```
 QQQQQQ      UU    UU    EEEEEEE      SSSSS     TTTTTTTT
QQ    QQ    UU    UU    EE          SS             TT
QQ QQ QQ    UU    UU    EEEE         SSSSS         TT
QQ  QQQQ    UU    UU    EE               SS       TT
 QQQQQQ      UUUUUU     EEEEEEE      SSSSS         TT
      QQ
```

```
   TTTTTTTT    IIII     NNN   NN      YY    YY
      TT         II      NNNN  NN       YY  YY
      TT         II      NN NN NN        YYYY
      TT         II      NN  NNNN         YY
      TT        IIII     NN   NNN         YY
```

```
BBBBBBB         AA         SSSSS       IIII      CCCCCC
BB    BB       AAAA       SS           II      CC    CC
BBBBBBB       AA  AA       SSSSS       II      CC
BB    BB     AAAAAAAA          SS      II      CC    CC
BBBBBBB     AA      AA      SSSSS      IIII      CCCCCC
```

QUEST ELECTRONICS

TINY BASIC User Manual

Congratulations! You have received the first of what we hope is a long line of low cost software for hobby computers. We are operating on a low margin basis, and hope to make a profit on volume. Please help us to stay in business by respecting the Copyright notices on the software and documentation.

If you are in a hurry to try TINY BASIC, Appendix C will tell you how to get on the air. Then come back and read the rest of this manual --- most of it is useful information.

The TINY BASIC interpreter program has been extensively tested for errors ("bugs"), but it would be foolish to claim of any program that it is guarenteed bug-free. This program does come with a "Limited Warranty" in that any errors discovered will be corrected in the first 90 days. Catastrophic bugs will be corrected by automatically mailing out corrected versions to all direct mail customers and local dealers. Minor bugs will be corrected by request. In any case this warranty is limited to replacement of the Program Tape and/or documentation, and no liability for consequential damages is implied.

If you think you have found a bug, make a listing of the program that demonstrates the bug, together with the run input and output. Indicate on the listing what you think is wrong and what version number you are running

Mail this to:

QUEST ELECTRONICS
TINY BASIC
P.O.Box 4430
Santa Clara CA 95054

TINY BASIC was conceived by the dragons at the People's Computer Company (PCC), a non-profit corporation in Menlo Park, CA, and its implementation defined by Dennis Allison and others in the PCC newspaper and an offshoot newsletter. The implementation of this program follows the philosophy defined there. The reader is referred to PCC v.4 Nos 1-3 for a discussion of the inner workings of this software.

In keeping with the "small is good" philosophy, TINY BASIC employs the two level interpreter aproach (with its consequent speed cost) so that the whole system occupies only 2K of program memory (exclusive of user program; some versions are slightly larger). With 1K of additional RAM small but useful user programs (50 lines or less) may be accomodated. A system with 4K of RAM can contain about 200 lines of user program.

TINY BASIC is designed to be I/O independent, with all input and output funnelled through three jumps placed near the beginning of the program. In the non-standard versions these are preset for the particular operating system I/O, so the discussion to follow is primarily concerned with the standard versions. For this discussion, it is assumed that the interpreter begins at hex address 8406 though the remarks may be applied to other versions with an appropriate offset.

Location 8406 is a LBR to a subroutine to read one ASCII character from the console/terminal. Location 8409 is a LBR to a subroutine to type or display one ASCII character on the console/terminal. In both cases the character is in the D accumulator.

It is assumed that the character input routine will simultaneously display each character as it is input; if this is not the case, the JMP instruction in location 0106 may be converted to a JSR, so that each character input flows through the output subroutine (which in this case must preserve A) before being fed to TINY. Users with terminals using Baudot or some other non-ASCII code should perform the character conversion in the Input and Output subroutines.

If your console is a CRT and/or you have no need to output or display extra pad characters with each Carriage Return and Linefeed, you may intercept these in the output routine to bypass their display. Each input prompt by TINY is followed by an "X-ON" character (ASCII DC1) with the sign bit set to 1 (all other characters except rubout are output with the sign bit set to 0) so these are also readily detected and deleted from the output stream. Appendix C shows how to perform these tests.

A third subroutine gives TINY a means to test for the Break condition in your system. Location 840C is a LD[illegible] to the break subroutine which returns with the break condition recorded in the Carry flag (1 = BREAK, 0 = no BREAK). The Break condition is used to interrupt program execution, or to prematurely terminate a LIST operation. Tiny responds to the Break condition any time in the LIST, or just before examining the next statement in program execution. If a LIST statement included within a program is aborted by the Break condition, the Break condition must be held over to the next statement fetch (or repeated) to stop program execution also.

All input to Tiny is buffered in a 72 character line, terminated by a Carriage Return ("CR"). Excess characters are ignored, as signalled by ringing the console/terminal bell. When the CR is typed in, Tiny will echo it with a Linefeed, then proceed to process the information in the line. If a typing error occurs during the input of either a program line or data for an INPUT statement, the erroneous characters may be deleted by "backspacing" over them and retyping. If the entire line is in error, it may be cancelled (and thus ignored) by typing the "Cancel" key. The Backspace code is located near the beginning of the program (location 84 0F), and is set by default to "left-arrow" or ASCII Underline (shift-O on your Teletype). Similarly the Cancel code is located at memory address 10, and is set by default to the ASCII Cancel code (Control-X). Four characters which may not be used for line edits (Backspace or Cancel) are DC3 (hex 13), LF (0A), NUL (00), and DEL (FF). These codes are trapped by the TINY BASIC input routines before line edits are tested.

When Tiny ends a line (either input or output), it types a CR, pad characters, The pad character used is defined by the sign bit in location 0111, and is set by default to the "Rubout" or Delete code (hex FF; Location 8411 Bit 7 = 1) to minimize synchronization loss for bit-banger I/O routines.

