BUILD YOUR OWN BASIC

by Dennis Allison & Others
(reprinted from *People's Computer Company* Vol. 3, No.4)

A DO IT YOURSELF KIT FOR BASIC??

Yes, available from PCC with this newspaper and a lot of your time. This is the beginning of a series of articles in which we will work our way through the design and implementation of a reasonable BASIC system for your brand X computer. We'll be working on computers based on the INTEL 8008 and 8080 microprocessors. But it doesn't make much difference – if your machine is the ZORT 9901 or ACME X you can still build a BASIC for it. But remember, it's a hard job and will take lots of time particularly if you haven't done it before. A good BASIC system could easily take one man six months!

We'd like everyone interested to participate in the design. While we could do it all ourselves, (we have done it before) your ideas may be better than ours. Maybe we can save you, or you can save us, a lot of work or problems. Write us and we'll publish your letter and comments.

WHICH BASIC?

There is not any one standard BASIC (yet). The question is which BASIC should we choose to implement. A smaller (fewer statements, fewer features) BASIC is easier to implement and (more important) takes less space in the computer. Memory is still expensive so the smaller the better. Yet maybe we can't give up some goodies like string variables, dynamic array allocation, and so on.

There is a standard version of BASIC which is to be the minimal language which can be called BASIC. It's a pretty big language with lots of goodies. Maybe too big. Is there any advantage to being compatible with, say, the EDU BASICS? We don't have to make any decision yet; but the time will come . . .

DATA STRUCTURES

Data structures are places to put things so you can find them or use them later. BASIC has at least three important ones: a symbol table which looks up a program name, A or Z9 or A$, with its value. If we had a big computer where space was not a huge problem, we could simply preallocate all storage since BASIC provides for only 312 different names excluding arrays. When memory is so costly this doesn't make much sense. Somewhere, also, we've got to store the names which BASIC is going to need to know, names like LET and GO TO and IF. This table gets pretty big when there are lots of statements.

Lastly, we need some information about what is a legal BASIC statement and which error to report when it isn't. These tables are called parsing tables since they control the decomposition of the program into its component parts.

STRATEGY

Divide and Conquer is the programmers maxim. BASIC will consist of a lot of smaller pieces which communicate with each other. These pieces themselves consist of smaller pieces which themselves consist of smaller pieces, and so forth down to the actual code. A large problem is made manageable by cutting it into pieces.

What are the pieces, the building blocks of BASIC? We see a bunch of them:

* a supervisor which determines what is to be done next. It receives control when BASIC is loaded.
* a program and line editor. This program collects lines as they are entered from the keyboard and puts them into a part of computer memory for later use.
* a line executor routine which executes a single BASIC statement, whatever that is.
* a line sequence which determines which line is to be executed next.
* a floating point package to provide floating point on a machine without the hardware.
* terminal I/O handler to input and output information from the Teletype and provide simple editing (backspace and line deletion).
* a function package to provide all the BASIC functions (RND, INT, TAB, etc.)
* an error handling routine (part of the supervisor).
* a memory management program which provides dynamic allocation data objects.

These are the major ones. As we get futher into the system we'll begin to see others and we'll begin to be able to more fully define the function of each of these modules.

COMPILER OR INTERPRETER?

We favor using an interpreter. An interpreter is a program which will execute the BASIC program from its textual representation. The program you write is the one which gets executed. A compiler converts the BASIC program into the machine code for the machine it is to run on. Compiled code is a lot faster, but requires more space and some kind of mass storage device (tape or disk). Interpretative BASIC is the most common on small machines.

DIRECT MODE?

Some kind of "desk calculator" mode of operation would be nice. At least, we would like to be able to look at and set different variables in a program and restart execution at any given point. This feature makes it easier to find and gently terminate the existence of "bugs."

HOW MUCH MEMORY? AND . . . WHAT KIND?

Can we make some guesses about how big the BASIC system will be? Only if you don't hold us to it. Suppose we want to be able to run a 50 line BASIC program. We need about 800 bytes to store the program, another 60 or so bytes for storing program values (all numeric) without leaving any space for the interpreter and its special data. Past experience has shown that something like 6 to 8 Kbytes are needed for a minimum BASIC interpreter and that at least 12K bytes are necessary for a comfortable system. That's a lot of memory, but not too much more than you need to run the assembler. A lot of BASIC could be put into ROM (Read Only Memory) once developed and checked out. ROM is a lot cheaper than RAM (Read and Write) memory, but you can't change it. It's lots better to make sure everything works first.

