR THE "REM", AND ONE FOR THE TERMINAL CR.
```

If you insert one line in front to GOTO the first program line, then your program will RUN a little faster, and you do not need the letters REM at the beginning of each line (though you still need the line number and the carriage return). If you are careful, you can remove the carriage returns from all but the last text line, and the line numbers from all but the first text line (replace them with data characters), and it will look like a single line to the interpreter. Under no circumstances should you use a carriage return as a data character; if you do, none of the GOTOs, GOSUBs or RETURNs in your program will work.

Gee, you say, if it weren't for that last caveat, I could use the same technique for storing arrays of numbers.

ARRAYS

So the question arises, can the USR function help get around the fact that TINY BASIC does not have arrays? The answer is of course, yes. Obviously the small amount of space left in Page 00 and elsewhere in your system after TINY has made its memory grab is not enough to do anything useful. The possibility that one of the numbers might take on the value 13 means that you cannot use the program space. What else is there? Remember the memory bounds in 0020-0023. If you start TINY with the Warm Start (S+3), you can put any memory limits you wish in here, and TINY will stay out of the rest of memory. Now you have room for array data, subroutines, or anything else. You can let the variable A hold the starting address of an array, and N the number of elements, and a bubble sort would look like this:

```
500 LET I=1
510 LET K=0
520 IF USR(S+20,A+I)>=USR(S+20,A+I-1) GOTO 540
530 K=USR(S+20,A+I)+USR(S+24,A+I,USR(S+20,A+I-1))
535 K=USR(S+24,A+I-1,K)*0+1
540 I=I+1
550 IF I<N GOTO 520
560 IF K<>0 GOTO 500
570 END
```

Of course this not the most efficient sort routine and it will be veerrry slow. But it is probably faster than writing one in machine language, even though the machine language version would execute faster.

THE STACK

A kind of sneaky place to store data is in the GOSUB stack. There are two ways to do this without messing with the Warm Start. But first let us think about the rationale.

When you execute a GOSUB, the line number of the GOSUB is saved on a stack which grows downward from the end of the user space. Each GOSUB makes the stack grow by two bytes, and each RETURN pops off the most recent saved address, to shrink the stack by two bytes. Incidentally, because the line number is saved and not the physical location in memory, you do not need to worry about making changes to your program in case of an error stop within a subroutine. Just don't remove the line that contains an unRETURNed subroutine (unless you are willing to put up with TINY's complaint).

The average program seldom needs to nest subroutines (i.e. calling subroutines from within subroutines) more than five or ten levels deep, and many computer systems are designed with a built-in limitation on the number of subroutines that may be nested. The 8008 CPU was limited to eight levels. The 6502 is limited to about 120. Many BASIC interpreters specify some maximum. I tend to feel that stack space, like most other resources, obeys Parkinson's Law: the requirements will expand to exhaust the available resource. Accordingly, the TINY BASIC subroutine nest capacity is limited only by the amount of available memory. This is an important concept. If my program is small (the program and the stack contend for the same memory space), I can execute hundreds or even thousands of

GOSUBs before the stack fills up. If there are no corresponding RETURN statements, all that memory just sits there doing nothing.

If you read your User's Manual carefully you will recall that memory locations 0026-0027 point to the top of the GOSUB stack. Actually they point to the next byte not yet used. The difference between that address and the end of memory (found in 0022-0023) is exactly the number of bytes in the stack. One greater than the value of the top-of-stack pointer is the address of the first byte in the stack.

If you know how many bytes of data space you need, the first thing your program can do is execute half that many GOSUBs:

```
400 REM B IS THE NUMBER OF BYTES NEEDED
410 LET B=B-2
420 IF B> -2 THEN GOSUB 410
430 REM SIMPLE, ISN'T IT?
```

Be careful that you do not try to call this as a subroutine, because the return address will be buried under several hundred "420"s. If you were to add the line,