TINY BASIC has a provision for suppressing output (in particular line prompts) when using paper tape for loading a program or inputting data. This is activated by the occurrence of a Linefeed in the input stream (note that the user normally has no cause to type a Linefeed since it is echoed in response to each CR), and disables all output (including program output) until the tape mode is deactivated. This is especially useful in half-duplex I/O systems such as that supported by UT4 since any output would interfere with incoming tape data. The tape mode is turned off by the occurrence of an X-OFF character (ASCII DC3, or Control-S) in the input, by the termination of an executing program due to an error, or after the execution of any statement or command which leaves Tiny in the command mode.

Memory location 0113 is of interest to those 6800 users with extensive operating systems. Normally Tiny reserves 32 bytes of stack space for use by the interpreter and I/O routines (including interrupts). Up to half of these may be used by Tiny in normal operation, leaving not more than 16 bytes on the stack for I/O. If your system allows nested interrupts or uses much more than ten or twelve stack bytes for any purpose, additional space must be allocated on the stack. Location 0113 contains the reserve stack space parameter used by Tiny, and is normally set to 32 (hex 20). If your system requires more reserve, this value should be augmented accordingly before attempting to run the interpreter.

All of these memory locations are summarized in Appendix D. Note that there are no Input or Output instructions or interrupt disables in the interpreter itself; aside from the routines provided for your convenience (which you may connect or disconnect), your system has complete control over the I/O and interrupt structure of the TINY BASIC environment.

TINY BASIC is designed to use all of the memory available to it for user programs. This is done by scanning all the memory from the beginning of the user program space (e.g.0003 for the standard version) for the end of contiguous memory. This then becomes the user program space, and any previous contents may be obliterated. If it is desired to preserve some part of this memory for machine language subroutines or I/O routines, it will be necessary to omit the memory scan initialization. This is facilitated in TINY BASIC by the definition of two starting addresses. Location 8400 (or the beginning of the interpreter) is the "Cold Start" entry point, and makes no assumptions about the contents of memory, except that it is available. Location 8403 is the "Warm Start" entry point, and assumes that the upper and lower bounds of the user program memory have been defined, and that the program space is correctly formatted. The Warm Start does not destroy any TINY BASIC programs in the program space, so it may be used to recover from catastrophic failures. The lower bound is stored in locations 9820-9821 and the upper bound is in locations 9822-9823. When using the Warm Start to preserve memory, you should be sure these locations contain the bounds of the user space.

STATEMENTS

TINY BASIC is a subset of Dartmouth BASIC, with a few extensions to adapt it to the microcomputer environment. Appendix B contains a BNF definition of the language; the discussion here is intended to enable you to use it. When TINY issues a line prompt (a colon on the left margin) you may type in a statement with or without a line number. If the line number is included, the entire line is inserted into the user program space in line number sequence, without further analysis. Any previously existing line with the same line number is deleted or replaced by the new line. If the new line consists of a line number only, it is considered a deletion, and nothing is inserted. Blanks are not significant to TINY, so blanks

imbedded in the line number are ignored; however, after the first non-blank, non-numeric character in the line, all blanks are preserved in memory.

The following are valid lines with line numbers:

```
123 PRINT "HELLO"
456    G O T O 1 2 3
7 8 9 PRINT "THIS IS LINE # 789"
123
32767 PRINT "THIS IS THE LARGEST LINE #"
1PRINT"THIS IS THE SMALLEST LINE #"
10000 TINY BASIC DOES NOT CHECK
10001 FOR EXECUTABLE STATEMENTS ON INSERTION.
```

0 Is not a valid line number.

If the input line does not begin with a line number it is executed directly, and must consist of one of the following statement types:

LET	GOTO	REM
IF...THEN	GOSUB	CLEAR
INPUT	RETURN	LIST
PRINT	END	RUN

These statement types are discussed in more detail in the pages to follow.

Note that all twelve statement types may be used in either the Direct Execution mode (without a line number) or in a program sequence (with a line number). Two of the statements (INPUT and RUN) behave slightly differently in these two operating modes, but otherwise each statement works the same in Direct Execution as within a program. Obviously there is not much point in including such statements as RUN or CLEAR in a program, but they are valid. Similarly, a GOSUB statement executed directly, though valid, is likely to result in an error stop when the corresponding RETURN statement is executed.

EXPRESSIONS

Many of these statement types involve the use of EXPRESSIONS. An Expression is the combination of one or more NUMBERS or VARIABLES, joined by OPERATORS, and possibly grouped by Parentheses. There are four Operators:

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division

These are hierarchical, so that in an expression without parentheses, multiplication and division are performed before addition and subtraction. Similarly, sub-expressions within parentheses are evaluated first. Otherwise evaluation proceeds from left to right. Unary operators (+ and -) are allowed in front of an expression to denote its sign.

A Number is any sequence of decimal digits (0, 1, 2, ... 9), denoting the decimal number so represented. Blanks have no significance and may be imbedded within the number for readability if desired, but commas are not allowed. All numbers are evaluated as 16-bit signed numbers, so numbers with five or more digits are truncated modulo 65536, with values greater than 32767 being considered negative. The following are some valid numbers (note that the last two are equivalent to the first two in TINY):

```
0
100
10 000
1 2 3 4
32767
65536
65 636
```

A Variable is any Capital letter (A, B, ... Z). This variable is assigned a fixed location in memory (two bytes, the address of which is twice the ASCII representation of the variable name). It may assume any value in the range, -32768 to +32767, as assigned to it by a LET or INPUT statement.

The following are some examples of valid expressions:

```
A
123
1+2-3
B-14*C
(A+B)/(C+D)
-128/(-32768+(I*I))
(((((Q)))))
```

All expressions are evaluated as integers modulo 65536. Thus an expression such as

```
N / P * P
```

may not evaluate to the same value as (N), and in fact this may be put to use to determine if a variable is an exact multiple of some number. TINY BASIC also makes no attempt to discover arithmetic overflow conditions, except in the case of an attempt to divide by zero (which results in an error stop). Thus all of the following expressions evaluate to the same value:

```
-4096
15*4096
32768/8
30720+30720
```

TINY BASIC allows two intrinsic functions. These are:

```
RND (range)
USR (address,Xreg,Areg)
```

Either of these functions may be used anywhere an (expression) is appropriate.

