# A fast GOTO routine for the INS8073 NSC Tiny Basic Microinterpreter

Bruce J. Edmundson
National Semiconductor Corp.

April 28, 1983

---

In many control applications the speed of the "GOTO" routine in the 8073 is unacceptably slow. This is due to the fact that the GOTO (and GOSUB as well) will search for the "target" or "destination" line with a character-by-character search starting from the beginning of the program, regardless of whether the reference is is a forward or backward direction. Therefore, a GOTO with a destination that is many lines into the program may take a considerable time. If interrupts are used, some interrupts could be missed and data lost since they would not be processed during the execution of the GOTO.

For those cases with a fixed program in which the destination addresses can be calculated in advance, the following assembly language routine will prove useful. In this routine, Tiny Basic will put the address of the destination into the "A" variable, then call the routine to "GOTO" the line at that address.

The routine would be called from Tiny Basic like this:

```
10 LET A=#9000:LINK #A000
```

or, compressed:

```
10LETA=#9000:LINK#A000
```

where the variable A contains the address of the target line, 9000 in this case, and the assembly language GOTO routine is located at address A000 (hex). Note that an assignment statement will be somewhat faster if the word LET is used, at the expense of three more bytes of code.

The assembly language GOTO routine is:

```
A000 3A       POP   EA       ; Remove return address from stack
A001 8200     LD    EA,0,P2  ; Get data from "A" variable
A003 5F       POP   P3       ; Restore P3 (arith expr stack ptr)
A004 5E       POP   P2       ; Restore P2 (text pointer)
A005 46       LD    P2,EA    ; Put destination address into P2
A006 24F701   JMP   01F8h    ; GOTO new line
```

Any method of passing the destination address the the routine is acceptable; two "pokes" with the "@" function could also be used.

# Notes on the
# INS8073 NSC Tiny Basic Microinterpreter

Bruce J. Edmundson
National Semiconductor Corp.

May 3, 1982

---

## 1. Introduction

This paper describes how to implement single-character input and output routines for the NSC Tiny Basic Microinterpreter chip, the INS8073. The internal I/O routines are shown along with the assembly language code needed to reference them. All assembly routines shown here are shown assembled at location 9000 for example purposes; a real system would obviously have each routine at a different location (probably in a ROM). The Tiny Basic code shown is similarly fragmentary.

Tables of some useful internal subroutines and memory locations are also given.

## 2. Sending a single character using strings

Sending a single character from the 8073 to the outside world is possible without any additional code, by making use of the built-in string facilities. For example, suppose that we want to send the Ascii code for "BEL" (hex value 07) to ring the bell on the terminal. One way to do this would be to run this program:

```
10 @#9000=#87
20 PRINT $#9000;
```

Line 10 stores the (hexadecimal) value 87 into (hexadecimal) location 9000. Line 20 then prints a string which starts at location 9000. This method takes advantage of the design of the built-in string processing routines.

The internal string output routine has two methods of detecting the end of the string to be printed: first, it stops sending characters if it encounters a carriage return ("CR", hex value 0D); the CR is treated as a delimiter rather than part of the string and is not sent itself. This is the normal string processing mode and is used when storing strings into memory (with $X = "string", for example).

The second method of string termination is to set bit 7 (the most significant binary bit) in the last character of the string to a "1". Thus the normal code for "BEL", which is 07, becomes 87. Transmission of the string is terminated after this final character is sent. Therefore in the example, only one character is sent. Note that the semi-colon ";" is necessary at the end of the PRINT statement to suppress the carriage return and line feed which would otherwise be sent at the end of any PRINT statement.

## 3. Calling PUTC with LINK

Another method of sending a single character is to use the built-in serial output routine ("PUTC") directly with an assembly language interface called by the LINK instruction. This is necessary to pass the character to be sent to the PUTC routine. The required interface routine is:

```
9000 C200   LD    A,0,P2  ; Get char from "A" variable
9002 17     CALL  7       ; Call PUTC routine
9003 5C     RET           ; Return to 8073
```

This routine would then be called from the 8073 like this:

```
10 A=7
20 LINK #9000
```

The LINK instruction calls the routine at the indicated address with register P2 pointing to the beginning of the variable area for the Tiny Basic program. Variable "A" is at a displacement of 0, variable "B" is at a displacement of 2 (remember, variables are 16 bits), and so on through variable "Z" which is at a displacement of 50. Another method of passing values would be to store them in some fixed location in RAM (somewhere outside of the program area) by using the @ function in Tiny Basic to store the value into the memory or retrieve a result.

## 4. Receiving a single character

Receiving a single character from the serial input routine is very similar to sending a character to the output routine. The interface routine is:

```
9000 202A09  JSR  GECO     ; Call "Get character and echo"
                           ;    routine at 092B
9003 C400    LD   A,=0     ; Clear A (char still in E)
9005 01      XCH  A,E      ; Exchange char to A register
9006 8A00    ST   EA,0,P2  ; Store char into "A" variable
9008 5C      RET           ; Return to 8073
```

This routine would then be called from the 8073 like this:

```
10 LINK #9000 : REM received char is now in A.
```

The GECO routine gets a character from the serial input (waiting as long as necessary until one is received) and returns it as a 7-bit value in both the A and E registers; the routine shown then stores it into the low-order byte of the "A" variable with the high-order byte set to zero.