But . . . if we can agree on some chunks of code and get it properly checked out, some enterprising person out there might make a few thousand ROMs and save us all some $$$. Let's see now . . . how about ROMs for floating point arithmetic, integer arithmetic, Teletype I/O . . .

TINY BASIC

Pretend you are 7 years old and don't care much about floating point arithmetic (what's that?), logarithms, sines, matrix inversion, nuclear reactor calculations and stuff like that.

And . . . your home computer is kinda small, not too much memory. Maybe its a MARK-8 or an ALTAIR 8800 with less than 4K bytes and a TV typewriter for input and output.

You would like to use it for homework, math recreations and games like NUMBER, STARS, TRAP, HURKLE, SNARK, BAGELS, . . .

Consider then, TINY BASIC

- Integer arithmetic only – 8 bits? 16 bits?
- 26 variables: A, B, C, D, . . ., Z
- The RND function – of course!
- Seven BASIC statement types
  - INPUT
  - PRINT
  - LET
  - GO TO
  - IF
  - GOSUB
  - RETURN
- Strings? OK in PRINT statements, not OK otherwise.

BUILD YOUR OWN BASIC--REVIVED

(reprinted from *People's Computer Company* Vol. 4, No. 1)

WHAT IS TINY BASIC???

TINY BASIC is a very simplified form of BASIC which can be implemented easily on a microcomputer. Some of its features are:

Integer arithmetic 16 bits only

26 variables (A, B, . . ., Z)

Seven BASIC statements

INPUT	PRINT	LET	GOTO
IF	GOSUB	RETURN	

Strings only in PRINT statements

Only 256 line programs (if you've got that much memory)

Only a few functions including RND

It's not really BASIC but it looks and acts a lot like it. I'll be good to play with on your ALTAIR or whatever; better, you can change it to match your requirements and needs.

TINY BASIC LIVES!!!

We are working on a version of TINY BASIC to run on the INTEL 8080. It will be an interpretive system designed to be as conservative of memory as possible. The interpreter will be programmed in assembly language, but we'll try to provide adequate descriptions of our intent to allow the same system to be programmed for most any other machine. The next issue of PCC will devote a number of pages to this project.

✳ In the meantime, read one of these.

Compiler Construction For Digital Computers, David Gries, Wiley, 1971 493 pages, $14.95

Theory & Application of a Bottom-Up Syntax Directed Translator Harvey Abramson, Academic Press, 1973, 160 pages, $11.00

Compiling Techniques, F.R.A. Hopgood, American Elsevier, 126 pages $6.50

A BASIC Language Interpreter for the Intel 8008 Microprocessor A.C. Weaver, M.H. Tindall, R.L. Danielson. University of Illinois Computer Science Dept, Urbana IL 61801. June 1974. Report No. UIUCDCS-R-74-658. Distributed by National Technical Information Service, U.S. Commerce Dept, Springfield VA 22151. $4.25.

A BASIC language interpreter has been designed for use in a microprocessor environment. This report discusses the development of 1) an elaborate text editor and 2) a table-driven interpreter. The entire system, including text editor, interpreter, user text buffer, and full floating point arithmetic routines fits in 16K 8-bit words.

The TINY BASIC proposal for small home computers was of great interest to me. The lack of floating point arithmetic however, tends to limit its usefulness for my objectives.

As a matter of a suggestion, consideration should be given to the optional inclusion of floating point arithmetic, logarithm and trigonmetric calculation capability via a *scientific calculator chip interface.*†

The inclusion of such an option would tend to extend the interpreter to users who desire these complex calculation capabilities. A number of calculator chip proposals have been made, with the Suding unit being of the most interest.

Thank you for the note of 13 June, regarding my letter on the Tiny BASIC article (PCC Vol. 3 No. 4). It was with regret that I learned that the series was not continued in the next volume. Even though few responded to the article published, conceptually the knowledge and principles which would be disseminated regarding a limited lexicon, high level programming language are of importance to the *independent* avocational microcomputer community.

At this time, PCC may not have a wide distribution in the avocation microcomputer community. This could be possibly the cause for the low number of respondies. Never the less, this should not detract from the dissemination and importance of concepts and principles which are of significance.

The thrust of my letter of 15 April, 1975, was to suggest a mechanism for the inclusion of F.P. in a limited lexicon and memory consumptive BASIC. I hope that the implication that F.P. must be included was not read into my letter.