FUNCTIONS

RND (range)

This function has as its value, a positive pseudo-random number between zero and range-1, inclusive. If the range argument is zero an error stop results.

USR (address)
USR (address, R8)
USR (address, R8,RA)

This function is actually a machine-language subroutine call to the address in the first argument. If the second argument is included, the index registers contain that value on entry to the subroutine, with the most significant part in R8 If the third argument is included, RA contain that value on entry to the subroutine, with the least significant part RD On exit, the value in the Accumulator and RA.1 becomes the value of the function, with the least significant part in All three arguments are evaluated as normal expressions.

It should be noted that machine language subroutine addresses are 16-bit Binary numbers. TINY BASIC evaluates all expressions to 16-bit binary numbers, so any valid expression may be used to define a subroutine address. However, most addresses are expressed in hexadecimal whereas TINY BASIC only accepts numerical constants in decimal. Thus to jump to a subroutine at hex address 40AF, you must code USR(16559). Hex address FFB5 is similarly 65461 in decimal, though the equivalent (-75) may be easier to use.

For your convenience two subroutines have been included in the TINY BASIC interpreter to access memory. If S contains the address of the beginning of the TINY BASIC interpreter (33792), then location S+20 (hex 8414) is the entry point of a subroutine to read one byte from the memory address in the index register, and location S+24 (hex84 18) is the entry point of a subroutine to store one byte into memory.

A GO TO USR (35827) will return to the SUPER Monitor if you have the video driver version.

```
PRINT print-list
PR print-list
```

This statement prints on the console/terminal the values of the expressions and/or the contents of the strings in the print-list. The print-list has the general form,

```
item,item...    or    item;item...
```

The items may be expressions or alphanumeric strings enclosed in quotation marks (e.g. "STRING"). Expressions are evaluated and printed as signed numbers; strings are printed as they occur in the PRINT statement. When the items are separated by commas the printed values are justified in columns of 8 characters wide; when semicolons are used there is no separation between the printed items. Thus,

```
PRINT 1,2,3
```

prints as

```
1       2       3
```

and

```
PRINT 1;2;3
```

prints as

```
123
```

Commas and semicolons, strings and expressions may be mixed in one PRINT statement at will.

If a PRINT statement ends with a comma or semicolon TINY BASIC will not terminate the output line so that several PRINT statements may print on the same output line, or an output message may be printed on the same line as an input request (see INPUT). When the PRINT statement does not end with a comma or semicolon the output is terminated with a carriage return and linefeed (with their associated pad characters). To aid in preparing data tapes for input to other programs, a colon at the end of a print-list will output an "X-OFF" control character just before the Carriage Return.

Although the PRINT statement generates the output immediately while scanning the statement line, output lines are limited to 125 characters, with excess suppressed.

While the Break key will not interrupt a PRINT statement in progress, the Break condition will take effect at the end of the current PRINT statement.

The following are some examples of valid PRINT statements:

```
PRINT "A=";a,"B+C=";B+C
PR                              (one blank line)
PRI                             (prints the value of I)
PRINT 1,",",Q*P;",",R/42:
```

INPUT input-list

This statement checks to see if the current input line is exhausted. If it is, a question mark is prompted with an X-ON control character, and a new line is read in. Then or otherwise, the input line is scanned for an expression which is evaluated. The value thus derived is stored in the first variable in the input-list. If there are more variables in the input-list, the process is repeated. In an executing program, several values may be input on a single request by separating them with commas. If these values are not used up in the current INPUT statement they are saved for subsequent INPUT statements. The question mark is prompted only when a new line of input values is required. Note that each line of input values must be terminated by a carriage return. Since expressions may be used as input values, any letter in the input line will be interpreted as the value of that variable. Thus if a program sets the value of A to 1, B to 2, and C to 3, and the following statement occurs during execution:

```
INPUT X,Y,Z
```

and the user types in

```
A,C,B
```

the values entered into X, Y, and Z will be 1, 3, and 2, respectively, just as if the numbers had been typed in. Note also that blanks on the input line are ignored by TINY, and the commas are required only for separation in cases of ambiguity. In the example above,

```
ACB
```

could have been typed in with the same results. However an input line typed in as

```
+1  -3  +6   0
```

will be interpreted by TINY as a single value (=58) without commas for separators. There is one anomaly in the expression input capability: if in response to this INPUT, the user types,

```
RND+3
```

TINY will stop on a bad function syntax error (the RND function must be of the form, RND(x)); but if the user types,

```
RN,D+3
```

the values in the variables R, N, and the expression (D+3) will be input. This is because in the expression evaluator the intrinsic function names are recognized before variables, as long as they are correctly spelled.

Due to the way TINY BASIC buffers its input lines, the INPUT statement cannot be directly executed for more than one variable at a time, and if the following statement is typed in without a line number,

```
INPUT A,B,C
```

the value of B will be copied to A, and only one value (for C) will be requested from the console/terminal. Similarly, the statement,

```
INPUT X,1,Y,2,Z,3
```

will execute directly (loading X, Y, and Z with the values 1, 2, and 3), requesting no input, but with a line number in a program this statement will produce an error stop after requesting one value.

If the number of expressions in the input line does not match the number of variables in the INPUT statement, the excess input is

saved for the next INPUT statement, or another prompt is issued for more data. The user should note that misallignment in these circumstances may result in incorrect program execution (the wrong data to the wrong variables). If this is suspected, data entry may be typed in one value at a time to observe its synchronization with PRINT statements in the program.

There is no defined escape from an input request, but if an invalid expression is typed (such as a period or a pair of commas) an invalid expression error stop will occur.

