tiny-c OWNER'S MANUAL

A Home Computing Software System

Thomas A. Gibson

and

Scott B. Guthery

Published by tiny c associates, Post Office Box 269, Holmdel, New Jersey 07733. Prepared on an LSI-11 processor running a modified version FORMAT by the Life Support System Group, and set on a Diablo 1620. Printed by the Trenton Printing Company, Inc., Trenton, New Jersey.

First Printing: June 1978
Second Printing: June 1979

001041

DISCLAIMER OF WARRANTIES AND LIMITATION OF LIABILITIES

The authors have taken due care in preparing this book and the programs in it, including research, development, and testing to ascertain their effectiveness. The authors and tiny c associates make no expressed or implied warranty of any kind with regard to these programs nor the supplementary documentation in this book. In no event shall the authors or tiny c associates be liable for incidental or consequential damages in connection with or arising out of the furnishing, performance or use of any of these programs.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

We are indebted to many friends and tiny-c users. Their response to tiny-c, both complimentary and critical, has helped us improve and extend the product.

The tiny-c installers -- Lou Katz, Dennis O'Neill, Jim Goodnow, Dale Walker, Morris Krieger, and Ira Ellenbogen -- have enabled us to spread the tiny-c gospel to territories we could not have explored alone. We thank them for their missionary zeal.

Most of all, we would like to thank Irene Gibson and Maria Nekam for their continuing efforts, patience and encouragement. Without their dedication to the daily responsibilities of a mail-order business, this enterprise would not be possible.

The amount of toil that an idea can absorb from its inception to its realization continues to astonish us all.

Thomas A. Gibson
Scott B. Guthery

May 1979

PREFACE

The sources of ideas that went into tiny-c are many. First there is BASIC [Kemeny & Kurtz 1967]. BASIC has become the de facto standard training language in the United States. It is popular in high schools, universities, even in industry, where it is used for some production work. Although BASIC has its faults, its one big strength is that it is easy to learn. This is largely because it offers a single computing environment. You can enter new program lines, change old ones, and start a program running all from one command environment. You do not have to remember the environment you are in, i.e., edit mode, compile mode, link mode, system mode, run mode, etc., when giving a command. There are no commands to shift from mode to mode. There are no relocatable object modules, link editors, and all the other paraphernalia of "real" computers. It is very simple and very adequate. Thus a focus is made on the essential elements of computing, as opposed to the elements of "wrestling" with a computer.

The LOGO language [Feurzeig 1975] is in many ways similar to tiny-c. It offers a well-structured language based on BASIC, as well as a single environment for programming and execution. LOGO was used experimentally in public schools with very young children. The experiment showed that children could grasp simple computer concepts and work through a prepared set of exercises, and then do creative work of their own.

C [Ritchie, Kernighan, & Lesk 1975] is a computer language designed by Dennis Ritchie, at Bell Telephone Laboratories. tiny-c borrows its overall structure from C. C is broadly used in universities and in industry. It has been used to program a very advanced and powerful computer operating system, called UNIX™ [Ritchie & Thompson 1974]. And yet it

---

™ UNIX is a trademark of Bell Laboratories.

is a very simple language. C has no native input/output, e.g., read or print statements. Input/output is done using functions. Thus C concentrates on COMPUTING facilities, and allows external development or elaborations of input/output. tiny-c has adopted this idea.

The command environment for tiny-c is written in tiny-c. It needs no translation to the micro-processor's machine language. This corresponds somewhat to the idea of using C as the programming language to implement UNIX. So, although intended as a training language for structured programming, tiny-c is a powerful language.

The tiny-c OWNER'S MANUAL is trying to reach four audiences at the same time. For those new to structured programming we have a brief tutorial and program walk-through so they can get the gist of it without getting bogged down in details. Experienced users of structured programming will find that the reference sections let them quickly discover the features of tiny-c. For those who want to know how the tiny-c interpreter works, we have described its operation. And, finally, for those who want to install tiny-c on their home computer, we have included a complete installation guide.

NOTES ON THE SECOND PRINTING: Several factual and clarification edits have been made in the text for this printing. The only major change is to Appendices A, B and C, where all the fixes in Newsletters 1 and 2 have been incorporated.

In Appendix A (8080), several "housekeeping" changes have been made. These include assembler calculated space allocations (BSTACK, ESTACK, etc.) and incorporation of patches XX1 through XX8 in line. This new version is labeled 80-01-02. A program to relocate 80-01-02 analogous to the Relocate Program in Chapter VI for 80-01-01 is also included in Appendix A.

The PDP-11 version of tiny-c has been derived from the compilation of the tiny-c interpreter written in the C programming language. This rendering of tiny-c in its big brother is also included in Appendix B.

FOREWORD

C is a versatile, expressive general-purpose programming language which offers economy of expression, modern control flow and data structures, and a rich set of operators. C is not a "very high level" language, nor a "big" one, nor is it specialized to any particular area of application. But its absence of restrictions and its generality make it remarkably convenient and effective for a wide variety of computing tasks.

C is concise -- you don't have to write a lot to get a job done. Yet at the same time, C programs are readable -- you can understand what you (or someone else) have written. This combination of brevity and readability is rare in programming languages, and is part of the reason that C is so widely used.

With tiny-c, Tom Gibson and Scott Guthery have designed a stripped-down version of C that is well-adapted to the microcomputer environment. tiny-c retains C's expressiveness, conciseness and readability, yet sacrifices very few of its features.

At the same time, tiny-c provides a computing environment that will make it easier to develop programs. It comes with an editor and other piece parts that together make a program preparation system.

The tiny-c OWNER'S MANUAL is more than a reference manual for tiny-c, however. It is also a vehicle for conveying ideas and insights about how to get the most out of your machine, and about good programming in general.

C has simply taken over in many computing environments, not because people have been ordered to use it, but because it is a good language. It seems very likely that tiny-c will have a similar effect in the microcomputer world.

Brian W. Kernighan
Bell Laboratories
Murray Hill, New Jersey

May 9, 1978

I. INTRODUCTION

What is tiny-c? tiny-c is

a language, plus
a standard library, plus
a program preparation system.

Without any other software aids, you can prepare tiny-c programs, run them, edit them, store them on a cassette or floppy disk, and read them back later.

tiny-c is a structured programming language which has if-then-else, while-loops, functions, global and local variables, and character and integer data types, pointers, and arrays.

tiny-c is independent of operating systems. You can interface it easily to the input/output routines on your computer.

tiny-c can invoke your own machine language subroutines so the tiny-c programming language can be fitted to your system and your system can be reflected in and extend the language.

1.1 A tiny-c Program Walk-Through

Figure 1-1 is a complete tiny-c program consisting of two functions.

FIGURE 1-1

```
/*guess a number between 1 and 100
/* T. A. Gibson, 11/29/76

guessnum [
   int guess, number
   number = random (1,100)
   pl "guess a number between 1 and 100"
   pl "type in your guess now"
   while (guess != number) [
       guess = gn
       if (guess == number) pl "right !!"
       if (guess > number) pl "too high"
       if (guess < number) pl "too low"
       pl""; pl""
   ]     /* end of game loop
]        /* end of program

/*
/* random-generates a random number

int seed, last   /* globals used by random
random int little, big [
   int range
   if (seed == 0) seed = last = 99
   range = big - little + 1
   last = last * seed
   if (last < 0) last = -last
   return little + (last/8) % range
]
```

End of FIGURE 1-1

How does this program work? Let's do a program walk-through:

Starting at the top, the first two lines are COMMENTS. A comment starts with /* and goes to the end of the line.

"guessnum" is the name of a FUNCTION which is called to start the program.

Following "guessnum" is a COMPOUND STATEMENT, which is 12 lines long, the last line being:

```
]       /* end of program
```

A compound statement is everything between balanced left-right brackets.

The first SIMPLE STATEMENT in guessnum is:

```
int guess, number
```

This declares two INTEGER VARIABLES named "guess" and "number". All variables in tiny-c must be declared. When executed, the int statement will create the variables, and give them an initial value of zero.

The second simple statement in guessnum is

```
number = random (1,100)
```

This sets number equal to the value of the tiny-c program function random executed with its first ARGUMENT equal to 1 and its second argument equal to 100. In our program the function random returns a random number between 1 and 100.

On the next line, pl is a tiny-c LIBRARY FUNCTION which prints a line. It prints the quoted STRING which is its argument.

while sets up a LOOP. The general form of while is:

```
while (expression) statement
```

In this instance, the EXPRESSION part is

```
guess != number
```

where != means not equal to. This expression is evaluated, and if it is true, the statement is done, and then the expression is evaluated again. If it is false, the statement is skipped. Initially, guess is 0 and number cannot be less than 1, so the expression is initially true. Therefore the statement is executed.

The statement is compound, and is composed of six simple statements. The first of these statements is

```
guess = gn
```

gn, which stands for "get number", is another standard library function. It reads a number typed in by the user at the terminal, and returns that value. So here the program waits until the user types a number and a carriage return, and then guess is assigned the number typed.

The next three simple statements are if statements. The general form of the if statement used here is

```
if (expression) statement
```

where statement is executed if expression is true.

Statements five and six of the while's compound statement are pl"". pl"" goes to a new line, and prints nothing. The semicolon allows you to write more than one simple statement on the same program line. So

```
pl"" ; pl""
```

prints two blank lines.

Now we are at the end of the while loop. Since the expression part of the while was true, the while statement is executed again. This starts with another evaluation of the expression to see if it is true or false. If the first guess is not equal to number, the compound statement is executed again. Another guess is read, the appropriate remark is made, and two more blank lines are printed; the while is done yet again. Eventually the user gets the right number and guess is equal to number. This will cause a "right!!" and two blank lines to be printed. The while condition is then tested again. The expression guess != number is evaluated and found to be false, so the entire compound statement part of the while is skipped, which brings us to the end of guessnum. The game is over. The program stops because the end (the last]) of guessnum is reached.

Before we walk through random, notice the integers seed and last are declared outside of both guessnum and random. They are called GLOBAL VARIABLES. They will be created once when the program is started. They are initially zero, and are

known and usable by both guessnum and random. On the other hand, guess, number and range are LOCAL VARIABLES. guess and number are known and usable only within guessnum, while range is local to random.

The first line of random gives the function name. And, before the [, it declares two integer arguments, little and big. A VALUE must be supplied for each argument when a function is called. The call in the sixth line of guessnum sets little to 1, and big to 100. Now we enter the BODY of the function random.

range is declared an integer and is initially zero. On the first call, seed is zero. Now seed and last are both set to 99. range is calculated, and is 100. last is set to the product of last and seed which is 9801. This is not less than 0, so the statement part of the if is not evaluated.

We next come to the return statement. It does two things. First, it evaluates the expression. The result is made the VALUE OF THE FUNCTION. Second, it returns control to the program that called the function. tiny-c expressions are similar to algebraic expressions. The symbol + means add, / means divide, and - means subtract (or take the negative). To indicate multiplication, a * is used. An unusual symbol is %, which means divide the left side by the right side and take the REMAINDER (not the quotient). So, for example,

```
1225 % 100
```

is 25.

Thus the return statement calculates the expression:

```
little + (last/8) % range
= 1  + (9801/8) remainder 100
= 1  +   1225   remainder 100
= 1  +    25
= 26
```

The value 26 is returned as the value of function random. It also leaves 9801 in last, and 99 in seed. Since these are

global variables, their values are retained between function calls. This is not true of local variables like range. Their values are retained only during the execution of the function in which they are defined. When that function is left their values are lost.

On a second call to random, range is recreated, and reinitialized to zero. seed is not zero, so seed and last are not set to 99, but remain 99 and 9801 respectively. range is recalculated as 100. Then

```
last = last * seed
     = 9801 * 99
     = 970299
```

This number is too big for tiny-c. Any computer has a limit on the size of the numbers that can be computed. tiny-c numbers must be in the range

```
-32768 <= number <= 32767.
```

last OVERFLOWS this range. It will be assigned the value -12741! (We explain this more completely in Section 2.11.) This is less than 0, so the next statement assigns last the value 12741. Then the return statement calculates:

```
  1 + (12741/8) remainder 100
= 1 + 1592 remainder 100
= 93
```

This is returned as the second value of random.

REVIEW OF THE WALK-THROUGH

The purpose of the walk-through is to get a feeling for programming in tiny-c. We have seen that

* A tiny-c program is a set of functions.
* Some functions are standard library functions, like gn and pl.
* Global variables stay around and hold their values. Local variables come and go.

* Function and variable names can be as long as you want.

* A group of statements enclosed in brackets makes a compound statement which can be treated just like a simple statement.