It is my interest that information, concepts and the principles of compiler/interpreter construction as it related to microcomputers be available to the limited budget avocational user. The MITS BASIC, which you brought up, appears from my viewpoint to be a *licensed*, blackbox program which is not currently available to: (a) 8008 users, (b) IMP-16 users, (c) independent 8080 users (except at a very large expense) or (d) MC6800 users who will shortly be on line.

Presently it appears that microcomputer compiler interpretor function langauges will be coming available from a number of sources (MITS, NITS, Processor Technology and etc.). However, few will probably deal in the conceptualizations which are the basis of the interpreter. Information which will fill the void in the interpreter construction knowledge held by the avocation builder, should be made available.

I strongly urge that the series started with Vol. 3 No. 4 article be continued. Possibly the hardware, peripheral, machine programming difficulties incurred by the microcomputer builder, is prohibiting a major contribution at this time. However, I would expect that by Autumn a number of builders should have their construction and peripheral difficulties far enough along to start thinking about higher level languages. The previous objective for the article series sounds reasonable. It was not my purpose in submitting the letter to detract from the objective of a very limited lexicon BASIC, ie., to be attractive and usable by the young and beginner due to its simplicity.

If wives, children, neighbors or anyone who is not machine language or programming oriented is expected to use a home-base unit created under a restrained budget a high level language will be a necessity. It is with this foresight that I encourage the continuance of the "Build Your Own BASIC" series.

This issue aside, I would like to encourage the PCC to continue the quite creditable activities which have been its order of business with regard to avocational computing.

Michael Christoffer
4139 12th NE No. 400
Seattle, Wash. 98105

† Please see Dr. Robert Suding's article on p. 18

DESIGN NOTES FOR TINY BASIC

by Dennis Allison, happy Lady, & friends
(reprinted from *People's Computer Company* Vol. 4, No. 2)

SOME MOTIVATIONS

A lot of people have just gotten into having their own computer. Often they don't know too much about software and particularly systems software, but would like to be able to program in something other than machine language. The TINY BASIC project is aimed at you if you are one of these people. Our goals are very limited--- to provide a minimal BASIC-like language for writing simple programs. Later we may make it more complicated, but now the name of the game is **keep it simple.** That translates to a limited language (no floating point, no sines and cosines, no arrays, etc.) and even this is a pretty difficult undertaking.

Originally we had planned to limit ourselves to the 8080, but with a variety of new machines appearing at very low prices, we have decided to try to make a portable TINY BASIC system even at the cost of some efficiency. Most of the language processor will be written in a pseudo language which is good for writing interpreters like TINY BASIC. This pseudo language (which interprets TINY BASIC) will then itself be implemented interpretively. To implement TINY BASIC on a new machine, one simply writes a simple interpreter for this pseudo language and not a whole interpreter for TINY BASIC.

We'd like this to be a participatory design project. This sequence of design notes follows the project which we are doing here at PCC. There may well be errors in content and concept. If you're making a BASIC along with us, we'd appreciate your help and your corrections.

Incidentally, were we building a production interpreter or compiler, we would probably structure the whole system quite differently. We chose this scheme because it is easy for people to change without access to specialized tools like parser generator programs.

THE TINY BASIC LANGUAGE

There isn't much to it. TINY BASIC looks like BASIC but all variables are integers There are no functions yet (we plan to add RND, TAB, and some others later). Statement numbers must be between 1 and 255 so we can store them in a single byte. LIST only works on the whole program. There is no FOR-NEXT statement. We've tried to simplify the language to the point where it will fit into a very small memory so impecunious tyros can use the system.

The boxes shown define the language. The guide gives a quick reference to what we will include. The formal grammar defines exactly what is a legal TINY BASIC statement. The grammar is important because our interpreter design will be based upon it.

IT'S ALL DONE WITH MIRRORS------ OR HOW TINY BASIC WORKS

All the variables in TINY BASIC: the control information as to which statement is presently being executed and how the next statement is to be found, the return addresses of active GOSUBS-----all this information constitutes the state of the TINY BASIC interpreter.

There are several procedures which act upon this state. One procedure knows how to execute any TINY BASIC statement. Given the starting point in memory of a TINY BASIC statement, it will execute it changing the state of the machine as required. For example,

100 LET S = A+6 (cr)

would change the value of S to the sum of the contents of the variable A and the interger 6, and sets the next line counter to whatever line follows 100, if the line exists.