Because Tiny Basic does not allow arrays, about the only way to process large volumes of data is through paper tape files. Each input request prompt consists of a question mark followed by an X-ON (ASCII DC1) control character to turn on an automatic paper tape reader on the Teletype (if it is ready). A paper tape may be prepared in advance with data separated by commas, and an X-OFF (ASCII DC3 or Control-S) control character preceding the CR (a Teletype will generally read at least one more character after the X-OFF). In this way the tape will feed one line at a time, as requested by the succession of INPUT statements. This tape may also be prepared from a previous program output (see the PRINT statement).

```
LET var = expression
var = expression
```

This statement assigns the value of the expresion to the variable (var). The long form of this statement (i.e. with the keyword LET) executes slightly faster than the short form. The following are valid LET statements:

```
LET A = B+C
I = 0
LET Q = RND (RND(33)+5)
```

GOTO expression

The GOTO statement permits changes in the sequence of program execution. Normally programs are executed in the numerical sequence of the program line numbers, but the next statement to be executed after a GOTO has the line number derived by the evaluation of the expression in the GOTO statement. Note that this permits you to compute the line number of the next statement on the basis of program parameters during program execution. An error stop occurs if the evaluation of the expression results in a number for which there is no line. If a GOTO statement is executed directly, it has the same effect as if it were the first line of a program, and the RUN statement were typed in, that is, program execution begins from that line number, even though it may not be the first in the program. Thus a program may be continued where it left off after correcting the cause of an error stop. The following are valid GOTO statements:

```
GOTO 100
GO TO 200+I*10
G O T O X
```

GOSUB expression

The GOSUB statment is like the GOTO statement, except that TINY remembers the line number of the GOSUB statement, so that the next occurrence of a RETURN statement will result in execution proceeding from the statement following the GOSUB. Subroutines called by GOSUB statements may be nested to any depth, limited only by the amount of user program memory remaining. Note that a GOSUB directly executed may result in an error stop at the corresponding RETURN. The following are some examples of valid GOSUB statements:

```
GOSUB 100
GO SUB 200+I*10
```

RETURN

The RETURN statement transfers execution control to the line following the most recent unRETURNed GOSUB. If there is no matching GOSUB an error stop occurs.

```
IF expression rel expression THEN statement
IF expression rel expression statement
```

The IF statement compares two expressions according to one of six relational operators. If the relationship is True, the statement is executed; if False, the associated statement is skipped. The six relational operators are:

=	equality
<	less than
>	greater than
<=	less or equal (not greater)
>=	greater or equal (not less)
<>, ><	not equal (greater or less)

The statement may be any valid TINY BASIC statement (including another IF statement). The following are valid IF statements:

```
IF I>25 THEN PRINT "ERROR"
IF N/P*P=N GOTO 100
IF 1=2 Then this is nonsense
IF RND (100) > 50 THEN IF I <> J INPUT Q,R
```

END

The END statement must be the last executable statement in a program. Failure to include an END statement will result in an error stop after the last line of the program is executed. The END statement may be used to terminate a program at any time, and there may be as many END statements in a program as needed. The END statement also clears out any saved GOSUB line numbers remaining, and may be used for that purpose in the direct execution mode.

REM comments

The REM statement permits comments to be interspersed in the program. Its execution has no effect on program operation, except for the time taken.

CLEAR

The CLEAR statement formats the user program space, deleting any previous programs. If included in a program (i.e. with a line number) the program becomes suicidal when the statement is executed, although no error results. If the Warm Start is used to initialize the interpreter, this must be the first command given.

RUN
RUN,expression-list

The RUN statement is used to begin program execution at the first (lowest) line number. If the RUN statement is directly executed, it may be followed by a comma, followed by values to be input when the program executes an INPUT statement.

If the RUN statement is included in a program with a line number, its execution works like a GO TO first statement of the program.

LIST
LIST expression
LIST expression,expression

The LIST statement causes part or all of the user program to be listed. If no parameters are given, the whole program is listed. A single expresion parameter is evaluated to a line number which, if it exists, is listed. If both expression parameters are given, all of the lines with line numbers between the two values (inclusive) are listed. If the last expression in the LIST statement evaluates to a number for which there is no line, the next line above that number which does exist (if any) is listed as the last line. Zero is not a valid line number, and an error stop will occur if one of the expressions evaluates to zero. A LIST statement may be included as part of the program, which may be used for printing large text strings such as instructions to the operator. A listing may be terminated by the Break key.

If the terminal punch (or cassette recorder) is turned on for a LIST operation, the tape may be saved to reload the program into TINY at a later time.

The following are valid LIST statements:

```
LIST
LIST 75+25          (lists line 100)
LIST 100,200
LIST 500,400        (lists nothing)
```

GETTING THE MOST OUT OF TINY BASIC

TINY BASIC 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 this Manual. Specifically, memory locations 9820-9823 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 Page98 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.

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 SEP4instruction. This pushes the return address onto the stack and jumps to the subroutine whose address follows the SEP4 w/ P=3. When the subroutine has finished its operation it executes the Sep5instruction, 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 SEP4,A(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 SEP with the specified address in it, but rather simulates the SEP 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 SEP5 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 R8 of the CPU; if you write a third argument it goes into the accumulator & RA If there are results from the subroutine's operation, they may be returned in the accumulator & RA.1 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

```
LDI 0              SEP4 ,(12345)        Glo RA
PHI 8              LDI A.0(p)           STR R8
PLO 8              PLO R8
PHI RA             LDI A.1 (p)
LDI #0D            PHI R8
```

Now actually there are some discrepancies. The 1802 is a 8-bit CPU but TINY does everything in 16-bit numbers. So the returned value is expected to be 16 bits, so the most significant 8 bits are assumed to be in RA.1