1.2 Structured Programming -- What tiny-c Is All About

Perhaps you have heard structured programming described as "go-to-less" programming. Or programming with just if-then-else and do-while control statements. Such remarks oversimplify what structured programming is all about. The essence of structured programming is PROGRAM CLARITY. You can write programs in small, modular parts, with easy-to-follow program flow. You can use well-chosen, descriptive variable names. This leads to clear, understandable programs. Program clarity is what structured programming is all about.

We discuss here four principle ideas that make program clarity possible. These are: modularity, predictable program flow, local variables, and the simple idea of meaningful variable names.

MODULARITY in software is just as important as modularity in hardware. It makes it humanly possible to deal with complexity. A module is a brick or atom used for building bigger modules. Seen from within, a module may be very complex but from the outside it is an indivisible whole. Software modularity is achieved through the use of FUNCTIONS.

PROGRAM FLOW is predictable if you can point to any statement and easily answer the question "under what conditions is this statement executed?" This is particularly important if the program is 20 or 30 pages long, and still has bugs. Scanning the whole program and drawing arrows is no fair. That's not considered an easy way to answer the question. Predictable program flow can be achieved in many ways. In tiny-c, it is done with COMPOUND STATEMENTS.

Functions also make it possible to hide variables used in a strictly local context. The variable n is very popular; it's used frequently to count things. Have you ever had a program blow up because you were using n in two places for two purposes? The fix was to change one of them to n1. A better idea is in the concept of LOCAL and GLOBAL variables.

As for long, MEANINGFUL NAMES for variables and functions -- just look at the sample programs to see the improvement.

David Gries suggests structured programming be called "simplicity theory", and characterizes it as "an approach to understanding the complete programming process" [Gries 1974]. As a pleasant dividend, structured programming is more enjoyable than monolithic programming. It should certainly, therefore, be a part of personal computing. To begin our look at tiny-c as a structured programming language, let's look at the foundation of functions and predictable program flow -- the compound statement.

1.2.1 Compound Statements

When you write a program, you write a list of statements:

```
x = x-1
a = b+c
b = b*2-c
x = b-a
```

The idea behind a compound statement is to make one statement -- a molecule -- out of a set of statements -- some atoms. This is done in tiny-c by

```
[x = x-1
 a = b+c
 b = b*2-c
 x = b-a]
```

Anywhere you can write a simple statement you can also write a compound statement. This sounds simple, but the effect is powerful. For example most programming languages have an if statement similar to this:

```
if (logical expression) statement
```

So you can write

```
if (x>Ø) x = x-1
```

But make the statement part compound, and you have this capability:

```
if (x>Ø) [
    b = b*2-c
    a = b+c
    x = x-1
]
```

This multiline if is not some special kind of if. It is still:

```
if (logical expression) statement
```

But the statement part is compound. The compound statement is treated as an indivisible unit. It is either all done or all not done depending on the value of the logical expression.

The compound statement also is a natural for LOOPS. There is a big difference among the various programming languages in how you write loops, but they all have one thing in common. A loop has a beginning and an end. A compound statement can be used to express this. The looping statement is:

```
while (logical expression) statement
```

Notice the similarity with the if. Only the keyword has changed. Here's how while works. The logical expression is evaluated. If it is true, then the statement is executed, and then the while is done again. The effect is a repeated if, i.e., a loop. As long as the logical expression remains true, the statement is done again and again. Eventually something in the statement causes the logical expression to become false, and the loop terminates. Of course, the statement can be compound, as in:

```
while (x>Ø) [
  a = b+c
  b = b*2-c
  x = x-1
]
```

The compound statement is a natural way to delimit the beginning and end of loops.

With one simple idea, the compound statement, two things are achieved. The if statement is more powerful than is common in non-structured programming languages. The concept of a loop collapses to a simple repeated if or while statement. In both situations you are stating conditions under which the statement -- whether simple or compound -- is to be executed.

1.2.2 Nesting Compound Statements

> ANYWHERE YOU CAN WRITE A SIMPLE STATEMENT, YOU CAN WRITE A COMPOUND STATEMENT.

That is a fundamental rule. A compound statement contains simple statements. Therefore a compound statement can contain compound statements. Figure 1-2 illustrates this.

FIGURE 1-2

```
[ x = x-1
  a = b+c
  b = b*2-a
  x = b-a
]
```

a=b+c is a simple statement. The rule says a compound statement can be written here. For example:

```
[ x = x-1
  [   a=b+c
      w = y+2*x+w
      y = 17
  ]
  b = b*2-a
  x = b-a
]
```

End of FIGURE 1-2

The substitution of a compound for a simple shown in Figure 1-2 is certainly allowable, but is of no practical value.

The real utility in nested compounds is in writing nested if and while statements. Figure 1-3 is therefore a more realistic example of the use of compound statements.

FIGURE 1-3

```
if (x>Ø) [
    while (x<limit) [
        if (case==1) [
            y = Ø; w = 99
        ]
        if (case==2) [
            y = 99; w = Ø
        ]
        nextaction
        x = x+1
    ]
]
```

End of FIGURE 1-3

In Figure 1-3, if you remove everything except the brackets, you have this:

```
[ [ [ ] [ ] ] ]
```

This is what is meant by nested compound statements. Brackets are used to form program units the same way parentheses are used to create arithmetic statements. The main difference is that a pair of brackets is preceded by a function name, or a logical expression. In the first case you're naming the contents of the brackets and in the second you're stating the conditions under which the contents are to be executed.

1.2.3 Readable Program Flow

In Figure 1-3, look at the "y=0" in the fourth line. How can it be reached? Only if case is 1, and x is less than limit. No go-to can lead here, either accidentally or on purpose.

How can "nextaction" be reached? Only if x is less than limit, and then only after possible changes to y and w. This

program has simple, predictable flow. The only way a statement other than a while can be reached is from directly above. whiles can also be reached from their matching] below.

1.2.4 Indenting and the Placement of Brackets in Compound Statements

The brackets alone define the "structure" of a program. Indenting means nothing. But one of the purposes of structured programming is to make programs more readable and, hence, more understandable. A good choice of indenting style is very important to program readability. There are several styles to choose from. The actual choice is not too important. But once you choose a style, stick to it. Consistency IS important.

One easily explained style is to align matching brackets vertically. This looks like:

```
if (x<Ø)
[   statement
    statement
        "
        "
        "

]
```

A problem with this is that when editing the first statement, care must be taken to keep the [intact. So some use this style:

```
if (x<Ø)
[
    statement
    statement
        "
        "
        "

]
```

This takes an extra line. Also there is a visual break between the if and its statements. So some take the left bracket and move it to the end of the preceding line:

```
if (x<Ø) [
    statement
    statement
        "
        "
        "
]
```

The right bracket is now vertically aligned with the if or while that preceded the compound statement.

You may pick one of these, or invent a style of your own. But, we repeat, whatever you decide to do, do it consistently.

1.2.5 Functions

A large software project can usually be broken into natural parts, and each part programmed and debugged as a separate unit. Each of these units then becomes a reliable building block for the construction of still larger parts of the project. Sometimes units can be designed to be useful in many projects.

In various programming languages these building blocks are called subprograms, subroutines, or, as in tiny-c, FUNCTIONS. Here is a tiny-c function for any computer versus human game:

```
game [
    getready
    while (stillplaying ()) [
       humanturn
       if (stillplaying ()) computerturn
    ]
    gameover
]
```

The name of the function is "game". The compound statement that follows is called the body of the function. Each [can be read as "do all of this", and its matching] read as "end of this". game divides the design of a game program into five parts:

getready (which initializes things, and prints instructions if requested),

stillplaying (which determines if the game is still going, and returns true if it is, otherwise false),

humanturn (which conducts the human's turn),

computerturn (which conducts the computer's turn),

gameover (which computes and prints scores, makes remarks about the human's skill, promotes the human, or whatever).

The game function is the first step in divide-and-conquer or top-down program development. Let's carry this development one step further. The getready function can be expanded this way:

```
getready [
    ps "Do you want instructions?"
    if (gc()=='y') instructions
    setupboard
]
```

getready divides the initialization into two parts: instructions, and setupboard.

(Note: ps prints a character string, gc() reads a character, and == 'y' tests if the character is a y.)

Notice that both game and getready are universal. They can be used in many game programs. Programming in this fashion eventually leads to a library of useful, general purpose

functions. These can be pulled off the shelf into a software project. You know they work because they were used before. Your programming becomes more productive, and more pleasant.

The next time you're programming a sizable project, i.e., anything more than a page, try to identify subsets of the logic usable in other projects. Capture these as functions. It is gratifying to discover a general purpose function where none was suspected.

1.2.6 Local and Global Variables

A LOCAL VARIABLE is one that is known only inside a function. It can be used and changed only within the body of the function. Even its name is unknown outside the function. In fact, its name can be used in other functions without conflict. This is what makes local variables useful.

Take a look at Figure 1-4. There are four local variables in these two functions. The variables n and maximum are local to afunction. The variables n and total are local to anotherfunction. If either of these functions calls the other, the values of n will not be confused since they only have meaning inside the body of their own functions. It helps to think of local variable names as being preceded by the possessive form of the function to which they are local. For example, afunction's n and anotherfunction's n.

FIGURE 1-4

```
afunction [
    int n, maximum
    n=Ø
    while (n<maximum) [
            .
            .
            .
        n=n+1
    ]
]
anotherfunction [
    int n, total
            .
            .
            .
    n = n+2
    total = total+n
            .
            .
            .
]
```

End of FIGURE 1-4

The value of locals is obvious to anyone who has spent a nasty debugging session trying to find out where, in a huge program, some variable is getting changed.

Of course not all variables can be local. Some must be shared by many functions. These are called GLOBALS. They should be used infrequently, as they do cause debugging headaches. Choosing good, descriptive names for globals alleviates the problem. A global named "k" is inviting disaster. Call it "klingonsleft" and you're less likely to accidentally use it for two purposes. Also you've given a reader of your program a pretty good clue to the variable's use.

1.2.7 Summary -- And Where We Go from Here

We've walked through a simple program to get a feel for tiny-c, and we've discussed the virtues of structured programming. These are just the preliminaries. Now it's time for the main events. First, a complete definition of the tiny-c language. Chapter II is devoted to this task. To prepare programs you need an editor and a way of debugging. The Program Preparation System (PPS) is described in Chapter III. Examples are excellent learning tools: Chapter IV has several sample programs. Maybe you want to make it bigger, better, or faster? Chapter V explains how tiny-c works. Finally, of course, you'll want to get tiny-c up and running on your own computer. Chapters VI and VII explain how to install tiny-c on an 8080 or PDP-11™.

™ PDP is a trademark of Digital Equipment Corporation.

II. THE LANGUAGE

The tiny-c programming language is described completely here.

2.1 Comments

Comments begin with /* and continue to the end of the line. Apostrophes ('), quotes ("), and brackets ([]) should not be used in comments.

2.2 Names

Names of functions and variables can be one or more characters long. If more than eight characters are used, only the first seven and the last are significant. The first character must be alphabetic, either upper or lower case. The rest must be alphanumeric. Names cannot have imbedded blanks. Upper and lower case are considered distinct, so

```
GUESS
guess
```

are different names.

Names may NOT begin with any of these:

```
if
else
while
return
break
char
int
MC
```

2.3 Data and Variables

There are two kinds of data, integers and characters. A DATUM is an actual value:

```
 7 is an integer datum
'a' is a character datum
```

A VARIABLE is a named cell that holds one datum. A variable must be created by declaring its existence and the type of datum it can hold in an int or char statement. For example, the two tiny-c statements

```
int a,b
char letter
```

declare the existence of three variables; two integer variables named a and b and one character variable named letter. Each can hold one datum of its respective type. Initially, integers are 0 and characters are ASCII null, i.e., also 0.

2.3.1 Arrays and Array Elements

An ARRAY is a list of variables of the same type.

```
int value (10)
char buffer (80)
```

declares an array with eleven integer elements, and a character array with 81 elements.

An array element is picked out using a subscript. For example,

```
value(7)
```

names the seventh element of the array value. Every array has a zero-th element

```
value(0)
```

and a last element

```
value(10).
```

When you declare an array, you name its last element. Thus, value has eleven elements:

```
value(0), value(1), ..., value(10)
```

The subscript of an array can be any expression.

```
value(i + 10 * k)
```

Even in a declaration, the subscript can be an expression. This is a convenient way of setting several arrays to the same or related sizes.

```
int size
size = 10
int x(size), y(size), matrix(size*size)
```

Note that matrix is NOT a two-dimensional array, but a single list of 101 elements. However, it can be addressed as a two-dimensional (0-9, 0-9) matrix this way:

```
int row,col
row = 7
col = 3
matrix(row + size * col) = ...
```

2.3.2 Locals, Globals, and Arguments

There are three scopes of variable declarations.