```
440 RETURN
```

the entire stack space would be emptied before you got back to the calling GOSUB. Remember also that if you execute an END command the stack is cleared, but an error stop or a Break will not affect it. Before you start this program you should be sure the stack is clear by typing "END"; otherwise a few times through the GOSUB loop and you will run out of memory.

If you are careful to limit it to the main program, you can grab bytes out of the stack as the need arises. An example of this is the TBIL Assembler included in this document. Whether you allocate the memory with one big grab, or a little at a time, you may use the USR peek and poke functions to get at it.

The other way to use the stack for storing data is a little more prodigal of memory, but it runs faster. It also has the advantage of avoiding the USR function, in case that still scares you. It works by effectively encoding the data in the return address line numbers themselves. The data is accessed in true stack format: last in, first out. I used this technique successfully in implementing a recursive program in TINY BASIC.

This method works best with the computed GOTO techniques described later, but the following example will illustrate the principle: Assume that the variable Q may take on the values (-1, 0, +1), and it is desired to stack Q for later use. Where this requirement occurs, use a GOTO (not a GOSUB!) to jump to the following subroutine:

```
3000 REM SAVE Q ON STACK
3010 IF Q<0 THEN GOTO 3100
3020 IF Q>0 THEN GOTO 3150
3050 REM Q=0, SAVE IT.
3060 GOSUB 3200
3070 REM RECOVER Q
3080 LET Q=0
```

```
3090 GOTO 3220
3100 REM Q<0, SAVE IT.
3110 GOSUB 3200
3120 REM RECOVER Q
3130 LET Q=-1
3140 GOTO 3220
3150 REM Q>0, SAVE IT.
3160 GOSUB 3200
3170 REM RECOVER Q
3180 LET Q=1
3190 GOTO 3220
3200 REM EXIT TO (SAVE) CALLER
3210 GOTO ...
3220 REM EXIT TO (RECOVER) CALLER
3230 GOTO ...
```

When the main program wishes to save Q, it jumps to the entry (line 3000), which selects one of three GOSUBs. These all converge on line 3200, which simply jumps back to the calling routine; the information in Q has been saved on the stack. To recover the saved value of Q it is necessary only to execute a RETURN. Depending on which GOSUB was previously selected, execution returns to the next line, which sets Q to the appropriate value, then jumps back to the calling routine (with a GOTO again!). Q may be resaved as many times as you like (and as you have memory for) without recovering the previous values. When you finally do execute a RETURN you get the most recently saved value of Q.

For larger numbers, the GOSUBs may be nested, each saving one bit (or digit) of the number. The following routine saves arbitrary numbers, but in the worst case requires 36 bytes of stack for each number (for numbers less than -16383):

```
1470 REM SAVE A VALUE FROM V
1480 IF V>=0 THEN GOTO 1490
1482 LET V=-1-V
1484 GOSUB 1490
1486 LET V=-1-V
1488 RETURN
1490 IF V>V/2*2 THEN GOTO 1500
1500 GOSUB 1520
1502 LET V=V+V
1504 RETURN
1510 GOSUB 1520
1512 LET V=V+V+1
1514 RETURN
1520 IF V=0 THEN GOTO 1550
1522 LET V=V/2
1524 GOTO 1490
1550 REM GO ON TO USE V FOR OTHER THINGS
```

Note that this subroutine is designed to be placed in the path between the calling routine and some subroutine which re-uses the variable V. When the subroutine returns, it returns through the restoral part of this routine, which eventually returns to the main program with V restored. The subroutine which starts at line 1550 is assumed to be recursive, and it may call on itself through this

save routine, so that any number of instances of V may be saved on the stack. The only requirement is that to return, it first set V to 0, so that the restoration routine will function correctly. Alternatively, we could change line 1550 to jump to the subroutine start with a GOSUB:

```
1550 GOSUB ...
1552 LET V=0
1554 RETURN
```

This requires another two bytes on the stack, but it removes the restriction on the exit from the recursive subroutine.

If you expect to put a hundred or more numbers on the stack in this way you might wish to consider packing them more tightly. If you use ten GOSUBs and divide by 10 instead of 2, the numbers will take one third the stack space. Divide by 41 and any number will fit in three GOSUBs, but the program gets rather long.

BIGGER NUMBERS

Sixteen bits is only good for integers 0-65535 or (-32768)-(+32767). This is fine for games and control applications, but sometimes we would like to handle fractional numbers (like dollars and cents), or very large range numbers as in scientific notation. Let's face it: regular BASIC has spoiled us. Granted. But if you could balance your checkbook in TINY BASIC, your wife might complain less about the hundreds of dollars you spent on the computer. One common way to handle dollars and cents is to treat it as an integer number of cents. That would be OK if your balance never went over $327.67, but that seems a little unreasonable. Instead, break it up into two numbers, one for the dollars, the other for cents. Now your balance can go up to $32,767.99, which is good enough for now (if your balance goes over that you probably don't balance your own checkbook anyway). We will keep the dollars part of the balance in D and the cents in C. The following routine could be used to print your balance:

```
900 REM PRINT DOLLARS & CENTS
910 IF D+C<0 GOTO 960
920 PRINT "BALANCE IS $";D;".";
930 IF C<10 THEN PRINT 0;
940 PRINT C
950 RETURN
960 PRINT "BALANCE IS -$";-D;".";
970 IF -C<10 THEN PRINT 0;
980 PRINT -C
990 RETURN
```

If line number 930 is omitted, then the balance of $62.03 would print as "62.3".

Reading in the dollars and cents is easy if you require that the operator type a comma instead of a period for a decimal point (the European tradition). If that is unacceptable, you can input the dollars part, then increment the input line buffer pointer (memory location 002E-002F) by one to skip over the period, then input the cents part. Be careful that that was not the carriage return you incremented over. The USR function and the peek and poke

subroutines will do all these things nicely.

Adding and subtracting two-part numbers is not very difficult. Assume that the check amount has been input to X (dollars) and Y (cents). This routine will subtract the check amount from the balance:

```
700 REM SUBTRACT DOLLARS AND CENTS FROM BALANCE
710 C=C-Y
720 IF C>=0 THEN GOTO 750
730 C=C+100
740 D=D-1
750 D=D-X
760 IF D>=0 RETURN
770 IF C=0 RETURN
780 D=D+1
790 C=C-100
800 RETURN
```

Adding is a little easier because you cannot go negative (except for overflow), so it is only necessary to check for C>99; if it is, subtract 100 and add 1 to D. If your dollars and cents are in proper form (i.e. no cents values over 99), the sum will never exceed 198, so it is not necessary to retest after adjustment.

Using this same technique you can of course handle numbers with as many digits as you like, putting up to four digits in each piece. A similar technique may be used to do floating point arithmetic. The exponent part is held in one variable, say E, and the fractional part is held in one or more additional variables; in the following example we will use a four-digit fractional part in M, adding to it a number in F and N:

```
1000 REM FLOATING POINT ADD FOR TINY BASIC
1010 IF E-4>F THEN RETURN
1020 IF N=0 RETURN
1030 IF E+4<F THEN LET M=0
1040 IF M=0 THEN LET E=F
1050 IF E=F GOTO 1130
1060 IF E>F GOTO 1100
1070 E=E+1
1080 M=M/10
1090 GOTO 1040
1100 F=F+1
1110 N=N/10
1120 GOTO 1020
1130 M=M+N
1140 IF M=0 THEN E=0
1150 IF M=0 RETURN
1160 IF M>9999 THEN GOTO 1230
1170 IF M>999 RETURN
1180 IF M<-9999 THEN GOTO 1230
1190 IF M<-999 RETURN
1200 M=M*10
1210 E=E-1
1220 GOTO 1170
1230 E=E+1
1240 M=M/10
```

```
1250 RETURN
```

This subroutine is a decimal floating point routine; by changing the divisors and multipliers appropriately, it can be made into a binary, hexadecimal, or even terniary floating point machine. By using the multiple precision techniques described in the checkbook balance example, greater precision can be obtained in the fractional part.

COMPUTED GOTO

One of the more powerful features of TINY BASIC is the computed line address for GOTO and GOSUB statements. A recently published[2] set of games to run in TINY had several large blocks of the program devoted to sequences of IF statements of the form,

```
110 IF I=1 GOTO 1000
120 IF I=2 GOTO 2000
130 IF I=3 GOTO 3000
140 IF I=4 GOTO 4000
150 GOTO 100
```

Now there is nothing wrong with this form of program, but I'm too lazy to type all that, and besides, I could not get the whole program into my memory. Instead of lines 110-140 above, the single line