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 (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+20,41+C/1800)-26624,13)
```

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 982E or 982F
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

You can 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 9830-9878 (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 98 2E-982F 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 in98 00-98□F and 98C8-98FF, 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 9820-9823. 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 this Manual carefully you will recall that memory locations 9826-9827 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 9822-9823) 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 93 2E-982F) 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

[2] Doctor Dobb's Journal, v1 No.7, p.26. Available from PCC, P.O. Box 310, Menlo Park, CA 94025.

A P P E N D I X A

ERROR MESSAGE SUMMARY

0	Break during execution
8	Memory overflow; line not inserted
9	Line number 0 not allowed
13	RUN with no program in memory
18	LET is missng a variable name
20	LET is missing an =
23	Improper syntax in LET
25	LET is not followed by END
34	Improper syntax in GOTO
37	No line to GO TO
39	Misspelled GOTO
40,41	Misspelled GOSUB
46	GOSUB subroutine does not exist
59	PRINT not followed by END
62	Missing close quote in PRINT string
73	Colon in PRINT is not at end of statement
75	PRINT not followed by END
95	IF not followed by END
104	INPUT syntax bad - expects variable name
123	INPUT syntax bad - expects comma
124	INPUT not followed by END
132	RETURN syntax bad
133	RETURN has no matching GOSUB
134	GOSUB not followed by END
139	END syntax bad
154	Can't LIST line number 0
164	LIST syntax error - expects comma
183	REM not followed by END
184	Missing statement type keyword
186	Misspelled statement type keyword
188	Memory overflow: too many GOSUB's ...
211	... or expression to complex
224	Divide by 0
226	Memory overflow
232	Expression too complex ...
233	... using RND ...
234	... in direct evaluation;
253	... simplify the expression
259	RND (0) not allowed
266	Expression too complex ...
267	... for RND
275	USR expects "(" before arguments
284	USR expects ")" after arguments
287	Expression too complex ...
288	... for USR
290	Expression too complex
293	Syntax error in expression - expects value
296	Syntax error - expects ")"
298	Memory overflow (in USR)
303	Expression too complex (in USR)

304 Memory overflow (in function evaluation)
306 Syntax error - expects "(" for function arguments
330 IF syntax error - expects relation operator

Error number 184 may also occur if TINY BASIC is incorrectly interfaced to the keyboard input routines. A memory dump of the input line buffer may disclose this kind of irregularity.

A P P E N D I X B

FORMAL DEFINITION OF TINY BASIC

```
line ::= number statement CR
         statement CR
statement ::= PRINT printlist
              PR printlist
              INPUT varlist
              LET var = expression
              var = expression
              GOTO expression
              GOSUB expression
              RETURN
              IF expression relop expression THEN statement
              IF expression relop expression statement
              REM commentstring
              CLEAR
              RUN
              RUN exprlist
              LIST
              LIST exprlist
printlist ::=
              printitem
              printitem :
              printitem separator printlist
printitem ::= expression
              "characterstring"
varlist ::= var
            var , varlist
exprlist ::= expression
             expression , exprlist
expression ::= unsignedexpr
               + unsignedexpr
               - unsignedexpr
unsignedexpr ::= term
                 term + unsignedexpr
                 term - unsignedexpr
term ::= factor
         factor * term
         factor / term
factor ::= var
           number
           ( expression )
           function
function ::= RND ( expression )
             USR ( exprlist )
number ::= digit
           digit number
separator ::= , ! ;
var ::= A ! B ! ... ! Y ! Z
digit ::= 0 ! 1 ! 2 ! ... ! 9
relop ::= < ! > ! = ! <= ! >= ! <> ! ><
```

A P P E N D I X C

IMPLEMENTING I/O ROUTINES

COSMAC TINY occupies 8400-8BFF
Similarly, the general parameters occupy 9820-98B7, as defined in the manual. However, COSMAC TINY also uses locations 9811-981F to contain copies of interpreter parameters and other run-time data; do not attempt to use these locations while running TINY.

COSMAC TINY contains no I/O instructions (nor references to Q or EF1-4), no interrupt enables or disables, and no references to an operating system. The three jumps (LBR instructions) at 9806, 9809, and 980C provide all necessary I/O, as defined in the manual. If you are using UT3 or UT4, you may insert the following LBR instructions, which jump to the necessary interface routines:

```
                ..  LINKS TO UT3/4
9806 C0076F             LBR UTIN
9809 C00776             LBR UTOUT
980C C00766             LBR UTBRK
```

If you are not using the RCA monitor, you must write your own I/O routines. For this the standard subroutine call and return linkages are used, except that D is preserved through calls and returns by storing it in RF.1. Registers R2-RB and RD contain essential interpreter data, and if the I/O routines make any use of any of them they should be saved and restored. Note however, that R2-R6 are defined in the customary way and may be used to nest subroutine calls if needed. R0,R1,RC, RE and RF are available for use by the I/O routines, as is memory under R2. Both the call and return linkages set X and the I/O data character is passed in the accumulator ("D", not RD).

After connecting TINY to the I/O routines, start the processor at 00 (the Cold Start). Do not attempt to use the Warm Start without entering the Cold Start at least once to set up memory from 11- 23. Any register may be serving as program counter when entering either the Cold Start or the Warm Start.

The USR function works the same way as described in the manual, except that the second argument in the call is loaded into R8, and the third argument is loaded into RA with the least significant byte also in the Accumulator. On return RA.1 and the accumulator contain the function value (RA.0 is ignored). The machine language subroutine must exit by a SEP R5 instruction. USR machine language subroutines may use R0, R1, R8, RA, RC-RF, so long as these do not conflict with I/O routine assignments. TINY BASIC makes no internal use of R0, R1, RC, or RE.

CRT OR TVT

If a TV Typewriter is used for I/O it may be desirable to remove excess control characters from the output stream. All controls except Carriage Return may be removed by the following instructions at the beginning of the output subroutine shown):