Locals: Local variables are declared within the body of a function (i.e., inside the [] part of the function.)

Arguments: Function argument variables are declared after a function name, and before the [beginning the function body.

Globals: Global variables are declared outside all functions.

In Figure 1-1, guess, number and range are local variables. The first two, guess and number, are local to the function

guessnum. The last, range, is local to the function random. The variables little and big are arguments. The variables seed and last are globals as are the function names guessnum and random.

Locals are created when a function is entered, and destroyed when the function is exited. When they are created they are also set to 0, (ASCII null, for characters).

Function arguments are always copied into a function, and then treated as locals.

Global variables may be accessed by all functions and preserve their values between function calls.

Within a function, the following names can be used:

locals for the function,
arguments of the function,
all globals for the program, and
all functions for the program.

Technically, arguments are locals, and function names are globals, so this rule is easier to remember as:

A FUNCTION CAN USE ITS OWN LOCALS, AND ALL GLOBALS.

All the locals within a function must have different names. But two different functions can each have their own local variable named x, and the two x's are kept separate.

All global names including function names must be different.

Duplicate names are not detected or diagnosed. A program will execute, but the second declaration of the name will be ignored. The first declared name is always used.

One form of duplication is important. You can have a local and global variable of the same name. They are kept separate. Within the function that has the local, the local name prevails. Elsewhere, the global prevails.

2.4 Expressions

Expressions are formed from operators, parentheses, and primaries. They are used to calculate and store data, and to invoke functions.

2.4.1 Primaries

Primaries designate the data and/or destinations for results of expressions. They are the atomic elements of expressions. There are six types of primaries:

primaries	examples
constants	10, 'c'
strings	"hello"
variables	x
subscripted variables	buff(7)
array names	buff
functions	gn, ps "hello"

An integer constant may be signed. Its value must be between -32768 and 32767, inclusive. An integer uses two bytes of storage.

A character constant is always enclosed in apostrophes. A character uses one byte of storage.

Integers and characters are completely interchangeable in expressions. A character variable may be used as a one-byte integer whose value is in the range -128 to 127. This is occasionally useful, as in:

```
char newline, ch, digit
newline = 10      /* Puts an LF in new line.
ch = getchar
digit = ch - '0'      /* Converts an ASCII digit to
                      /*    its integer value.
```

A character string constant is technically a two-byte constant which has as its value the address of the first element of an array of characters. Thus,

```
"hello"
```

is the address of the first element of an array of six characters initialized with h-e-l-l-o-null. Two-byte constants or variables which may also be used as addresses are called pointers. Thus, a character string is a pointer to its first character. Pointers are covered in detail later.

A subscript expression may be an arbitrary expression. The smallest subscript is 0, the largest is the declared size of the array. If an array's subscript falls outside these bounds, a subscript error is recognized. An exception to this rule is when an array is declared with size 0. Then any positive or negative subscript may be used. In effect, such an array can access any element in the direct address space of the computer.

Since function names have values, they may be used in expressions as though they were variable names. The value of a function name is the value returned by the function program of that name. In Figure 1-1, the use of the function name random

```
number = random(1,100)
```

is an example of the use of a function name as a variable.

2.4.2 Operators

The tiny-c operators are used to do arithmetic, compare values, and assign values to variables. We first show their

use in simple circumstances using one or two primaries. Then we consider more complex uses.

OPERATOR	USE	DEFINITIONS
unary +	+a	When used alone to the left of a variable, the plus sign is called a unary plus. It has no effect, and is used sometimes for readability.
unary -	-a	When used alone to the left of a variable, the minus sign is called a unary minus. The value of -a is the negative of a.
*	a*b	Multiplication. The value is a multiplied by b.
/	a/b	Division. The value is a divided by b. If there is a fraction, it is discarded, so the result is an integer. So 7/2 is 3. Also -7/2 is -3.
%	a%b	Remainder. The value is the remainder of a/b. So 7%2 is 1. Also -7%2 is -1.
+	a+b	Addition. Also called plus or binary plus. The value is the sum of a and b.
-	a-b	Subtraction. Also called minus or binary minus. The value is the difference between a and b.
<	a<b	Less than. The value is 1 if a is arithmetically less than b. Otherwise it is 0.
>	a>b	Greater than. The value is 1 if a is arithmetically greater than b. Otherwise it is 0.

OPERATOR	USE	DEFINITIONS
<=	a <= b	Less than or equal to. The value is 1 if a is less than or equal to b. Otherwise it is 0.
>=	a >= b	Greater than or equal to. The value is 1 if a is greater than or equal to b. Otherwise it is 0.
==	a == b	Equal to. The value is 1 if a and b are equal. Otherwise it is 0.
!=	a != b	Not equal to. The value is 1 if a and b are not equal. Otherwise it is 0.
=	a=b	Assignment. a is assigned the value b. Then the expression a=b assumes the value of b.

USES OF ASSIGNMENT: The assignment operator = can be used anywhere a binary + or - can be used. For example,

```
x(k=k+1) = a-(b=c/d)
```

performs three assignments. b is set to the value of c/d. k is set to k+1. The array element x (new value of k) is set to a minus new value of b. Also consider

```
a = b = c = 0
```

c is set to 0. Then b is set to c, i.e., to 0. Then a is set to b, also 0.

ORDER OF EVALUATION: When you write expressions with 3 or more primaries, the order of evaluation becomes important. For example, is

```
9 + 6/3
```

equal to 5 or 11? Standard algebra conventions would do the division first, then the addition. So the answer is 11. tiny-c works the same way. All the operators are assigned a PRECEDENCE. In the absence of parentheses, the operators with the highest precedence are done first.

precedence	operators
highest	unary + unary -
	* / %
	+ -
	< > <= >= == !=
lowest	=

So the value of

7+3<5 is (7+3)<5 is 0 [not 8].

-1+7 is (-1)+7 is 6 [not -8].

a = 1+c = 2 is a = (1+c) = 2 is illegal, since you may not set an expression, 1+c, to anything.

But

a = 1+(c=2) sets c to 2 and a to 3.

a = 1+c == 2 is a = ((1+c) == 2) which sets a to 1 if 1+c is 2; otherwise sets a to 0.

All the above cases are resolved by the precedence rule, but sometimes that is not enough. For example, is

7 - 2 + 1

equal to 4 or 6? Standard algebra would say 6. Note that + and - have the same precedence, so we cannot use precedence to determine which goes

first. The tiny-c convention is that the evaluation is done left to right. So,

```
7-2+1 = (7-2)+1 = 6.

7/2*2 = (7/2)*2 = 6.  [Remember 7/2 is 3.]
```

But what about

```
a = b = c = 0
```

This is the exception. A series of assignments is done right to left.

USE OF PARENTHESES: Parentheses are used to change the order of performing operations. So in our very first example, if the desired result was 5, you can write it

```
(9+6)/3
```

An expression within parentheses is evaluated and then this value is used with operators outside the parentheses. Within parentheses, precedence and grouping rules determine order. So

```
22/(9+6/3) is 22/11 is 2 [not 4]
```

because the precedence rule says 9 + 6/3 is 11.

2.5 Functions

A function can be used as a primary anywhere in an expression (except to the left of an assignment.) When a function is used in an expression, we say the function is CALLED. When you call a function, it must be defined somewhere in your program, be in the standard library, or be the special function MC.

Every function is defined with a specific number of arguments. random has two arguments, little and big. When a function is called, values must be supplied for each of the

function's arguments. If you supply too many or too few values, an arguments error is recognized. Thus, for example, random must always be called with two arguments, ps must always be called with one, and gn must always be called with none.

The argument values are written as a list of expressions separated by commas. The list, even if empty, can always be enclosed in parentheses, and sometimes must be. Arguments themselves may invoke functions. Arguments are evaluated left to right. Here are some legal calls on random.

```
random (1, 100)
random (gn (), gn())
random (k = random (-10,10), k+10)
```

Notice each call to random has two arguments, because random is defined with two arguments.

If an argument is an array, then the supplied value must be a pointer of the same data type. For example, the argument to the library function pl is a character array. So a character pointer is the only valid argument.

```
pl "hello"
```

is valid, because "hello" is a character pointer to a string initialized with h-e-l-l-o-null. Other cases of pointer values are described in Section 2.6.

If an argument is not an array, then ANY value can be supplied. But usually it will not be a pointer. Neither argument to random is an array. So the supplied values may be of any type.

```
random 1,100
```

returns a number between 1 and 100.

```
random 'a','z'
```

returns a random lower case letter.

```
char a(100)
random a,a+100
```

returns a random address within a. Of course, this rule also applies to function definitions which are included in a tiny-c program. In the following example, the function len has one array argument and one non-array argument:

```
     char a (10)
     int k,l
     k = len (a,1) /* a is a pointer
                   .
                   .
                   .
/* definition of len function
len
     char string (10) /* string is an array
     int n            /* n is not an array
[                  .
                   .
                   .
```

Parentheses around the argument list are always allowed. tiny-c allows them to be removed in certain cases. This is done principally to make input/output functions more legible. In the following forms, omitting parentheses around arguments is allowed.

```
k = function arg, arg, arg

k = function

function arg, arg, arg

function
```

The general rule is:

> IF A FUNCTION AND ITS ARGUMENTS ARE THE LAST PART OF AN EXPRESSION, ITS PARENTHESES MAY BE OMITTED.

Whenever the function is involved in a more complex expression, parentheses must be used. For example

```
number = gn
```

is allowed, but gn in the expression

```
number = (gn () + 10)/2
```

needs the parentheses as shown.

There is no problem with complicated function arguments without parentheses. So this

```
pn 7 + 11/w - 142/g - x/17 + x%42
```

will print a number.

If the first argument begins with (, the argument list must be enclosed in parentheses. For example:

```
pn (7+2)/3
```

will do this:

a) determine that (7+2) is the argument to pn,
b) call pn, which prints a 9,
c) pn returns a 0 as its value,
d) 0/3 is evaluated, and the result discarded.

This is probably not what was desired. To print the value of 7+2 divided by 3 you should write

```
pn ((7+2)/3)
```

When a function is invoked with arguments, the value of each argument is passed to the function. A local copy is made within the function. The function can change the local copy, but this will not change the original.

For example

```
        x = 10
        blast x
        pn x
          .
          .
          .
blast int x [
    x = 9999
]
```

The pn will print a 10. blast changes its local copy of x but not the "original" one.

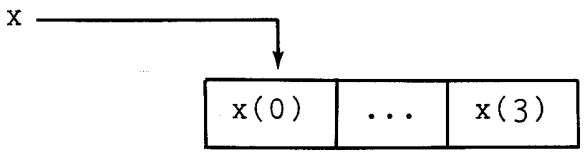
2.6 Pointers

A pointer is a memory address. A pointer variable is a variable whose value is an address. And a pointer expression is an expression whose value is an address.

We have seen that

```
char x(3)
```

declares a list of four character variables. They have names x(0), x(1), x(2), and x(3). In addition, it declares a pointer variable named x, and initializes it with the address of x(0). It is easy to visualize this way:

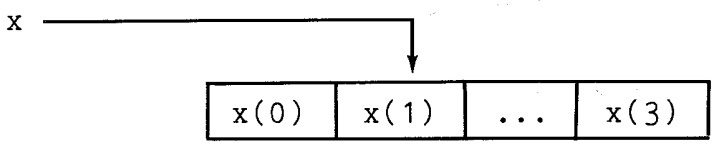


The arrow from x to x(0) indicates that the value in x is the address of x(0).

A pointer expression is a pointer plus or minus an integer expression. A simple case is x+1, which points to x(1). A pointer variable can be assigned a new value. For example,

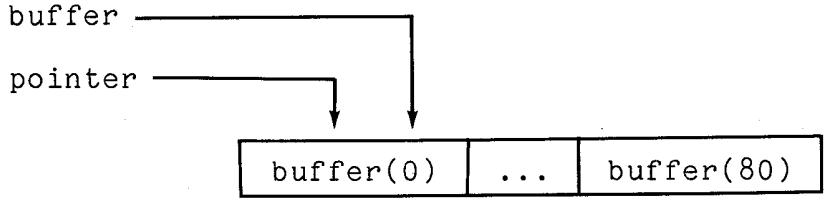
```
x = x+1
```

results in:

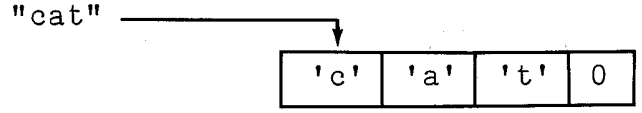


Or:

```
char buffer(80), pointer(0)
pointer = buffer
```



A character string is a pointer to an array initialized with the string and a null byte at the end:



Pointers are frequently used as arguments to functions because they let a called function change variables local to the CALLING function and thus return more than one result. The library function num is a good example. It must return both the number of characters scanned, and the value derived from those characters.