A second procedure really controls the interpretation process by telling the line interpreter what to do. When TINY BASIC is loaded, this control routine performs some initialization, and then attempts to read a line of information from the console. The characters typed in are saved in a buffer, LBUF. It first checks to see if there is a leading line number. If there is, it incorporates the line into the program by first deleting the line with the same line number (if it is present) then inserting the new line if it is of nonzero length. If there is no line number present, it attempts to execute the line directly. With this strategy, all possible commands, even LIST and CLEAR and RUN are possible inside programs. 'Suicidal' programs are also certainly possible.

TINY BASIC GRAMMAR

The things in **bold face** stand for themselves. The names in lower case represent classes of things. '::=' is read 'is defined as'. The asterisk denotes zero or more occurances of the object to its immediate left. Parenthesis group objects. ε is the empty set. | denotes the alternative (the exclusive-or).

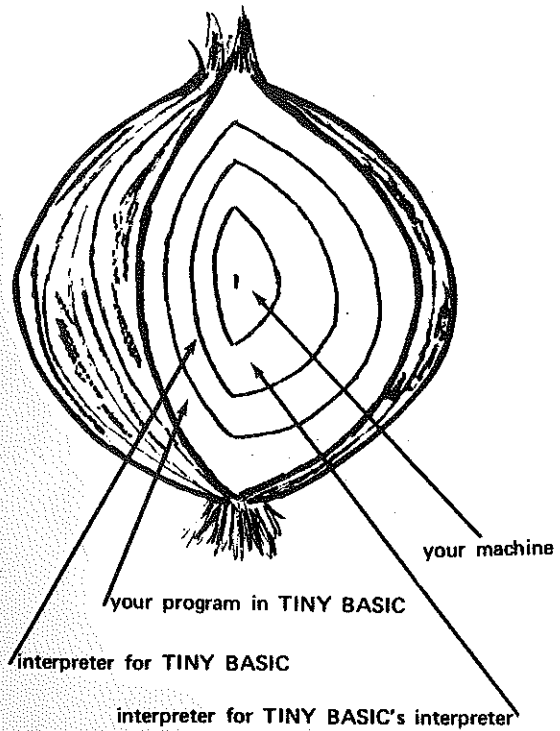
```
line::= number statement (cr) | statement (cr)
statement::= PRINT expr-list
             IF expression relop expression THEN statement
             GOTO expression
             INPUT var-list
             LET var = expression
             GOSUB expression
             RETURN
             CLEAR
             LIST
             RUN
             END
expr-list::= (string | expression) ( , (string | expression) *)
var-list::= var (, var)*
expression::= (+ | - | ε) term ( (+ | -) term)*
term::= factor ( (* | /) factor)*
factor::= var | number | (expression)
var::= A | B |C ... |Y | Z
number::= digit digit*
digit::= 0| 1 | 2 |... | 8 | 9
relop::=< (>| = | ε) | >(< | = | ε) | =
```

A **BREAK** from the console will interrupt execution of the program.

IMPLEMENTATION STRATGIES AND ONIONS

When you write a program in TINY BASIC there is an **abstract machine** which is necessary to execute it. If you had a compiler it would make in the machine language of your computer a program which emulates that abstract machine for your program. An **interpreter** implements the abstract machine for the entire language and rather than translating the program once to machine code it translates it dynamically as needed. Interpreters are programs and as such have their's as abstract machines. One can find a better instruction set than that of any general purpose computer for writing a particular interpreter. Then one can write an interpreter to interpret the instructions of the interpreter which is interpreting the TINY BASIC program. And if your machine is microprogrammed (like PACE), the machine which is interpreting the interpreter interpreting the interpreter interpreting BASIC is in fact interpreted.

This multilayered, onion-like approach gains two things: the interpreter for the interpreter is smaller and simpler to write than an interpreter for all of TINY BASIC, so the resultant system is fairly portable. Secondly, since the major part of the TINY BASIC is programmed in a highly memory efficient, tailored instruction set, the interpreted TINY BASIC will be smaller than direct coding would allow. The cost is in execution speed, but there is not such a thing as a free lunch.



LINE STORAGE

The TINY BASIC program is stored, except for line numbers, just as it is entered from the console. In some BASIC interpreters, the program is translated into an intermediate form which speeds execution and saves space. In the TINY BASIC environment, the code necessary to provide the transformation would easily exceed the space saved.