```
        SEP 5
OUTPUTXRI #80
        SDI #0a
        BDF OUTPUT -1
        ADI #8A
```

Some TV Typewriters do not scroll up when the cursor reaches the bottom of the screen, but rather wrap the cursor around to the top of the screen, writing over the previously displayed data. With this kind of display it is essential that the I/O routines (or the hardware) clear to the end of the line whenever a CR-LF is output, so that previous data does not interfere with the new. If your I/O routines are fixed in ROM, some sort of preprocessor may be required to recognize output CR's and convert them to the appropriate sequence of control functions. It may also be necessary to trap input CR's (suppressing their echo) since Tiny generally responds with both another CR and a linefeed.

Some users prefer to concatenate all output into one "line" of print, using the terminal comma or semicolon to suppress the line breaks. Since TINY was designed to limit line lengths to less than 128 characters, if this sort of concatenation is attempted it will appear that TINY has quit running. To eliminate the print suppression the most significant two bits of the print control byte (location 981B in most versions) may be cleared to zero periodicly with the USR function or in the output driver routine. The least significant three bits of this same byte are used for the "comma spacing" in the PRINT statement, and should be left unaltered.

CASSETTE I/O

Officially, TINY only speaks to one peripheral--the console. However a certain amount of file storage may be simulated by attaching these peripherals (such as cassette systems) to the character input and output routines. If the same electrical and software interface is used this is very easy. Otherwise the I/O drivers will require special routines to recognize control characters in the input and output data for setting internal switches which select one of several peripherals. The USR function may also be used either to directly call I/O routines or to alter switches in memory.

A P P E N D I X D

LOW MEMORY MAP

LOCATION	SIGNIFICANCE
9800-980F	Not used
9811-981F	Cosmac version temporaries
9820-9821	Lowest address of user program space
9822-9823	Highest address of program space
9824-9825	Program end + stack reserve
9826-9827	Top of GOSUB stack
9828-982F	Interpreter parameters
9830-987F	Input line buffer & Computation stack
9880-9881	Random Number Generator workspace
9882-9883	Variable "A"
9884-9885	Variable "B"
.	...
98B4-98B5	Variable "Z"
98B6-98C7	Interpreter temporaries
98B8	Start of User program (PROTO)
98C8-98FF	Unused by standard version
8400	Cold Start entry point (6800)
8403	Warm Start entry point
8406-8408	LBR (or SEP4) to character input
8409-840B	LBR to character output
840C-840E	LBR to Break test
840F	Backspace code
8410	Line Cancel code
8411	Pad character
8412	Tape Mode Enable flag (hex 80 = enabled)
8413	Spare stack size
8414	Subroutine to read one Byte from RAM to A (address in X)
8418	Subroutine to store A into RAM at address in X
0003	Beginning of User program

A P P E N D I X E

AN EXAMPLE PROGRAM

```
10 REM DISPLAY 64 RANDOM NUMBERS < 100 ON 8 LINES
20 LET I=0
30 PRINT RND (100),
40 LET I=I+1
50 IF I/8*8=I THEN PRINT
60 IF I<64 THEN GOTO 30
70 END


100 REM PRINT HEX MEMORY DUMP
109 REM INITIALIZE
110 A=-10
120 B=-11
130 C=-12
140 D=-13
150 E=-14
160 F=-15
170 X = -1
175 O = 0
180 LET S = 256
190 REMARK: S IS BEGINNING OF TINY (IN DECIMAL)
200 REM GET (HEX) ADDRESSES
210 PRINT "DUMP: L,U";
215 REM INPUT STARTING ADDRESS IN HEX
220 GOSUB 500
230 L=N
235 REM INPUT ENDING ADDRESS IN HEX
240 GOSUB 500
250 U=N
275 REM TYPE OUT ADDRESS
280 GOSUB 450
290 REM GET MEMORY BYTE
300 LET N = USR (S+20,L)
305 REM CONVERT IT TO HEX
310 LET M = N/16
320 LET N = N-M*16
330 PRINT " ";
335 REM PRINT IT
340 GOSUB 400+M+M
350 GOSUB 400+N+N
355 REM  END?
360 IF L=U GO TO 390
365 L=L+1
370 IF L/16*16 = L GOTO 280
375 REM DO 16 BYTES PER LINE
380 GO TO 300
390 PRINT
395 END
399 PRINT ONE HEX DIGIT
400 PRINT 0;
```

```
401 RETURN
402 PRINT 1;
403 RETURN
404 PRINT 2;
405 RETURN
406 PRINT 3;
407 RETURN
408 PRINT 4;
409 RETURN
410 PRINT 5;
411 RETURN
412 PRINT 6;
413 RETURN
414 PRINT 7;
415 RETURN
416 PRINT 8;
417 RETURN
418 PRINT 9;
419 RETURN
420 PRINT "A";
421 RETURN
422 PRINT "B";
423 RETURN
424 PRINT "C";
425 RETURN
426 PRINT "D";
427 RETURN
428 PRINT "E";
429 RETURN
430 PRINT "F";
431 RETURN
440 REM PRINT HEX ADDRESS
450 PRINT
455 REM CONVERT IT TO HEX
460 N = L/4096
470 IF L<0 N=(L-32768)/4096+8
480 GOSUB 400+N+N
483 LET N=(L-N*4096)
486 GOSUB 400+N/256*2
490 GOSUB 400+(N-N/256*256)/16*2
495 GOTO 400+(N-N/16*16)*2
496 GOTO=GOSUB,RETURN
500 REM INPUT HEX NUMBER
501 REM FORMAT IS NNNNX
502 REM WHERE "N" IS ANY HEX DIGIT
505 N=0
509 REM INPUT LETTER OR STRING OF DIGITS
510 INPUT R
520 IF R=X RETURN
525 REM CHECK FOR ERROR
530 IF R>9999 THEN PRINT "BAD HEX ADDRESS
531 REM NOTE ERROR STOP ON LINE 530 (ON PURPOSE!)
535 REM CONVERT INPUT DECIMAL DIGITS TO HEX
540 IF R>999 THEN N=N*16
545 IF R>99 THEN N=N*16
550 IF R>9 THEN N=N*16
```

```
555 IF R>0 THEN R=R+R/1000*1536+R/100*96+R/10*6
559 REM PICK UP NON-DECIMAL DIGIT LETTERS
560 IF R<0 THEN LET R=-R
565 REM ADD NEW DIGIT TO PREVIOUS NUMBER
570 LET N=N*16+R
580 GOTO 510
590 NOTE: DONT NEED END HERE


1000 TO RUN RANDOM NUMBER PROGRAM, TYPE "RUN"
1010 IT WILL TYPE 8 LINES THEN STOP.
1020 TO RUN HEX DUMP PROGRAM, TYPE "GOTO 100"
1030 IT WILL ASK FOR INPUT.  TYPE 2 HEX ADDRESSES
1040 EACH TERMINATED BY THE LETTER X,
1050 AND SEPARATED BY A COMMA
1055 (TYPE ALL ZEROS AS LETTER "OH").
1060 THE PROGRAM WILL DUMP MEMORY BETWEEN
1070 THOSE TWO ADDRESSES, INCLUSIVE.
1080 EXAMPLE:
1090 :GOTO 100
1100 DUMP: L,U? A03EX,A046X
1110 A03E EE FF
1120 A040 00 11 22 33 44 55 66
1130 IF THE RANDOM NUMBER PROGRAM
1140 IS REMOVED, OR IF YOU TYPE IN
1150 :1 GOTO 100
1160 THEN YOU CAN GET THE SAME DUMP BY TYPING
1170 :RUN,A03EX,A046X
1180 .
1190 NOTE THAT THIS PROGRAM DEMONSTRATES NEARLY
1200 EVERY FEATURE AVAILABLE IN TINY BASIC.