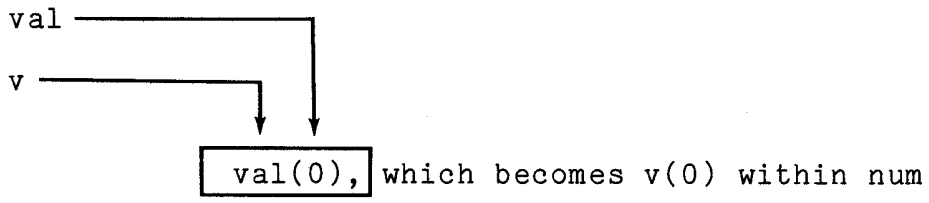
```
num char b(5);int v(0) [
      .
      .
      .
    v(0) = 0
      .
      .
      .
    v(0) = expression
      .
      .
    return k
]
```

The arguments to num are both pointers. Here is a call on num

```
int val(0)
m = num "17", val
```

Notice that the arguments "17" and val are both pointers. val is the interesting one here.

The standard rule for argument passing applies. A copy of the arguments' values is made into the functions' local variables. So we have



The original argument, val, cannot be changed, but the word it points to can be. In fact, v(0) = 0 has the effect of changing val(0).

So the derived value is returned via the pointer v, and the number of characters scanned is returned as the value of num. In effect a call on num says "here are the characters to examine, and here is where to put the value of what you see".

There are other uses for pointers but this is a common one.

2.7 Statements

Simple statements end with a ;, or the end of the line, or the beginning of a remark. A left bracket, [, begins a compound statement. The matching right bracket,], closes it. A right bracket also ends the immediately preceding simple statement.

There are six tiny-c simple statements:

EXPRESSION
The expression is evaluated including all associated assignments and function calls.

if (EXPRESSION) STATEMENT
The statement is executed if and only if the value of the expression is non-zero. The statement can be compound, as in:

```
if (expression) [
   statement
   statement
      "
      "
      "
]
```

if (EXPRESSION) STATEMENT1 else STATEMENT2
Statement1 is executed if and only if the value of the expression is non-zero. Otherwise, statement2 is executed. Either may be compound. Whether compound or not, either can start on a new line. The else can also be on a separate line.

while (EXPRESSION) STATEMENT
The while statement works just like the if statement except the statement part is done repeatedly until the value of the expression becomes zero. The statement may be done as few as zero times, i.e. if the expression is initially zero. As with if and else, the statement may be compound, and can start on a new line.

return EXPRESSION
The expression is optional. If omitted, zero is used. The function containing the RETURN is exited. It is assigned the value of the expression.

break
The innermost while is terminated immediately, regardless of the value of its conditional expression.

2.8 Notes on Using the Statements

Section 2.7 defines the statements completely. Here are some not-so-obvious but frequently used consequences of their definitions.

if and while statements can be nested in any way. It is wise to use a consistent style of indenting when doing so.

Any expression can invoke functions, as described in Section 2.5. So the expression in an if, while, or return can invoke functions.

The statement in an if statement can also be another if statement as in

```
if (x<10)   if (x>7)   a = 1
```

Since the second if statement is a simple statement, no brackets are necessary. The variable a is set equal to 1 exactly when x is less than 10 AND x is greater than 7. Any number of expressions can be "anded" together like this.

The statement2 of an if-else can be another if-else, whose statement2 is another if-else, etc. For example:

```
command [
   char c
   c = gc
   if (c == 'p') print
   else if (c == 'd') delete
   else if (c == 'w') write
   else if (c == 'r') read
   else pl "must be p d w or r"
]
```

Each else follows its respective if. No nested brackets are needed. If, followed by a series of else if pairs, followed by an else, graphically displays an alternation. One and only one alternative is executed.

Always keep in mind that anywhere you can write a statement you can write either a compound statement or any of the six simple statements.

2.9 Libraries and Libraries and Libraries

The sample program in Figure 1-1 uses two functions from the STANDARD LIBRARY, pl and gn. It also uses random, from the OPTIONAL LIBRARY. In Section 1.2.5 we suggested you build a set of useful, general purpose functions, your own PERSONAL LIBRARY. In Section 2.10 we will describe Machine Calls, from which two additional libraries can be formed: STANDARD MACHINE CALLS, and PRIVATE MACHINE CALLS.

What is a library and what distinguishes one library from another? A library is simply a collection of similar functions, and libraries differ from one another on the basis of what the functions contained within them have in common. The five libraries that have been mentioned so far could be briefly defined as follows:

standard library -- tiny-c functions used by the PPS and useful to all programs developed with the PPS.

optional library -- tiny-c functions generally useful to tiny-c programs but not required frequently enough to be included in the standard library.

personal library -- tiny-c functions frequently used at a particular computer installation, or by a specific class of application program.

standard machine calls -- functions which are used so frequently by all tiny-c programs that they have been implemented in machine language to improve processing speed.

private machine calls -- functions which are so frequently used by tiny-c programs at a particular computer installation, or by a specific class of application programs that they have been implemented in machine language to improve processing speed.

The spirit of the standard, optional, and standard machine call libraries is that they have the same definition at each tiny-c installation so that programs developed at one can be run at another. They are, in a sense, extensions of the tiny-c language.

The STANDARD LIBRARY is a set of functions that is loaded with and used by PPS. As a result, these functions are accessible to all programs developed with PPS. They need not be defined or specifically loaded to be used. They are defined in Section 2.9.1 below.

The OPTIONAL LIBRARY is a set of tiny-c functions frequently useful to, but not always required by, a project. random is one of these functions. They are defined in Section 4.1. They are not loaded with PPS, so whenever they are used they must be specifically loaded. In principle, the optional library is a large collection of tiny-c program tools.

Your PERSONAL LIBRARY is your own extension to the optional library.

MACHINE CALLS are coded in machine language. There is a standard set furnished with tiny-c (Section 5.10); and you

can build your own private one (Section 2.10). These are used for speed, or to interface with special input/output devices.

Machine calls have an awkward, undescriptive syntax. For example, it isn't immediately obvious what

```
MC 47,64,1,1001
```

means or does. It is customary to wrap a machine call up with a nice name, like this:

```
plot int r, c,nf [
   return MC r, c, nf, 1001
]
```

Now, when you write programs (especially for publication) you can use

```
plot 47,64,1
```

Although we haven't defined the plot function yet, "plot" is certainly somewhat more suggestive of what it does than "MC 1001".

2.9.1 Standard Library

The standard library includes functions that do input, output, and character manipulation. The definitions given here show the declaration of the function name and the arguments, if any.

gs char buffer(0)
: Reads a line, i.e., a string of characters terminated by a carriage return, from the terminal and puts it in buffer. The carriage return at the end of the line is changed to a null byte. The value of the function is the number of characters placed in the buffer excluding the null byte. A value of 0 is permitted.

ps char buffer(0)
Prints the string in buffer on the terminal. A null byte signals the end of the string. The null is not transmitted. The number of characters transmitted is returned as the value of the function.

pl char buffer(0)
The same as ps but prints the string on a new line. The number of characters transmitted, not including the leading return and line feed, is returned.

pn int n
Prints on the terminal an integer preceded by a blank. The number of characters transmitted, including the blank, is returned.

gn
Reads a line, and returns the integer at the beginning of the line. If there is no integer there, it prints "number required" and tries again.

gc
Reads a line, and returns the first character on the line.

putchar char c
Transmits the character c to the terminal. Any character, including control characters, can be transmitted, except that if c is null a quote is transmitted. The character c is returned.

getchar
Reads and returns a character from the terminal. Any character, including control characters, can be read by this function.

readfile char name(0), where(0), limit(0)
int unit
Reads data from a file. name is a character string terminated by a null byte. where and limit are pointers. unit is an input/output unit (or device or channel). The file with name "name" is opened for reading on device "unit". All its records are read and placed in sequentially higher addresses starting at where, but in no case going beyond limit. Then unit is closed. If successful, the total number of bytes read is returned.

If limit is exceeded, the message "too big" is printed and -2 is returned. If any other problem occurs, installation-dependent messages may be printed, and a negative value is returned.

writefile char name(0), from(0), to(0)
int unit
Writes data to a file. name is a character string terminated by a null byte. from and to are pointers. unit is an input/output unit (or device or channel). writefile opens unit "unit" for output. The contents of sequentially higher addresses from "from" to "to" inclusive are written to unit as a file named "name". Then unit is closed. If successful, the total number of bytes written is returned. If a problem occurs, installation-dependent messages may be written, and a negative value is returned.

num char b(5)
int v(0)
Converts a string of digits without leading sign or blanks to the corresponding numeric value which is put in v(0). The first non-digit stops the conversion. At most, 5 digits are examined. The number of bytes converted is returned as the value of the function. Note that the second argument must be a pointer to an integer.

atoi char b(0)
int v(0)
Converts a character string of the form: 0 or more blanks, optional plus or minus sign, 0 or more blanks, 0 to 5 digits, to its numeric value which is put in v(0). The first non-digit following the digit part stops the conversion. The number of characters in b that were used to form the value is returned as the value of the function.

ceqn char a(0), b(0)
int n
Compares two character strings for equality for n characters. Returns 1 on equals, 0 on not equals.

alpha char c
Returns a 1 if c is an alphabetic character, upper or lower case. Otherwise returns a 0.

```
index char s1(0)
      int n1
      char s2(0)
      int n2
```
Finds the leftmost copy OF the character string s1 which is n1 bytes long IN the character string s2 which is n2 bytes long. If s1 does not appear in s2, 0 is returned. If s1 does appear, return n+1 such that s2+n points to the first character of the copy in s2.

```
move char a(0), b(0)
```
Moves string a into b up to and including the null byte of a.

```
movebl char a(0), b(0)
       int k
```
Moves a block of storage up or down k bytes in memory. a and b point to the first and last characters of the block to be moved. If k is positive the move is to higher addresses, and if it is negative the move is to lower addresses. If k is positive, the byte at b is moved first, then the byte at b-1, etc. If k is negative the byte at a is moved first. Thus large blocks can be moved a few bytes without destruction.

```
countch char a(0), b(0), c
```
Counts the instances of the character c in the block of storage from a to b inclusive and returns the count.

```
scann char from(0), to(0), c
      int n(0)
```
Scans from "from" to "to" inclusive for instances of the character c. The integer n(0) is decremented for each c found. If n(0) reaches 0, or if the character in to(0) is examined, scann stops. scann returns the offset relative to the pointer, from, to the last examined character. Thus, if the third character position after from is the last examined, scann returns 3.

```
chrdy
```
Returns a copy of an input character from the terminal if a character has been typed but not yet read by another function, except that if the typed character is a null, a 1 is returned. If no unread character has been typed, a null byte is returned. The character is not cleared so a subsequent call to getchar or gc will return the same character.

```
pft char a(0), b(0)
```
Transfers all characters from a to b inclusive to the console terminal.

```
fopen int rw
      char name(0)
      int size, unit
```
Opens or creates a file for access on logical unit "unit". "name" contains a string, null terminated, giving the name of the file. There may be installation restrictions on file names. The file is opened for reading if rw is 1, and writing if rw is 2. If rw is 2 and the file does not exist, then it will be created and its size guaranteed to be at least "size" bytes. Otherwise size is ignored, but must be given. (Use a 0.) For a tape system, and some disk systems, rw and size may both be ignored, but they must be given nonetheless. If no error is detected, a 0 is returned. If an error is detected a nonzero is returned.

```
fread char a(0)
      int unit
```
Starting at a, reads into memory the next record of data from the file opened on "unit". The array a must be large enough to hold the largest expected record. The length in bytes of the record is returned as the value of fread. Note that the installation may place an upper bound on record lengths. A -1 is returned if an end-of-file is detected, i.e., if an attempt is made to read beyond the last record in the file. A larger negative number is returned if an error is detected.

```
fwrite char from(0), to (0)
       int unit
```
Writes one record with the bytes from "from" to "to" inclusive to the file opened on unit "unit". This becomes the next record of the file. Its length is to-from+1, and this is the length that will be returned when the record is read by fread. Note that the installation may place an upper bound on record lengths.

```
fclose int unit
```
The file opened on "unit" is made permanent, and arrangements are made for end-of-file detection by fread.