```
125 IF I>0 IF I<5 GOTO I*1000
```

does exactly the same thing in less memory, and probably faster.

Another part of this program simulated a card game, in which the internal numbers 11-14 were recognized (using the same kind of sequence of IFs) in three different places, and for each different number the name of the corresponding face card was printed. The astonishing thing was that the sequence of IFs, PRINTs, and GOTOs was repeated three different places in the program. Now I'm glad that Carl enjoys using TINY BASIC, and that he likes to type in large programs to fill his voluminous memory; but as I said, I'm lazy, and I would rather type in one set of subroutines:

```
10110 PRINT "JACK"
10115 RETURN
10120 PRINT "QUEEN"
10125 RETURN
10130 PRINT "KING"
10135 RETURN
10140 PRINT "ACE"
10145 RETURN
```

then in each of the three places where this is to be printed, use the simple formula,

```
2510 GOSUB 10000+B*10
```

Along the same line, when memory gets tight you may be able to save a few bytes with a similar technique. Suppose your program has thirteen "GO TO 1234" statements in it; if you have an unused

variable (say, U) you can, in the direct execution mode, assign it the value 1234 (i.e. the line number that all those GOTOs go to), then replace each "GO TO 1234" with a "GOTOU", squeezing out the extra spaces (TINY BASIC ignores them anyway). This will save some thirty or forty bytes, and it will probably run faster also.

EXECUTION SPEED

TINY BASIC is actually quite slow in running programs. That is one of the hazards of a two-level interpreter approach to a language processor. But there are some ways to affect the execution speed. One of these is to use the keyword "LET" in your assignment statements. TINY BASIC will accept either of the following two forms of the assignment statement and do the same thing,

```
R=2+3
LET R=2+3
```

but the second form will execute much faster because it is unnecessary for the interpreter to first ascertain that it is not a REM, RUN, or RETURN statement. In fact, the LET keyword is the first tested, so that it becomes the fastest-executing statement, whereas the other form must be tested against all twelve keywords before it is assumed to be an assignment statement.

Another way to speed up program execution depends on the fact that constant numbers are converted to binary each time they are used, while variables are fetched and used directly with no conversion. If you use the same constant over and over and you do not otherwise use all the variables, assigning that number to one of the spare variables will make the program both shorter and faster. You can even make the assignment in an unnumbered line; the variables keep their values until explicitly changed.

Finally it should be noted that GOTOs and GOSUBs always search the program from the beginning for their respective line numbers. Put the speed-sensitive part of the program near the front, and the infrequently used routines (setup, error messages, and the like) at the end. This way the GOTOs have fewer line numbers to wade through so they will run faster.

DEBUGGING

Very few programs run perfectly the first time. When your program does not seem to run right there are several steps you can take to find the problem.

First of all, try to break it up into its component parts. Use the GOTO command and the END statement to test each part separately if you can. Add extra PRINT statements along the way to print out the variables you are using; sometimes the variables do not have the values in them that we expected. Also the PRINT statements will give you an idea as to the flow of execution. For example, in testing the sort program above (lines 500-570) I inserted the following extra PRINT statements:

```
525 PR "X";
545 PR ".";
555 PR
```

This gave me an idea where in the sort algorithm I was, so I could

follow the exchanges (the "X"s), where each line represented one pass through the main loop. Endless loops become more obvious this way.

If you have not used all the sequential line numbers, you can insert breakpoints in the program in the form of a line number with an illegal statement -- I like to use a single period, because it is easy to type and does not take much space:

```
10 LET A=B+1234
11 .
20 GOSUB 100+A
```

Here when you type RUN, the program will stop with the error message,

```
!184 AT 11
```

Now we can PRINT A, B, etc., to see what might be wrong, or type in GOTO 20 to resume, with no loss to the original program.

As we have seen, there is not much that TINY BASIC cannot do (except maybe go fast). Sure, it is somewhat of a nuisance to write all that extra code to get bigger numbers or strings or arrays, but you can always code up subroutines which can be used in several different programs (like the floating point add above (lines 1000-1250), then save them off on paper tape or cassette.

Remember, your computer (with TINY BASIC in it) is limited only by your imagination.

REFERENCES

[1] TINY BASIC User's Manual. Available from ITTY BITTY COMPUTERS, P.O. Box 23189, San Jose, CA 95153.
[2] Doctor Dobb's Journal, v1 No.7, p.26. Available from PCC, P.O. Box 310, Menlo Park, CA 94025.