REMARK: TO FIND OUT HOW MUCH PROGRAM SPACE
REM... YOU HAVE LEFT, TYPE:
LET I=0
1 LET I=I+2
2 GOSUB 1
RUN
REMARK: AFTER A FEW SECONDS, THIS WILL STOP
REM... WITH AN ERROR; THEN TYPE:
END
PRINT "THERE ARE ";I;" BYTES LEFT"
```

```
0100 C430 B0C0 01ED C007 6FC0 0776 C007 665F;
0110 1882 8020 3022 3020 58D5 0781 0900 0000;
0120 4838 9DBA 48D5 C007 51D3 BFE2 8673 9673;
0130 83A6 93B6 46B3 46A3 9F30 29D3 BFE2 96B3;
0140 86A3 1242 B602 A69F 303B D343 ADF8 00BD;
0150 4DED 304A 0298 02A0 031F 02DD 02F0 02D4;
0160 0581 0349 01ED 054E 0204 06A2 02D3 02D3;
0170 05AA 02D3 02D3 03C5 03D5 0403 0379 0418;
0180 063C 02D3 0529 046C 04CB 04A7 0498 049B;
0190 050E 0560 056D 0681 02B6 0367 0448 044B;
01A0 02D3 02D3 02C9 02C5 034E 0344 0341 02D3;
01B0 F8B3 A3F8 01B3 D3BA F81C AA4A B24A A24A;
01C0 BDF8 00AD 0DBF E212 F0AF FBFF 52F3 EDC6;
01D0 9FF3 FCFF 8F52 3BC6 220A BDF8 23AD 8273;
01E0 9273 2A2A 0A73 8DFB 123A E3F6 C8FF 00F8;
01F0 F2A3 F801 B3D3 B4B5 B7F8 2AA4 F83C A5F8;
0200 4BA7 331A D720 BB4D AB9D 5B1B 5BD7 168B;
0210 F4BF D724 9F73 9B7C 0073 D722 B24D A2D7;
0220 2682 7392 73D4 03CC D71E B94D A9E2 49FF;
0230 3033 4BFD D733 85FE FCB0 A6F8 2D22 2273;
0240 9373 97B6 4652 46A6 F0B6 D5FF 103B 6AA6;
0250 FA1F 325C 5289 F473 997C 0038 7373 86F6;
0260 F6F6 F6FA FEFC 54A6 3042 FC08 FA07 B649;
0270 A633 7A89 7399 73D4 0337 D71E 86F4 A996;
0280 2D74 B930 2DFD 0752 D71A ADE2 F4A6 9DB6;
0290 0D52 065D 0256 302D 86FF 20A6 967F 0038;
02A0 96C2 037F B986 A930 2D1B 0BFF 2032 A9FF;
02B0 10C7 FD09 0BD5 D402 C54D AD9A 5D1D 8A5D;
02C0 30C9 D402 C5D4 02C9 BAD7 1A2D FC01 5DAD;
02D0 2D4D AAD5 D402 AAFB 0D32 2D30 A0D4 02AA;
02E0 FF41 3BA0 FF1A 33A0 1B9F FED4 0359 302D;
02F0 D402 AA3B A09D BAAA D403 544B FA0F AA9D;
0300 BAF8 0AAF ED1D 8AF4 AA9A 2D74 BA2F 8F3A;
0310 059A 5D1D 8A73 D402 AAC3 02FB C002 2D9B;
0320 BA8B AAD4 02AA 1B52 49F3 3223 FB80 321C;
0330 9ABB 8AAB C002 A0D7 2482 F52D 9275 337F;
0340 D549 3059 49BA 4930 55D4 0625 3055 D402;
0350 C5D4 0354 8AD4 0359 9A52 D719 F733 7FF8;
0360 01F5 5DAD 025D D5D4 02C9 AD4D BA4D 3055;
0370 FB2F 3266 FB22 D403 F44B FB0D 3A70 29D7;
0380 18B8 D403 CCF8 21D4 03F4 D71E 89F7 AA99;
0390 2D77 BAD4 0415 9832 A9F8 BDA9 93B9 D403;
03A0 C5D7 28BA 4DAA D404 15F8 07D4 0109 D403;
03B0 D5D7 1A9D 5DD7 26B2 4DA2 C002 2820 4154;
03C0 20A3 D403 F249 FC80 3BC2 30F2 D719 F880;
03D0 739D 7373 C8D7 1BFE 3366 D715 AAF8 0DD4;
03E0 0109 D71A 8AFE 32EF 2A9D C7F8 FF30 DF73;
03F0 F88A FF80 BFD7 1B2D FC81 FC80 3B66 5D9F;
0400 C001 09D7 1BFA 07FD 08AA 8A32 97F8 20D4;
0410 03F4 2A30 0AD4 0354 D71A ADD4 0513 3B25;
0420 F82D D403 F49D 73BA F80A D403 551D D404;
0430 E38A F6F9 3073 1D4D EDF1 2D2D 3A2E 1202;
0440 C202 C2D4 03F4 303E D72E 389B FB00 3A5E;
0450 8B52 F0FF 8033 5ED7 2E8B 739B 5DD5 D72E;
0460 B80D A88B 739B 5D98 BB88 ABD5 D402 C59A;
0470 FB80 738A 73D4 02C9 AFD4 02C5 128A F7AA;
0480 129A FB80 7752 3B92 8AF1 328F 8FF6 388F;
0490 F638 8FF6 C7C4 19D5 D405 0ED4 02C5 ED1D;
04A0 8AF4 739A 745D D5D4 02C5 F810 AF4D B80D;
04B0 A80D FE5D 2D0D 7E5D D405 223B C5ED 1D88;
04C0 F473 9874 5D2F 8F1D 3AB1 D5D4 02C5 9A52;
04D0 8AF1 C203 7F0D F373 D405 132D 2DD4 0513;
04E0 1D9D C89D 73AA BAF8 11AF ED8A F752 2D9A;
04F0 773B F6BA 02AA 1D1D 1DF0 7E73 F07E 738A
```