2.9.2 Notes on Using the Standard Library Functions

Note that the standard library has three functions to read characters. getchar and gc each read one character. getchar is the fundamental version. It:

1. Stops the program.
2. Waits for ONE CHARACTER to be typed.
3. Starts up again.
4. Returns the character.

getchar is useful for one letter commands. gc is oriented toward the input of full lines. It:

1. Stops the program.
2. Waits for a WHOLE LINE to be typed (ending with a carriage return).
3. Starts up again.
4. Returns the first character of the line.
5. Throws the rest of the line away.

gc is for one-letter answers to questions in a question and answer dialogue. A very common use is to test for the 'y' of a yes answer:

```
ps "Do you want to play again?"
if (gc() == 'y') [
     .
     .
     .
```

The third function, gs, reads a character string. Since gs is defined as

```
gs char buffer(0)
```

it must be called as

```
gs pointer-expression-to-character-data
```

For example,

```
char x(80)
gs x
```

will read a character string from the console terminal into x(0), x(1), ... If you wrote

```
gs x+10
```

the string would be read into x(10), x(11), ... Note that a quoted string is really a pointer. That's why

```
ps "hello"
```

works. The last quote of the string is replaced by a null within tiny-c.

Pointers are also used in standard library functions to return two or more results. num is an example. It returns the number of characters scanned. But it changes the integer v(0). Thus, num must be called like this:

```
int v(0), k
k = num "17", v
```

There is no subscript on v in the call. This call will put 17 into v(0), and 2 into k. atoi and scann also use this method.

atoi is just like num, except it handles a sign and leading blanks. num will NOT skip leading blanks.

ceqn is the standard way to compare two character strings since this type of comparison cannot be done with a tiny-c expression. For example, the following will NOT determine if "cat" has been entered at the console:

```
char x(10)
gs x
if (x == "cat") ...  /*WRONG
```

The if will compare the pointer x with the address where "cat" is stored, and will always be false. To test if "cat" has been entered, use ceqn:

```
char x(10)
gs x
if (ceqn(x,"cat",3)) ... /*RIGHT
```

See Chapter IV for another match function, ceq.

movebl, countch, and scann are implemented in assembly language, and are, therefore, quite fast. index is implemented partly in assembly language and partly in

tiny-c. scann is intended to scan quickly for the nth occurrence of a character in an array. Here's how to use it.

```
int n(0),
char x(100), where
n(0) = 7
where = scann(x,x+100,' ',n)
```

This call to scann scans for the 7th blank in x. x+where will point to the 7th, and n(0) will be zero if there are at least 7 blanks. where will be 100 (the address relative to x of the last character examined) and n(0) will be 7 minus the number of blanks in x if there are less than 7 blanks. So testing n is a convenient way to see why scann stopped.

There are two facilities for manipulating data files. The simplest are the readfile and writefile functions. Whole arrays are read or written as whole files. Slightly more complex, but more versatile, are the fopen, fread, fwrite, fclose functions. These can access large files a record at a time.

For either facility unit 1 is guaranteed to be available on all installations. Other units are available only on multiple-unit installations. Note that a unit is a logical concept. For example, accessing unit 1 does not mean accessing drive 1 of a multidrive disk system. How units map onto devices is determined by the installation. But generally they will be small positive integers, e.g., units 1 through 4 on a four-unit installation.

Note that this specification leaves much to the installation. tiny-c does not check that any of these limits are exceeded. Nothing is said in the specification about what happens when limits are exceeded. So if you write more than "size" bytes to a newly created file, you may abort, have your writes ignored, clobber an adjacent file, or your file may have its size extended and the writes will actually "take". It depends on the installation.

There are two points-of-view to take on exceeding the defined limitations of this package:

> PRIVATE VIEW: Learn what your installation does when a limit is exceeded. If it is reasonable and useful, use the capability. But don't publish the program without warning others of what you have done.

> PORTABILITY VIEW: Don't exceed the limits. Then you have a portable program that runs on anybody's installation of tiny-c.

Both views are valid, depending on your objectives. Clearly, PPS has adopted the portability view.

Among other limits is what happens if to-from+1 exceeds the installation record limit. As of this writing one installation truncates the excess, and writes one full record, whereas another writes multiple records so that all the data gets written. Each is reasonable and useful in the private view. In the portability view both installations do the same thing. In Section 4.3 the function writefile guarantees a limit of 256 bytes per record. So it is portable. Other "limits" are what happens if you read and write records to the same file, open the same file on two units, open the same unit on two files, etc. When adopting the portability view, don't do any of these things.

2.10 Machine Language Interface

There are several reasons why you may want to use your own machine language code for parts of a project. Usually this is done for execution speed, or to access new devices. If you write a machine language subroutine, and want to call it from a tiny-c program, the mechanism for doing so is the machine-call (MC) function. MC is passed arguments and returns a value. An argument can be an arbitrary expression. So, for example, you can write:

```
k = MC row-1, 2*col+6, 1, 1001
```

This passes four arguments to MC. The returned value is assigned to k. An MC can be used in an if statement, or an expression, or anywhere a function can be used:

```
if (MC(12)=='x') gotcha (MC(2))
```

This calls MC with the argument 12. If it returns the value 'x', then MC is called with the argument 2. The result returned is used as the argument in a call to gotcha.

The MC function differs from other tiny-c functions in three ways:

1. Its name, MC, is built into tiny-c.

2. It can have a variable number of arguments, but must have at least one. (Other functions must have exactly the number of arguments specified in their definitions, and can have none.)

3. It is coded in machine language, not in tiny-c.

The LAST argument determines which particular machine-coded function is to be executed. This argument is called the FUNCTION NUMBER.

Every function is assigned a unique number. Those furnished as "standard" MCs have numbers from 1 to 999. Those you write for your own local use can be assigned numbers from 1000 to 32767. This number assignment system guarantees that a future release of tiny-c won't have new function numbers that conflict with your own local ones.

An MC is invoked by tiny-c as follows:

1. The arguments are evaluated left to right, and their values pushed on a tiny-c stack. (This is not the processor stack, but a software implemented stack with special features.)

2. The last argument is the function number. It is popped from the tiny-c stack and examined. If it is less than 1000, the appropriate "standard" MC is executed.

3. If the function number exceeds 999, then 1000 is subtracted from it and a subroutine call is made to USERMC in the installation vector. You must place a jump instruction there to your MC code.

Note: In the 8080 version, this number is left in HL. In the PDP-11 version it is at 2(SP).

Now your MC has control. There are rules and tools for writing an MC:

1. You must write code to examine the adjusted function number and branch to your appropriate function. When done, a return is used to return control to tiny-c.

2. You must use all the arguments given.

3. You must return a result.

4. You cannot modify any standard cell used by tiny-c in its internal operation.

5. You must arrange for your MC code to be loaded, and the jump at USERMC must have the address where it receives control.

6. You must also arrange the four tiny-c data areas so they do not conflict with where you loaded your MC code. (See Section 6.5.2.)

Step 1 -- is easily done with 8080 code like this:

```
USERMC    JMP    MYMCS
            .
            .
            .
MYMCS     MOV    A,L          ;branch to one of
          CPI    1            ;two user MCs.
          JZ     MC1001
          CPI    2
          JZ     MC1002
MCERR     JMP    MCESET
```

Jumping to MCESET signals to tiny-c that an error was detected in an MC, in this case an invalid function number. MCESET is one of the MC writing tools. It can be used for other MC-detected errors.

In PDP-11 code the beginning of user MCs might look like this:

```
USERMC    JMP    MYMCS
            .
            .
            .
MYMCS     CMP    #1,2(SP)
          BEQ    MC1001
          CMP    #2,2(SP)
          BEQ    MC1002
          MOV    #MCERR,-(SP)
          JSR    PC,@#ESET
          TST    (SP)+
          RTS    PC
```

Step 2 (8080) -- You can "use" an argument by calling the subroutine TOPTOI. (WARNING: This will modify all registers!) The value on the top of the stack is returned in DE, or just E if it is a one-byte value, and the stack is popped. Calling TOPTOI three times retrieves three arguments. Note that they are retrieved RIGHT to LEFT as they appear in the MC call. Note also that the function number was already retrieved (popped) within tiny-c, so the first call gets the next-to-last argument. Don't call TOPTOI too often. There is no way to reassemble the stack the way it was, should you do so. Don't call it too few times. This leaves garbage on the stack, and the rest of your tiny-c program will get truly sick. What will probably happen is an arguments error for some innocent tiny-c function called later on. To help, the byte called MCARGS contains the number of arguments given by the call to the MC, including the function number. You can use this for checking, or for an MC with a variable number of arguments.

Step 2 (PDP-11) -- You can "use" an argument by calling TOPTOI. The value on the top of the stack is returned in R0 and the stack is popped. The arguments are retrieved RIGHT to LEFT as they appear in the MC call. The total number of arguments in the MC call is in 4(SP).

Step 3 (8080) -- To return a result, put a two-byte value in DE, and call PUSHK. This must be done ONCE in an MC before returning. Failure will probably cause an arguments error as described in Step 2.

Step 3 (PDP-11) -- To return a result, put the value to be returned in R0 and execute the following code:

```
MOV     R0,(SP)
MOV     #1,-(SP)
MOV     #101,-(SP)
CLR     -(SP)
JSR     PC,@#PUSH
ADD     #6,SP
```

This must be done ONCE in an MC before returning.

Step 4 -- Well, it's obvious you can cream tiny-c from a machine-coded subroutine. Just be careful.

Steps 5 and 6 -- How you assemble and load your code depends on your operating system. Be sure there is no conflict with other uses of memory. Section 6.5.2 describes how memory is allocated for tiny-c and includes recommendations for the placement of user MC code. Study this, and make adjustments to your memory allocation addresses so that your MC machine code doesn't overlap a tiny-c data area. Here's where some helpful jump addresses are located in 8080 tiny-c:

USERMC	ORG + 1F
MYMCS	determined by where User Machine Call program is loaded.
MCESET	ORG + 2B
TOPTOI	ORG + 2E
PUSHK	ORG + 31
MCARGS	ORG + 34

Section 6.2.2 defines ORG. The offsets are in hex.

Those are the basic steps to follow. Sample MCs (the 14 built-ins) are given in the listings and definitions are given in Section 5.10. These can be used as examples for building your own MCs.

2.11 Computer Arithmetic

All computers have limits on how large a number can be handled. When the limits are exceeded the number is said to OVERFLOW.

For both the 8080 and the PDP-11 versions of tiny-c, the numbers must be in the range

```
-32768 <= number <= 32767.
```

When a calculation would be outside this range, this is how to determine what happens. Subtract (or add if the calculation is too negative) 65536 repeatedly from the result until it does lie in the correct range. That is the answer.

The example in Section 1.1 is

```
last = last * seed
     = 9801 * 99
     = 970299
```

which is outside the range. Subtracting 65536 fifteen times gives the "correct" (sic) answer, i.e. the one returned by the computer:

```
     = 970299 - 15 * 65536
     = -12741
```

III. THE PROGRAM PREPARATION SYSTEM (PPS)

The Program Preparation System (PPS) is a tiny-c program that lets you type in, edit, run, and write a program on a cassette or floppy disk, and read it back later for more runs and/or edits.

3.1 Fundamentals of PPS

A PPS session begins by reading PPS and starting it. (Refer to the installation chapter for your particular system to find out how to accomplish this.) PPS prints:

```
>
```

indicating it is ready for input. You can now type lines of your program, or give commands for PPS to execute.

Any line you type must end with a carriage return; PPS does nothing with your input line until the carriage return is given.

After the carriage return, PPS will either enter the line into your program, or execute the command. Then it gives another

```
>
```

and awaits another line or command.

If you mistype before giving a carriage return, the ASCII DEL character "kills" the most recently typed character. You can enter it several times to kill several characters. [Note: if DEL is unsuitable for your terminal, or your editing habits, the appropriate installation chapter describes how to modify this to a character of your choice.]

If you mistype a line so hopelessly that you want to do the whole line over, then the ASCII CAN in tiny-c/8080 or NAK in tiny-c/11 "kills" the whole line. CAN is control-X on most keyboards, while NAK is control-U. It must be given before the carriage return. You CANNOT give it several times to kill several lines. It will kill only the line which you have started, for which you have not yet given a carriage return.

If you have typed a carriage return and still want to make a change, this can be done. You must explicitly delete the line, using the delete command described in Section 3.3. Then you can re-enter the line.

DO NOT type in line numbers at the beginning of each line. tiny-c does not use them, in fact, does not even tolerate them.

To indent, use tabs or spaces. The use of indentation to show the number of logical conditions in effect at each point in a program can greatly enhance its readability.