When a line is read in from the console device, it is saved in a 72-byte array called **LBUF** (Line BUFfer). At the same time, a pointer, **CP**, is maintained to indicate the next available space in LBUF. Indexing is, of course, from zero.

Delete the leading blanks. If the string matches the BASIC line, advance the cursor over the matched string and execute the next IL instruction. If the match fails, continue at the IL instruction labeled lbl.

QUICK REFERENCE GUIDE FOR TINY BASIC

LINE FORMAT AND EDITING

- Lines without numbers executed immediately
- Lines with numbers appended to program
- Line numbers must be 1 to 255
- Line number alone (empty line) deletes line
- Blanks are not significant, but key words must contain no unneeded blanks
- '←' deletes last character
- X^C deletes the entire line

EXECUTION CONTROL

CLEAR delete all lines and data
RUN run program
LIST list program

EXPRESSIONS

Operators

Arithmetic		Relational	
+	−	>	>=
*	/	<	<=
		=	<>,><

Variables
A.....Z (26 only)

All arithmetic is modulo 2^{15}
(±32762)

INPUT / OUTPUT

```
PRINT X,Y,Z
PRINT 'A STRING'
PRINT 'THE ANSWER IS'
INPUT X
INPUT X,Y,Z
```

ASSIGNMENT STATEMENTS

```
LET X=3
LET X= -3+5*Y
```

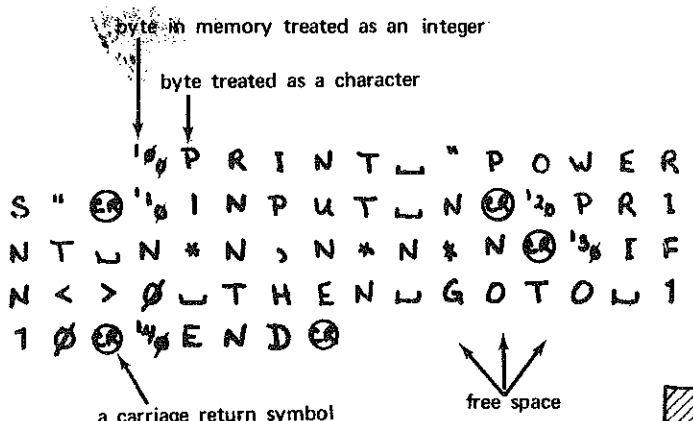
CONTROL STATEMENTS

```
GOTO X+10
GOTO 35
GOSUB X+35
GOSUB 50
RETURN
IF X >Y THEN GOTO 30
```

The TINY BASIC program is stored as an array called **PGM** in order of increasing line numbers. A pointer, **PGP**, indicates the first free place in the array. PGP=0 indicates an empty program; PGP must be less than the dimension of the array PGM. The PGM array must be reorganized when new lines are added, lines replaced, or lines are deleted.

Insertion and deletion are carried on simultaneously. When a new line is to be entered, the PGM array searches for a line with a line number greater than or equal to that of the new line. Notice that lines begin at PGM (0) and at PGM (j+1) for every j such that PGM (j)=[carriage return]. If the line numbers are equal, then the length of the existing line is computed. A space equal to the length of the new line is created by moving all lines with line numbers greater than that of the line being inserted up or down as appropriate. The empty line is handled as a special case in that no insertion is made.

TINY BASIC AS STORED IN MEMORY



ERRORS AND ERROR RECOVERY

There are two places that errors can occur. If they occur in the TINY BASIC system, they must be captured and action taken to preserve the system. If the error occurs in the TINY BASIC program entered by the user, the system should report the error and allow the user to fix his problem. An error in TINY BASIC can result from a badly formed statement, an illegal action (attempt to divide by zero, for example), or the exhaustion of some resource such as memory space. In any case, the desired response is some kind of error message. We plan to provide a message of the form:

! mmm AT nnn

where mmm is the error number and nnn is the line number at which it occurs. For direct statements, the form will be:

! mmm

since there is no line number.

Some error indications we know we will need are:

1 Syntax error	5 RETURN without GOSUB
2 Missing line	6 Expression too complex
3 Line number too large	7 Too many lines
4 Too many GOSUBs	8 Division by zero

THE BASIC LINE EXECUTOR

The execution routine is written in the interpretive language, **IL**. It consists of a sequence of instructions which may call subroutines written in IL, or invoke special instructions which are really subroutines written in machine language.