## 5. Checking serial input status

The built-in I/O routines use a software routine to process the characters and do the proper time delays between bits. The on-chip flags and sense inputs are used to transmit and receive the characters. The serial input comes into the chip on the SENSEA line which is bit 4 of the status register (mask value 10 in hexadecimal). Testing for an incoming character involves watching the serial input line for a start bit (a "0"). This must be done often enough so that a character is not missed or misread by starting the read routine too late. From the Tiny Basic level, the serial input may be tested using the status register, like this:

```
10 IF STAT AND #10 THEN GOTO 10  (Loop back if no start bit)
```

Unfortunately, the 8073 will take about 10 milliseconds to execute that statement loop, which is really too slow even for 110 baud (which has a bit time of 9.09 milliseconds). Therefore, an assembly language routine is required in order to process the character without losing too much time. A simple extension to the GECO routine shown above would be to add a status check so that it would read a character if one was starting to come in, or return a status value to indicate that no character is yet ready. This routine would be as follows:

```
9000 06          LD   A,S     ; Load status register
9001 D410        AND  A,=SA   ; Mask out all but SENSEA (10H)
9003 7C06        BNZ  NC      ; Branch if no char ready
9005 202A09      JSR  GECO    ; Call "Get character and echo"
                              ;    routine at 092B
9008 CA00   RTN: ST   A,0,P2  ; Store char into "A" variable
900A 5C          RET          ; Return to 8073
900B C400   NC:  LD   A,=0    ; Return a "NUL" (no char
900D 74F9        BRA  RTN     ;    ready value)
```

This routine would then be called from the 8073 in the same way as the previous example. The value 0 would be returned if no character is ready or if a null character is received. The routine could also be written to put the status value into a different variable than the received character.

## 6. Useful built-in subroutines

Several routines which are built in to the 8073 Tiny Basic interpreter may be called from user programs. These may be called from assembly language as follows:

CALL 4 — APUSH: push the contents of EA onto the arithmetic expression stack (using P3). This is the stack used for processing expressions within Tiny Basic (and is also used by PRNUM below). This stack is reset at the start of each Tiny Basic statement. Overflowing the stack will cause an ERROR 9 (expression nesting error) and return control to the command interpreter level.

CALL 5 — APULL: pop the contents of the arithmetic expression stack into the EA registers.

CALL 7 — PUTC: Send the character in A to the serial output.

CALL 8 — CRLF: Send a CR/LF character pair to the output routine.

CALL 10 — NEGATE: Negate the 16-bit value in EA (two's complement).

CALL 14 — PRTLN: Print a string of characters starting at the location pointed to by P2. The string is terminated either when bit 7 is set in the last character printed, or when a CR character is found (which is not printed unless it has the bit 7 set).

CALL 15 — ERROR: This routine prints out an error message with an error number, then returns to the interpreter in command mode. The error number is given in the byte following the CALL instruction.

JSR 04FCH — PRNUM: pops a 16-bit number from the arithmetic expression stack and prints it.

JSR 084DH — GETLN: reads a line from the console into the line buffer at location BUFAD. This is the routine used by the editor and INPUT. Terminating the input with control-C returns control to the command interpreter.

JSR 092BH — GECO: receives the next character (with echo) from the input routine into the A and E registers.

JSR 09CBH — DLYX: delay by the time value specified in EA. A millisecond delay would have a value of 63. Other delays are proportional.

## 7. Some on-chip RAM variables

Some of the interpreter variables which are stored in the on-chip RAM may be useful to be tested by user assembly language routines. These are all 16-bit values unless otherwise indicated. Sixteen bit values are stored in memory with the low-order byte followed by the high-order byte.

| Location | Name | Description |
|---|---|---|
| FFC3 | CURRNT | Current line number |
| FFC6 | EXTRAM | Starting address of the external RAM. |
| FFD0 | BUFAD | Address of the line buffer in the external RAM (used by the editor and INPUT). |
| FFD2 | STACK | Starting address of the stack area. |
| FFD4 | TXTBGN | Starting address of the current program area. |
| FFD6 | TXTUNF | Address of the first byte after the end-of-program flag; this value is used by the editor to control entry and deletion of program lines. This value must be properly set for the editor to work (it is not changed by NEW with an address specified). |
| FFD8 | TXTEND | Address of the first byte after the end of the RAM where the program can be stored. Also used by the editor. |
| FFE8 | ONE | A constant with a value of 1. |
| FFEA | ZERO | A constant with a value of 0. |
| FFEC | DLYTIM | The delay time constant for the current baud rate. |