Now, what is a command, and what is a line of text?

```
A COMMAND ALWAYS BEGINS
WITH A PERIOD, A PLUS, OR
A MINUS.  ANYTHING ELSE
IS A TEXT LINE.
```

We will cover the PPS commands later; first, we will go over text lines. To do this we need the concepts of PROGRAM BUFFER, LINE ZERO, and CURRENT LINE.

As you enter program lines, they go into a character array called the PROGRAM BUFFER. Think of the program buffer as a series of text lines. If you enter a text line, it goes into the program buffer. It may be either added at the end of, or inserted between, lines already in the buffer.

Initially the program buffer has only one line, called LINE ZERO. Line zero has no text, just a carriage return. No matter what else you put in the program buffer, line zero is always there, and is always just a carriage return.

One line in the program buffer is always the CURRENT LINE. Initially it is line zero. You can always display the

current line by giving the print command:

>.p

(The ">" is the prompter printed by PPS. The ".p" is the print command typed in by the user.)

3.2 Entering Text Lines

Now, where do new text lines go?

A TEXT LINE IS ENTERED AFTER THE CURRENT LINE. THE NEWLY ENTERED LINE BECOMES THE CURRENT LINE.

Initially the zero line is current. You type a text line. It goes into the program buffer as line 1, and becomes the current line. You type a second text line. It goes in after the current line (i.e., line 1), and becomes line 2; now line 2 is current. So if all you do is enter text lines, they each go into the program buffer one after the other.

There are commands (described below) to make current any line in the buffer. Whenever you enter a series of text lines, each is inserted one after the other below the current line. So you can enter text lines anywhere in the program buffer. You will discover that this text-line-entry rule is simple, natural, and powerful.

3.3 The PPS Commands

In the commands below, the "n" represents an unsigned (no + or -) integer. Exactly one blank must separate an integer n from preceding characters.

>.p	Print the current line.
>.p n	Print n lines, starting with the current line. The last line printed becomes current.

>.d Delete the current line. Make the line BEFORE it current.

>.d n Delete n lines, starting with the current line. Make the line BEFORE the first deleted line current.

>+ Move down one line; i.e., make the line after the "present" current line, the "new" current line.

>+n Move down n lines.

>- Move up one line.

>-n Move up n lines.

>.n Make the n-th line in the program buffer the current line.

>.l text Starting with the line AFTER the current line, and proceeding to the end of the program buffer if necessary, locate a line containing "text". If found, print the line, and make it current. If not found, print "?", and leave the current line unchanged. There must be one blank between the "l" and the first character of text. A "^" (ASCII octal code 136) as the first character of text means the text must begin the line. A "^" as the last character of text means the text must end the line. Text may contain blanks.

>.l Same as above, but using the same text as given in a previous locate or change command.

>.c text newtext

In the current line, the first occurrence of text is replaced by newtext. If text does not occur in the current line, no change is made. In either case the resulting line is printed. As shown, there are exactly two blanks in the command, one after the c, and one between text and newtext. In this form, no blanks can be used in text or newtext, because a blank is the delimiter that separates them. However, since any punctuation character can be used as the delimiter, the command can be given as:

>.c/text/newtext

In this case, blanks can be used in the texts, but /'s may not be used. An optional delimiter is permitted at the end of newtext:

>.c/text/newtext/

Either text or newtext can be empty.

>.c//newtext/

will insert newtext at the beginning of the line, whereas:

>.c/text//

will erase text from the line. Finally, when making a series of identical changes, you need not retype the texts over and over:

>.c Makes a change on the current line using text given in the most recent change or locate command, and newtext given in the most recent change command. The carriage return must be immediately after the c.

>./ Prints the current line number, the total number of lines, the total number of characters used in the program buffer, and the total number of characters unused.

>.r filename

Reads a file from cassette or floppy disk, putting what is read AFTER the last line. The current line is not changed. [Note: Some installations of tiny-c may not use the filename.]

>.w filename

Writes all lines to a cassette or disk, giving the name "filename" to what is written. [Note: Some installations may not use the filename.]

A command line starting with a period and at least two alphabetic characters is executed immediately as a tiny-c statement. This is how programs are started. Thus, to run the "guessnum" program in Figure 1-1:

```
>.guessnum
```

Arguments can be given. To add 7 and 11 and print the answer, call the pn library function:

```
>.pn 7+11
```

A compound statement can also be given, but it must fit on one line (64 characters including the period and the carriage return.)

```
>.[char a; while ((a=a+1)<=127) putchar c ]
```

Machine calls can be directly executed:

```
>.MC 24,64,1,1001
```

When a program is running, it can be halted and control returned to PPS by typing the ASCII ESC key. (Note: if ESC is unsuitable for your terminal, it can be changed to another character. See Section 6.5.4.2).

3.4 Notes on Using PPS

3.4.1 Bumping the Top and Bottom

Several commands can "bump into" the top or bottom of the program buffer. This is all right, and in fact, can be useful. For example, suppose there are 50 or so lines in the buffer. Then

```
>.0
>.p 999
```

makes line zero current, then prints the whole buffer. The

.p 999 "bumps into" the bottom of the buffer, and stops. The commands .d, +, -, .n, and .l can also bump into the top or bottom. A convenient way to go to the last line is:

```
>.999
```

3.4.2 Deleting

Line zero cannot be deleted. To delete lines at the top, go to line 1, then give the appropriate delete.

Notice that delete moves the current line up, not down. Thus any new lines typed after a delete will replace the deleted lines.

3.4.3 Line Numbers

When lines are inserted or deleted, lines further down in the buffer immediately have different line numbers. So .n is not the best way to locate a line. For example, in Figure 1-1, the command:

```
>.22
```

makes the first line of random current. But if edits are made to guessnum, then the 22nd line may or may not be the first line of random. For this reason, locate is a more powerful tool.

3.4.4 Using Locate and Change

The ^ convention in locate text makes it easy to locate the beginning of a function. To locate the random function:

```
>.l ^random
```

A match occurs only if the text "random" is at the left margin of the page. So in Figure 1-1, line 6 will not match, but line 22 will match.

Locating all lines with a given text is done like this:

```
>.0
>.1 random
```

This will match line 6 in the sample program. Then:

```
>.1
```

matches line 19, the comment containing the word "random". The following .1s match line 19, line 21, and line 22, while the final .1 prints "?" indicating no further appearance of "random".

Making a common change throughout the buffer is just as easy. To change the variable named "number" to one named "num", type:

```
>.0
>.1 number
```

This will make line 1 current. It is a comment so we do not want to change this occurence of "number". Type:

```
>.1
```

and line 5 becomes current. We do want to change "number" here, so type:

```
>.c number num
```

Line 5 is changed. Continue with:

```
>.1
```

which makes line 6 current. Change it by typing:

```
>.c
```

and resume with:

```
>.1
```

You continue in this fashion, changing lines selectively, until you bump the bottom.

3.5 Errors

When a program error is detected, the program halts. A return is made to the system. Your program text is intact, and you can edit it, or restart it, or write it to a cassette. It prints three lines, as shown:

```
17  -- err 26
text of bad line
      <
```

The first line shows the line number, and the error number. The second line is the text of the bad line. Immediately above or to the left of the < is where the problem was DETECTED. This may or may not be the real problem, depending on the logic of the program.

Upon halting, the current line is the line printed.

The error numbers and their meanings are:

1 Illegal statement
2 Cursor ran off end of program. Look for missing] or)
3 Symbol error. A name was expected. For example 10 + + will cause this.
5 Right parenthesis missing, as in: x = (x+a*b
6 Subscript out of range
7 Using a pointer as a variable or vice versa
9 More expression expected, as in: x = x +
14 Illegal equal sign, as in: 7=2
16 Stack overflow. Either an expression is too tough, or you are deeply nested in functions, or a recursion has gone too deep.
17 Too many active functions
18 Too many active variables
19 Too many active values. Values share space with program text. Crunch the program and this error may go away. (Remove remarks and unnecessary blanks, and shorten variable names.) Or settle for fewer features, or buy more memory.
20 Startup error. Caused by a "garbage" line outside of all [ ], i.e., where globals are declared. A missing [or] can cause this.
21 Number of arguments needed and number given don't agree
22 A function body must begin with [.
24 An illegal invocation of MC
26 Undefined symbol. Perhaps name is misspelled, or you need an int or char statement for it, or the function isn't loaded.

3.6 Sample Session with PPS

```
>./
 Ø Ø Ø
>/* Guess a number between 1 andq1ØØ
>.c/q/ /
/* Guess a number between 1 and 1ØØ
>/* T. A. Gibson, 11/29/76
>guessnum[
>  int guess, number
>  number=random(1,1ØØ)
>  pl "guess a number between 1 and 1ØØ"
>  pl "tyype in your guess now"
>  while(guess != number) [
>    guess = gn
>    if(guess == number)pl "right!"
>    if(guess > number)plq"too high"
>    if(guess < number)pl "too low"
>    pl"";pl""
>  ]       /* end of game loop
>]         /* end of program
>./
 15 15 4Ø5 4595
>.1
/* guess a number between 1 and 1ØØ
>.p 99
/* guess a number between 1 and 1ØØ
/* T. A. Gibson, 11/29/76
guessnum[
  int guess, number
  number=random(1,1ØØ)
  pl "guess a number between 1 and 1ØØ"
  pl "tyype in your guess now"
  while(guess != number)[
    guess = gn
    if(guess == number)pl "right!"
    if(guess > number)plq"too high"
    if(guess < number)pl "too low"
    pl"";pl""
  ]       /* end of game loop
]         /* end of program
```

```
>.r random
 247
 15 27 652 4382
>.p
]         /* end of program
>+
/* random -- generates a random number between little
>.p 99
/* random -- generates a random number between little
/*           and big
random int little,big [
  int range
  if(last==Ø)last=seed=99
  range=big-little+1
  last=last*seed
  if(last<Ø)last=-last
  return little + (last/8)%range
]
int seed,last
>.guessnum

guess a number between 1 and 1ØØ
tyype in your guess nowSØ


 11 --- err 26
    if(guess > number)plq"too high"
                           <
>.c/q/ /
    if(guess > number)pl "too high"
>.Ø

>.1 yy
  pl "tyype in your guess now"
>.c/yy/y/
  pl "type in your guess now"
>.guessnum

guess a number between 1 and 1ØØ
type in your guess nowSØ

too high

25
```

```
too low

37

too high

27

too high

26

right!


>.w guess
 3 27 651 4349
   651
>
```

IV. tiny-c PROGRAM EXAMPLES

Since we all know the value of pictures versus words, this chapter is devoted to tiny-c program examples. Section 4.1 contains some software tools which are candidates for inclusion in the optional library. Section 4.2 contains a complete original computer game called Piranha Fish. The tiny-c owner initially interacts most with the PPS; thus, Section 4.3 provides a readable and fully commented version of PPS together with the standard library. Sections 4.4, 4.5 and 4.6 show how tiny-c can be interfaced with specific hardware devices.

Programming can best be learned by reading programs. Such reading helps you learn style, idiomatic usages, and, in general, get an appreciation of the possibilities of a language.

The sample programs included here are intended not only to be useful, but also to be read. Therefore (wherever appropriate) we have commented on their style.

4.1 Optional Library Functions

These routines complement those in Section 2.9. We give their definitions, then the code, then a few comments on the programming style.

random int little,big
: A pseudo-random number between little and big (inclusive) is generated. little cannot be larger than big, and their difference should not be larger than 4096. The global integers seed and last are part of random. These can be initialized to any value. Their value determines the sequence of numbers generated.

```
htoi char b(0)
     int val(0)
```
Converts a string of hex digits to an integer. The hex digits may be preceded by blanks. The value of the first non-hex digit (except for leading blanks) stops the conversion. The integer is put in val(0), and the number of characters scanned in b, including blanks, is returned.

```
blanks char b(0)
```
Counts the number of leading blanks in the string b, and returns the count.

```
ceq char a(0), b(0)
```
Matches the two strings a and b up to but not including a null byte in a. 0 is returned on mismatch, 1 on match. Thus, a must be a leading substring of b to get a match.

```
char b(0)
ceq b, "yes"
```

returns 1 if b has "y", "ye", or "yes".

```
itoh int n
     char b(4)
```
The integer n is converted to four hex digits plus a null byte, which are put in b.

```
itoa int n
     char b(7)
```
The integer n is converted to an ASCII representation of the integer: a minus (-) if needed, followed by 1 to 5 digits followed by a null byte. The string is put into b. The number of bytes of the string (excluding the null) is returned.

```
moven char a(0), b(0)
      int n
```
n bytes are moved from a to b.