Two different things are going on at the same time. The routines must determine if the TINY BASIC line is a legal one and determine its form according to the grammar; secondly, it must call appropriate action routines to execute the line. Consider the TINY BASIC statement:

GOTO 100

At the start of the line, the interpreter looks for BASIC key words (LET, GO, IF, RETURN, etc.) In this case, it finds GO, and then finds TO. By this time it knows that it has found a GOTO statement. It then calls the routine EXPR to obtain the destination line number of the GOTO. The expression routine calls a whole bunch of other routines, eventually leaving the number 100 (the value of the expression) in a special place, the top of the arithmetic expression stack. Since everything is legal, the XFER operator is invoked to arrange for the execution of line 100 (if it exists) as the next line to be executed.

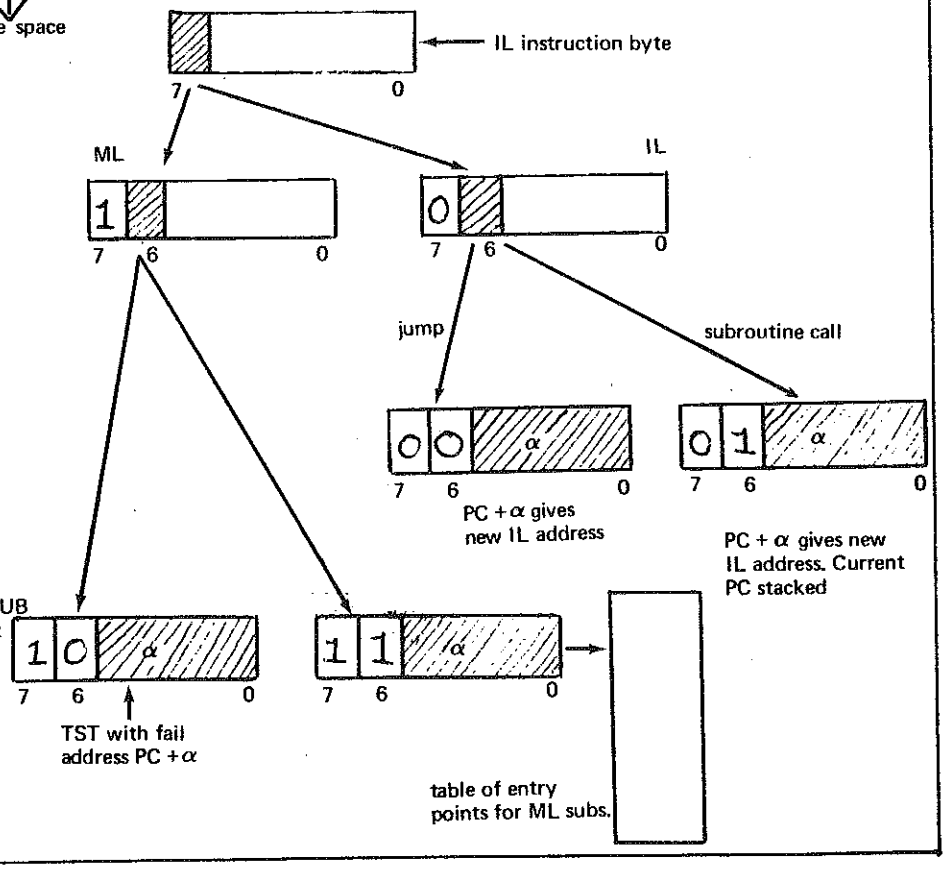
Each TINY BASIC statement is handled similarly. Some procedural section of an IL program corresponds to tests for the statement structure and acts to execute the statement.

ENCODING

There are a number of different considerations in the TINY BASIC design which fall in this general category. The problem is to make efficient use of the bits available to store information without loosing out by requiring a too complex decoding scheme.

In a number of places we have to indicate the end of a string of characters (or else we have to provide for its length somewhere). Commonly, one uses a special character (NUL = 00H for example) to indicate the end. This costs one byte per string but is easy to check. A better way depends upon the fact that ASCII code does not use the high order bit; normally it is used for parity

ONE POTENTIAL IL ENCODING



on transmission. We can use it to indicate the end (that is, last character) of a string. When we process the characters we must AND the character with 07FH to scrub off the flag bit.

The interpreter opcodes can be encoded into a single byte. Operations fall into two distinct classes---those which call machine language subroutines, and those which either call or transfer within the IL language itself. The diagram indicates one encoding scheme. The CALL operations have been subsumed into the IL instruction set. Addressing is shown to be relative to PC for IL operations. Given the current IL program size, this seems adequate. If it is not, the address could be used to index an array with the ML class instructions.