```
0500 7ED4 0524 2F8F CA04 EA12 02FE 3B21 D71A;
0510 AD30 18ED F0FE 3B21 1D9D F773 9D77 5DFF;
0520 00D5 8AFE AA9A 7EBA D5D7 18C2 03B1 4BFB;
0530 0D3A 2ED4 0698 324B D401 0C33 46D7 1CB9;
0540 4DA9 D717 5DD5 D71E B94D A9C0 037F D720;
0550 BB4D ABD4 0698 324B D71C 8973 995D 3042;
0560 D405 FE32 38D7 288A 739A 5D30 4BD4 058B;
0570 42BA 02AA D726 8273 9273 D406 013A 6530;
0580 88D4 058B 42B9 02A9 C002 2DD7 2212 1282;
0590 FC02 F32D 3A9C 927C 00F3 324B 12D5 D716;
05A0 389D FED7 1A9D 765D 30B2 F830 ABD4 0354;
05B0 9DBB D401 06FA 7F32 B252 FB7F 32B2 FB75;
05C0 329E FB19 32A1 D713 02F3 32D7 2D02 F33A;
05D0 DD2B 8BFF 3033 B2F8 30AB F80D 3802 5BD7;
05E0 198B F73B ECF8 07D4 03F4 0B38 4BFB 0D3A;
05F0 B2D4 03D5 D718 8B5D F830 ABC0 02C5 D402;
0600 C58A 529A F1C2 037F D720 BB4D ABD4 0698;
0610 C68D D5ED 8AF5 529A 2D75 E2F1 3312 4BFB;
0620 0D3A 1E30 0DD4 0628 D402 C54D B84D A84D;
0630 B64D A68D 52D7 1902 5DAD 8AD5 D72C 8B73;
0640 9B5D D405 FED7 2A8B 739B 73D4 05FE 2B2B;
0650 D72A 8BF7 2D9B 7733 7B4B BA4B AA3A 629A;
0660 327B D404 15F8 2DFB 0DD4 03F4 D401 0C33;
0670 7B4B FB0D 3A67 D403 D530 50D7 2CBB 4DAB;
0680 D5D7 2682 7392 5DD7 182D CED7 28AA 4D12;
0690 12E2 738A 73C0 022D D727 4B5D 1D4B 73F1;
06A0 1DD5 D404 5ED4 05FE FCFF 9DAF 33BA 9BBD;
06B0 8BAD 2F2F 2F4D FB0D 3AB4 2B2B D404 5ED7;
06C0 280B FB0D 735D 32D9 9A5D 1D8A 5D9B BA8B;
06D0 AA1F 1F1F 4AFB 0D3A D3D7 2EBA 4DAA D724;
06E0 8AF7 AA2D 9A77 BA1D 8FF4 BF8F FA80 CEF8;
06F0 FF2D 74E2 73B8 9F73 5282 F598 5292 75C3;
0700 037E 8F32 3052 FE3B 1ED7 2EBF 4DAF E2F7;
0710 A89F 7C00 B848 5F1F 1A9A 3A15 3030 9FAF;
0720 98BF D724 B84D A82A EF08 2873 1A9A 3A29;
0730 D724 1242 7302 5DD7 2EBA 4DAA D728 AFF1;
0740 324E 8F5A 1A4D 5A1A 4B5A FB0D 3A47 C003;
0750 B573 529D BA2D 43D5 5D2D 88FA 0FF9 605D;
0760 FA08 CEC4 12DD FC00 356E FF00 3D6C D5D7;
0770 118D 73C0 8140 D712 327E DC17 2D5D C081;
0780 A424 3A91 2710 E159 C52A 5610 112C 8B4C;
0790 45D4 A080 BD30 BCE0 131D 9447 CF88 54CF;
07A0 30BC E010 1116 8053 55C2 30BC E014 1690;
07B0 50D2 8349 4ED4 E571 88BB E11D 8FA2 2158;
07C0 6F83 AC22 5583 BA24 93E0 231D 30BC 2048;
07D0 9149 C630 BC31 3430 BC84 5448 45CE 1C1D;
07E0 380D 9A49 4E50 55D4 A010 E724 3F20 9127;
07F0 E159 81AC 30BC 1311 82AC 4DE0 1D89 5245;
0800 5455 52CE E015 1D85 454E C4E0 2D98 4C49;
0810 53D4 6C24 0000 0000 0A80 1F24 9323 1DE2;
0820 3957 30BC 395E 8552 55CE 380A 8643 4C45;
0830 41D2 2B84 5245 CD1D A080 BD38 1485 AD30;
0840 D317 6481 AB30 D385 AB30 D318 5A85 AD30;
0850 D319 542F 30E2 85AA 30E2 1A5A 85AF 30E2;
0860 1B54 2F97 524E C40A 8080 120A 0929 1A0A;
0870 1A85 1813 0980 120B 3130 6173 0B02 0402;
0880 0305 031B 1A19 0B09 060A 0000 1C17 2F8F;
0890 5553 D280 A830 BC31 2A31 2A80 A92E 2FA2;
08A0 122F C12F 80A8 30BC 80A9 2F83 AC38 BC0B;
08B0 2F80 A852 2F84 BD09 022F 8EBC 84BD 0903;
08C0 2F84 BE09 052F 0901 2F80 BE84 BD09 062F;
08D0 84BC 0905 2F09 042F 0A00 010A 7FFF 6684;
08E0 AC30 BC61 0B38 92
```