4.1.1 tiny-c Code for the Optional Library

random is shown in Section 1.1 (Figure 1-1). The other functions are given here.

FIGURE 4-1

```
/* Converts hex to integer.  Returns result in val(Ø).
/* Returns number of characters scanned as value of htoi.
/* First non-hex character stops the scan.
htoi char b(Ø)
     int val(Ø) [
  int n   /* Number of chars scanned.
  b=b+(n=blanks(b))  /* Skip blanks.  Set b to first nonblank.
                     /*   Set n to number of blanks skipped.
  val(Ø)=Ø
  while(1) [
    if(b(Ø)<'Ø')break
    else if(b(Ø)<='9')val(Ø)=16*val(Ø)+b(Ø)-'Ø'
    else if(b(Ø)<'A')break
    else if(b(Ø)<='F')val(Ø)=16*val(Ø)+b(Ø)-'7'
    else break
    b=b+1
    n=n+1
  ]
  return n
]
/* Counts leading blanks in b.
blanks char b(Ø) [
  int n
  while(b(n)==' ')n=n+1
  return n
]
/* Tests if a is a leading substring of b.  Returns 1 on
/*   true, Ø on false.
ceq char a(Ø), b(Ø) [
  while(a(Ø) != Ø) [
    if(a(Ø) != b(Ø)) return Ø
    a=a+1; b=b+1
  ]
  return 1
]
/* Converts integer to hex.
itoh int n
     char b(4) [
  int k
  b(k=4)=Ø
```

```
  while((k=k-1)>=Ø) [
    b(k)=n%16+'Ø'
    if(b(k)>'9')b(k)=b(k)+7
    n=n/16
  ]
]
/* Converts binary integer to ASCII.
itoa int n
     char b(7) [
  if(n<Ø) [
    b(Ø)='-'
    return 1+itoa(-n,b+1)
  ]
  if(n<1Ø) [
    b(Ø)=n+'Ø'
    return 1
  ]
  int k
  b(k=itoa(n/1Ø,b))=n%1Ø+'Ø'
  b(k+1)=Ø
  return k+1
]
/* Move n bytes from a to b.
moven char a(Ø), b(Ø)
      int n [
  if(n)movebl(a,a+n-1,b-a)
]
>
```

End of FIGURE 4-1

4.1.2 Comments on Style

Most of the code is straightforward. ceq is interesting because it increments the pointers a and b, instead of declaring an integer subscript and incrementing the subscript. Of course, only the local copy is being incremented. The pointers passed as arguments into ceq are not modified. This follows the general rules discussed in Sections 2.5 and 2.6.

itoh knows it's going to derive four hex digits. It gets the last one first, and puts it into b(3), and then works towards b(0).

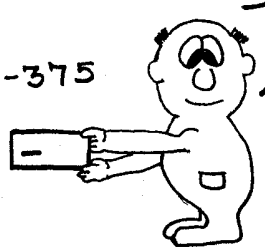
① Hi! I'm I to A, pronounced eye-to-A. I convert an integer to ascii character. I recursively use myself to do my job.

And so I give my caller these 5 bytes:

-375 null

⑨

② Give me -375 and a buffer (a character array). I put - in the first byte of the buffer and hand off 375 and the buffer offset by one ...

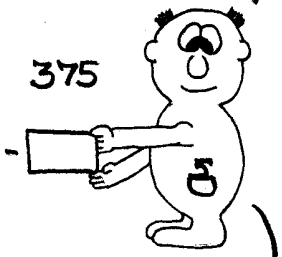


... byte to I to A to finish.

Of course I'm the only one that knows the buffer really starts one byte earlier and has a - in it. I put a null byte at the end.

⑧

③ Give me 375 and a buffer. I keep the 5 in my pocket, and hand off 37 and the buffer to I to A.

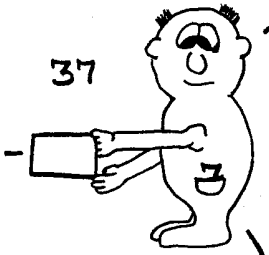


Now I stick my 5 at the end, and return 3 to signal that the buffer has 3 entries.

-375

⑦

④ Give me 37 and a buffer. I keep the 7 in my pocket, and hand off 3 and the buffer to I to A

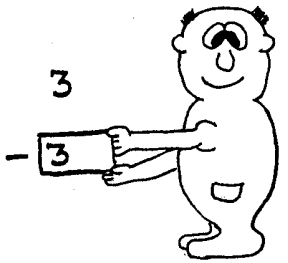


Now I stick my 7 at the end, and return 2 to signal that the buffer has 2 entries.

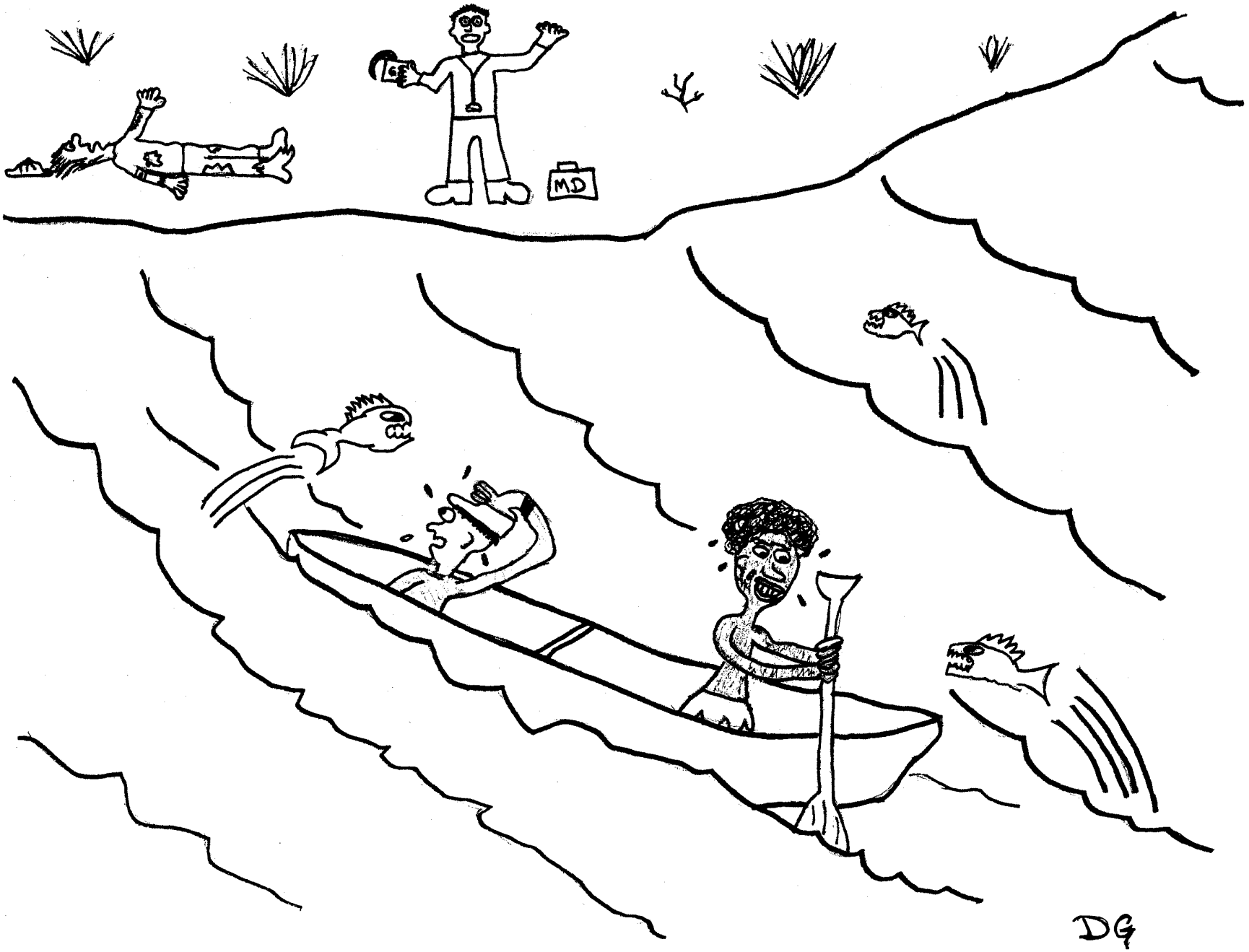
-37

⑥

⑤ Give me 3 and a buffer. This is only one digit, so I put it in the buffer and return 1 to signal that the buffer has one entry.



DL


MD
DG

itoa also derives its last digit first, but it does not know in advance how long the string will be and hence where to put the first digit. There are several ways to handle this problem:

1. A series of if statements on n, e.g., n<9999, n<999, n<99, n<9, could be used to compute the size of the output string in advance. Then the technique for itoh can be used.

2. The bytes can be put into b(0), b(1), ... in that order, then reversed.

3. The bytes can be put into b(6), b(5), ... in that order, then moved left if needed.

4. Recursion can be used to get everything into place directly, with no size computation required.

itoa uses the last technique. It is a good example of recursion. The illustration on the facing page describes better than words how it works.

4.2 Piranha Fish -- An Original Game

You are leading the following party on a safari through the jungle:

2 cannibals
2 big-game hunters
1 doctor
1 nurse
3 missionaries

You arrive at a 100-yard-wide river filled with piranha fish. You must cross the river. There is a leaky canoe on your shore, which can hold, at most, 4 people. The cannibals paddle the best, followed by the hunters, the doctor, the nurse, and the missionaries, who are notoriously weak. You must decide who gets in the canoe for each trip back and forth. Get the party across with a minimum of carnage.

The doctor can attend major and minor wounds, unless he is himself wounded. The nurse can attend minor wounds. If the doctor is wounded, and the nurse is on the same shore as the doctor, she can (under his guidance) also attend major wounds.

Commands:

s	Prints status of game.
digit	For identification, each player is assigned a digit from 1 to 9. (See a status report). Typing a player's digit puts him in the canoe.
-	Takes everybody out of the canoe. Use this when you have put somebody in, and change your mind.
.	Starts the trip.

Put all your commands on the same line. A carriage return is unnecessary. For example:

259.

puts players 2, 5, and 9 in the canoe, and starts the trip.

Now try a game or two. When you want to learn more, read facts. Good luck!

Type .pf to play the game. When "seed" is printed, enter a random number.

4.2.1 Facts

The speed of the canoe is the average of the paddling strengths of the players in the canoe. A speed of 100 gets the canoe to the opposite shore just as it fills.

The initial paddling strengths are:

cannibals	120
hunters	90
doctor	70
nurse	50
missionaries	40

Strengths are multiplied by the following factors for unhealthy paddlers:

minor wound, attended	0.9
major wound, attended	0.8
minor wound, unattended	0.8
major wound, unattended	0.7
dead	0.0

During a trip certain events happen, with probabilities shown:

Canoe fills at predetermined rate.
During each (speed/4) yard of the trip a single pf jumps in the boat with probability 0.25. He picks a random toe. Cannibals always spear the fish, and half the time make a hole in the boat. Hunters always panic, and capsize the boat. The doctor is quick half the time, and panics half the time. The nurse always panics; half of the time she is calmed down, and the other half, she jumps (alone) out of the boat and must swim ashore.
When the boat capsizes, everybody must swim. Dead players always float to the correct shore, and somehow the canoe gets there, too.

When swimming, the events that follow may occur to each player individually:

Dead players always float ashore.
Live players make it ashore unscathed half the time. The other half, they acquire minor wounds (prob. 0.67) or major wounds (prob 0.33). In no case do they come out of the river healthier than they went into it.
At the present time, these probabilities are independent of the length of the swim. (Improvers take note.)

On the shore, a player's health can become worse:

Healthy players never get worse.
Attended players get worse with prob 0.11.
Unattended players get worse with prob 0.33.
Dead players never get worse.
To get worse means a minor wound becomes major, or a player with a major wound dies.
When a minor attended wound gets worse, it becomes a major unattended wound.
These "worse health" events are computed for every player once per canoe trip, whether or not the player participated in the latest trip. So players wounded early have more chances to get worse than players wounded later.

Score:

1000 for a perfect game.
-100 per dead player.
-30 for major unattended wounds.
-15 for major attended wounds.
-10 for minor unattended wounds.
-5 for minor attended wounds.

Highest score achieved to date is 995.

Maximum Carnage Game

Certain people with twisted minds may decide to try for a minimum score. If you do this, a new rule is needed: On each successive round trip of the canoe, you must leave at least one more person on the far shore than on the previous round trip.

Happy paddling!