TINY BASIC INTERPRETIVE OPERATIONS

TST lbl, 'string'	delete leading blanks If string matches the BASIC line, advance cursor over the matched string and execute the next IL instruction. If a match fails, execute the IL instruction at the labeled lbl.
CALL lbl	Execute the IL subroutine starting at lbl. Save the IL address following the CALL on the control stack.
RTN	Return to the IL location specified by the top of the control stack.
DONE	Report a syntax error if after deletion leading blanks the cursor is not positioned to read a carriage return.
JMP lbl	Continue execution of IL at the label specified.
PRS	Print characters from the BASIC text up to but not including the closing quote mark. If a cr is found in the program text, report an error. Move the cursor to the point following the closing quote.
PRN	Print number obtained by popping the top of the expression stack.
SPC	Insert spaces to move the print head to next zone.
NLINE	Output CRLF to Printer.
NXT	If the present mode is direct (line number zero), then return to line collection. Otherwise, select the next sequential line and begin interpretation.
XFER	Test value at the top of the AE stack to be within range. If not, report an error. If so, attempt to position cursor at that line. If it exists, begin interpretation there; if not report an error.
SAV	Place present line number on SBRSTK. Report overflow as error.
RSTR	Replace current line number with value on SBRSTK. If stack is empty, report error.
CMPR	Compare AESTK(SP), the top of the stack, with AESTK(SP-2) as per the relation indicated by AESTK(SP-1). Delete all from stack. If condition specified did not match, then perform NXT action.
INNUM	Read a number from the terminal and push its value onto the AESTK.
FIN	Return to the line collect routine.
ERR	Report syntax error and return to line collect routine.
ADD	Replace top two elements of AESTK by their sum.
SUB	Replace top two elements of AESTK by their difference.
NEG	Replace top of AESTK with its negative.
MUL	Replace top two elements of AESTK by their product.
DIV	Replace top two elements of AESTK by their quotient.
STORE	Place the value at the top of the AESTK into the variable designated by the index specified by the value immediately below it. Delete both from the stack.
TSTV lbl	Test for variable (i.e. letter) if present. Place its index value onto the AESTK and continue execution at next suggested location. Otherwise, continue at lbl.
TSTN lbl	Test for number. If present, place its value onto the AESTK and continue execution at next suggested location. Otherwise, continue at lbl.
IND	Replace top of stack by variable value if indexed.
LST	list the contents of the program area.
INIT	Performs global initialization Clears program area, emptys GOSUB stack, etc.
GETLINE	Input a line to LBUF.
TSTL lbl	After editing leading blanks, look for a line number. Report error if invalid; transfer to lbl if not present.
INSRT	Insert line after deleting any line with same line number.
XINIT	Perform initialization for each stated execution. Empties AEXP stack.

A STATEMENT EXECUTOR WRITTEN IN IL

This program in IL will execute a TINY BASIC statement. The operators TST, TSTV, TSTN, and PRS all use a cursor to find characteristics of the TINY BASIC line. Other operations (NXT, XFER) move the cursor so it points to another TINY BASIC line.

THE IL CONTROL SECTION

```
START;  INIT                    ;INITIALIZE
        NLINE                   ;WRITE CR/LF
CO:     GET LINE                ;WRITE PROMPT & GET A LINE
        TSTL        XEC         ;TEST FOR LINE NUMBER...
        INSRT                   ;INSERT IT (MAY BE DELETE)
        JMP         CO
STMT:   XINIT                   ;INITIALIZE FOR EXECUTION
```

STATEMENT EXECUTOR

```
STMT:   TST         S1,'LET'    ;IS STATEMENT A LET?
        TSTV        S16         ;YES: PLACE VAR ADDRESS ON AESTK.
        CALL        EXPR        ;PLACE EXPR VALUE ON AESTK.
        DONE                    ;REPORT ERROR IF er NOT NEXT.
        STORE                   ;STORE RESULT.
        NXT                     ;AND SEQUENCE TO NEXT.
S1:     TST         S3,'GO'     ;GOTO OR GOSUB?
        TST         S2,'TO'     ;YES ...TO OR ...SUB.
        CALL        EXPR        ;GET LABEL.
        DONE                    ;ERROR IF er NOT NEXT.
        XFER                    ;SET UP AND JUMP.
S2:     TST         S14,'SUB'   ;ERROR IF NO MATCH.
        CALL        EXPR        ;GET DESTINATION.
        DONE                    ;ERROR IF er NOT NEXT.
        SAV                     ;SAVE RETURN LINE.
        XFER                    ;AND JUMP.
S3      TST         S8,'PRINT'  ;PRINT.
S4:     TST         S7,'"'      ;TEST FOR QUOTE.
        PRS                     ;PRINT STRING.
S6:     TST         S6,','      ;IS THERE MORE?
        SPC                     ;SPACE TO NEXT ZONE.
        JMP         S4          ;YES, JUMP BACK.
S6:     DONE                    ;NO, ERROR IF NO er.
        NLINE
        NXT
S7:     CALL        EXPR        ;GET EXPR VALUE.
        PRN                     ;PRINT IT.
        JMP         S5          ;IS THERE MORE?
S8:     TST         S9,'IF'     ;IF STATEMENT.
        CALL        EXPR        ;GET EXPRESSION.
        CALL        RELOP       ;DETERMINE (OP AND PUT ON STK.
        CALL        EXPR        ;GET EXPRESSION.
        CMPR                    ;PERFORMS COMPARISON--PERFORMS NEXT IF FALSE.
        JMP         STMT        ;GET NEXT STATEMENT.
S9:     TST         S12,'INPUT' ;INPUT STATEMENT.
S10:    CALL        VAR         ;GET VAR ADDRESS.
        INNUM                   ;MOVE NUMBER FROM TTY TO AESTK.'
        STORE                   ;STORE IT.
        TST         S11,','     ;IS THERE MORE?
        JMP         S10         ;YES.
S11:    DONE                    ;MUST BE er.
        NXT                     ;SEQUENCE TO NEXT.
S12:    TST         S13,'RETURN' ;RETURN STATEMENT.
        DONE                    ;MUST BE er.
        RSTR                    ;RESTORE LINE NUMBER OF CALL.[
        NXT                     ;SEQUENCE TO NEXT STATEMENT.
S13:    TST         S14,'END'
        FIN
S14:    TST         S15,'LIST'  ;LIST COMMAND.
        DONE
        LST
        NXT
S15:    TST         S16,'RUN'   ;RUN COMMAND.
        DONE
        NXT
S16:    TST S17,'CLEAR'         ;CLEAR COMMAND.
        DONE
        JMP START
S17:    ERR                     ;SYNTAX ERROR.

EXPR:   TST         E0,'-'
        CALL        TERM        ;TEST FOR UNARY -.
        NEG                     ;GET VALUE.
        JMP         E1          ;NEGATE IT.
E0:     TST         E1,'+'      ;LOOK FOR MORE.
        CALL        TERM        ;TEST FOR UNARY +.
E1:     TST         E2,'+'      ;LEADING TERM.
        CALL        TERM        ;SUM TERM.
        ADD
        JMP         E1
E2:     TST         E3,'-'      ;ANY MORE?
        CALL        TERM        ;DIFFERENCE TERM.
        SUB
        JMP         E3
E3: T2: RTN                     ;ANY MORE?
```

```
TERM:   CALL    FACT
TO:     TST     T1,'*'
        CALL    FACT        ;PRODUCT FACTOR.
        MPY
        JMP     TO
T1:     TST     T2,'/'      ;ANY MORE?
        CALL    FACT        ;QUOTIENT FACTOR.
        DIV
        JMP     TO

FACT:   TSTV    F0          ;VARIABLE.
        IND                 ;YES, GET THE VALUE.[
        RTN
F0:     TSTN    F1          ;NUMBER, GET ITS VALUE.
        RTN
F1:     TST     F2,'('      ;PARENTHESIZED EXPR.
        CALL    EXPR
        TST     F2,')'      ;MATCHING PARENTHESIS.
        RTN
F2:     ERR                 ;ERROR.

RELOP:  TST     R0,'='
        LIT     0           ;=
        RTN
R0:     TST     R4,'<'
        TST     R1,'='
        LIT     2           ;<=
        RTN
R1:     TST     R3,'>'
        LIT     3           ;<>
        RTN
R3:     LIT     1           ;<
        RTN
R4:     TST     S17,'>'
        TST     R5,'='
        LIT     5           ;>=
        RTN
R5:     TST R6,'<'
        LIT 3               ;<>
R6:     LIT 4               ;>
        RTN
```