4.2.2 Piranha Fish Code

```
char shore(9),health(9),canoe,move(4),ngoing,afloat
int hfactor(6),sinkrate,paddle(9)
/* Conducts the game.
pf [
  setup
  while(stillplaying())[
    whosgoing
    trip
    shoreacts
  ]
  wrapup
]

/* Sets up initial conditions.
setup [
  hfactor(Ø)=1Ø
  hfactor(1)=9
  hfactor(2)=hfactor(3)=8
  hfactor(4)=7
  paddle(1)=paddle(2)=12
  paddle(3)=paddle(4)=9
  paddle(5)=7
  paddle(6)=5
  paddle(7)=paddle(8)=paddle(9)=4
  sinkrate=25
  ps"seed"
  seed=last=gn
]

/* Game is still going if any player on shore Ø is alive.
stillplaying [
  int p
  while((p=p+1)<=9)
   if((shore(p)==Ø)*(health(p)<5)) return 1
]
```

```
/* Conducts dialog, determining which players make next trip.
whosgoing [
  char j,p,i
  char dup
  pl"";pl"move "
  while(1)[
    j=getchar
    if(j=='.')[    /* Trip command.
      i=Ø
      while((i=i+1)<=ngoing)     /* At least one paddler required.
        if(health(move(i))<5)return
      ps" nobody to paddle "
    ]
    else if(j=='-')[   /* Unload command.
      ngoing=Ø
      ps" canoe emptied"; pl""
    ]
    else if(j=='s')[   /* Print board.
      status
      ps"move "
    ]
    else if((j>='1')*(j<='9'))[    /* Put player in canoe.
      p=j-'Ø'
      dup=Ø
      i=Ø
      while((i=i+1)<=ngoing) if(p==move(i)) dup=1
      if(dup) ps" already in boat "
      else if(shore(p)!=canoe) ps" on other shore "
      else if(ngoing>=4) ps" canoe full "
      else move(ngoing=ngoing+1)=p
    ]
  ]
]
```

```
/* status prints the board.
status [
  char k(Ø),p
  pl"";pl""
  ps "near shore                    far shore  "
  pl"";pl""
  while((p=p+1)<=9)[
    if(shore(p)) ps "                                  "
    pn p; ps" "; pname p; ps" "
    if(health(p))[
  k="minor att  major att  minor unattmajor unattdead       "
      k=k+11*(health(p)-1)
      pft k,k+1Ø
    ]
    pl""
  ]
  if(canoe)ps "                      "
  ps "     canoe"
  pl"    "
  if(canoe)ps "                     "
  char i
  while((i=i+1)<=ngoing)pn move(i)
  pl""
]

/* Conducts a trip across the river.
trip [
  char i
  int speed,dist,full
  afloat=1
  while((i=i+1)<=ngoing)
    speed = speed + paddle(move(i))*hfactor(health(move(i)))
  speed=speed/(4*ngoing)    /* Yards per unit of time.
  while((dist=dist+speed)<1ØØ)[
    full=full+sinkrate
    if(afloat*(full>1ØØ))[
      pl"The boat is swamped....."
      capsize
```

```
      break
    ]
    if(afloat)[
      pl"Canoe has"; pn 100-dist; ps" yards to go, and is"
      pn full; ps"% full"
      if(random(1,4)==1)onefish
    ]
  ]
  i=0       /* The far shore is reached.
  while((i=i+1)<=ngoing) shore(move(i))=1-shore(move(i))
  canoe=1-canoe   /* Swap shores of players in canoe, and canoe.
  ngoing=0    /* Everybody out.
  pl"trip to "
  if(canoe)ps"far"; else ps"near"
  ps" shore is complete."
]

/* A fish jumped in the boat. This is what happens.
onefish
[
  char p
  pl"A piranha fish has jumped into the boat.  He is swimming"
  pl"around.  He is looking at the toe of the "
  pname(p=move(random(1,ngoing)))
  ps"."
  if(health(p)>4) pl"Oh, well.  He's dead anyway....."
  else if(p>6)[
    pl"The missionary is calm.  He is staring back at the"
    pl"fish.  The fish just jumped back into the river."
  ]
  else if(p<3)[
    pl"The cannibal has speared the fish.  "
    if(random(0,1))[
      pl"Unfortunately he made a hole in the"
      pl"boat, increasing its sink rate 10%."
      sinkrate=sinkrate+sinkrate/10
    ]
  ]
```

```
  else if(p<5)[
    pl"The hunter has panicked.  He is rocking the boat..."
    capsize
  ]
  else if(p==5)[
    if(random(Ø,1))[
      pl"The doctor is quick.  He shoots the fish full of"
      pl"a drug."
    ]
    else[
      pl"The doctor has panicked.  He is rocking the booooooat!"
      capsize
    ]
  ]
  else[
    pl"The nurse has panicked.  She is rocking the boat."
    pl"Everybody is yelling at her.  Yell - yell - yell."
    if(random(Ø,1))[
      pl"She is calm now, and sits down."
    ]
    else[
      pl"She falls out of the boat.  She is swimming."
      swim 6
    ]
  ]
]

/* Player p swims to shore.
swim char p [
  if(health(p)>4)[
    pl"Player"; pn p; ps" floats ashore."
  ]
  else if(random(Ø,1))[
    pl"Player"; pn p; ps" makes it."
  ]
  else[
    pl"BYTE!! BYTE!!  Player"; pn p
```

```
    if(random(Ø,2))[
      if(health(p)==2)health(p)=4
      else if(health(p)<2)health(p)=3
    ]
    else if(health(p)<4) health(p)=4
    if(health(p)==3) ps" fortunately escapes with minor wounds"
    else ps " major wounds acquired."
  ]
]

/* The canoe is capsized.
capsize [
  char p
  pl"CAPSIZE!!!  Everybody swim FAST!!  The fish are coming.."
  while((p=p+1)<=ngoing)swim move(p)
  afloat=Ø
]

/* When on shore, some players get mended.
shoreacts [
  char p
  while((p=p+1)<=9)[
    if(shore(p)==shore(5))[   /* Doctor with at most minor wounds can attend all
      if(health(5)<4) if(health(5)!=2)                    /*    wounds.
        if((health(p)==3)+(health(p)==4)) [
            health(p)=health(p)-2
            pl""; pn p; ps " attended by doctor."
        ]
    ]
    if(shore(p)==shore(6))[
      if(health(6)<4) if(health(6)!=2) if(health(p)==3) [
        health(p)=1
        pl""; pn p; ps" attended by nurse."
      ]
      else if(health(p)==4)   /* (And also major wounds with the doctor's advice.)
        if(shore(5)==shore(6))
          if(health(5)<5) [
```

```
            health(p)=2
            pl""; pn p; ps" attended by nurse"
          ]
      ]
      if(health(p)==Ø)[]    /* All done if healthy.
      else if(random(Ø,2))[]   /* All done for .67 of sick.
      else if(health(p)<3)[    /* But some get sicker.
        if(random(Ø,2)==Ø)[
          if((health(p)=health(p)+1)==3) health(p)=5
          pl""; pn p; ps" is much worse"
          if(health(p)==5) ps", in fact dead."
        ]
      ]
      else if(health(p)<5)[
        health(p)=health(p)+1
        pl""; pn p; ps" is much worse"
        if(health(p)==5)ps", in fact dead."
      ]
    ]
]

/* Computes score.
wrapup [
  int s,h,p
  s=1ØØØ     /* Perfect score.
  while((p=p+1)<=9)[
    h=health(p)
    if(h==5)s=s-1ØØ
    if(h==4)s=s-3Ø
    if(h==3)s=s-15
    if(h==2)s=s-1Ø
    if(h==1)s=s-5
  ]
  pl"";pl""
  status
  ps"Your score is"; pn s
]
```

```
/* Prints a player's name.
pname char p [
  char k(Ø)
  if(p<3)ps "cannibal"
  else if(p<5)ps "hunter"
  else if(p<6)ps "doctor"
  else if(p<7)ps "nurse"
  else ps "missionary"
]
```

4.2.3 Comments on Style

Notice how the functionality of this program makes it readable. You can find a feature quickly, and modify it with confidence that the house won't fall in.

Note the use of the character pointer k in status. It is used to compute for printing one of five possible health messages, depending on the health of player p. The function pft is used to print exactly 11 characters. Thus, from three lines of code any one of five messages is printed.

Piranha Fish uses only standard and optional library functions, and standard MCs. These are all furnished with tiny-c. So if you can get tiny-c up, you've got this program in the bag. If you use a plot function from your personal library, dramatic improvements to this game are possible.

4.3 The Standard Library and PPS

PPS is defined in Chapter III. We give its code here, including the standard library functions. This listing is in "human-readable" form, i.e., with comments, indenting, and long names. It's about 9000 bytes long. The machine-readable version of this program was "crunched" to about 4000 bytes. In general, programs should be written in a human-readable style, then a crunched version produced if the situation warrants it. This one clearly does.

```
/* Transmits c to the terminal.  If c is null transmits ".
putchar char c [
  if(c==Ø)c='""'
  return MC c,1
]
/* Reads one character from the terminal.
getchar [
 return MC 2
]
/* Reads a line from the terminal.  Implements character and line delete.
gs char b(Ø) [
  int 1
  while((b(1)=MC(2))!=13) [   /* Do until carriage return.
    if(b(1)==24) [   /* line kill
      1=Ø; pl""
    ]
    else if(b(1)==127) [   /* char kill
      if(1>Ø)1=1-1
    ]
    else 1=1+1
  ]
  b(1)=Ø   /* Put null at line's end.
  return 1
]
/* Prints a string.
ps char b(Ø) [
  int 1
  char c
  1=-1
  while((c=b(1=1+1))!=Ø)MC c,1
  return 1
]
```

```
/* Goes to new line and prints a string.
pl char b(Ø) [
  MC 13,1
  ps b
]
/* Tests if a is alphabetic.
alpha char a [
  if((a>='a')*(a<='z'))return 1
  if((a>='A')*(a<='Z'))return 1
]
/* Converts numeric character string b to integer.  Puts value in v(Ø).  Returns
/*    the number of bytes examined.  First non-digit stops the scan.
num char b(5)
    int v(Ø) [
  int k
  v(Ø)=Ø
  while(k<5) [
    if((b(k)<'Ø')+(b(k)>'9'))return k
    v(Ø)=1Ø*v(Ø)+b(k)-'Ø'
    k=k+1
  ]
  return k
]
/* Converts signed integer character string b to binary integer.  Puts value in
/*    v(Ø).  Returns the number of bytes examined.
atoi char b(Ø)
      int v(Ø) [
  int k,s
  char c
  s=1
  c=b(Ø)
  while((c==' ')+(c=='-')+(c=='+')) [
    if(c=='-')s=-1
    c=b(k=k+1)
  ]
  k=k+num(b+k,v)
  v(Ø)=s*v(Ø)
  return k
]
```

```
/* Prints a signed integer.
pn int n [
  MC ' ',1
  MC n,14
]
/* Reads a line from the terminal.  Gets a signed integer from the beginning of the
/*    line, and returns its value.  Insists on getting a number.
gn [
  char b(2Ø)
  int v(Ø)
  while(1) [
    gs b
    if(atoi b,v)return v(Ø)
    ps "number required "
  ]
]
/* Compares first n bytes starting at a with first n starting at b.  Returns 1 on
/*    match, Ø on no match.
ceqn char a(Ø),b(Ø)
      int n [
  int k
  k=-1
  while((k=k+1)<n) if(a(k)!=b(k))return Ø
  return 1
]
/* Searches string in (of length lin) for the first occurrence of the string find
/*    (of length lfind).  Returns Ø on not found, n>Ø on found, where n is the
/*    offset from in-1 where find was found.
index char in(Ø)
      int lin
      char find(Ø)
      int lfind [
  if(lfind<=Ø)return 1   /* Null text always found.
  if(lin<=Ø)return Ø
  int at,left(Ø)
  while(at+lfind<=lin) [
    left(Ø)=1            /* scann finds first char fast.
```

```
    at=at+1+scann(in+at,in+lin-1find,find(Ø),left)
    if(left(Ø))return Ø   /* If no first char, then fail.
    if(ceqn(in+at,find+1,1find-1))return at   /* If match, then succeed, else go
                                              /*    back to get another first
                                              /*    character.
  ]
]
/* Move string a to b.  First null in a stops the move.
move char a(Ø),b(Ø) [
  int k
  k=-1
  while(a(k=k+1)!=Ø)b(k)=a(k)
  b(k)=Ø     /* Move null too.
  return k   /* Number of bytes moved.
]
/* Read a line, return its first character.
gc [
  char f
  f=MC 2
  while(MC(2)!=13) []
  return f
]
/* Move the block a...b up or down n bytes.
movebl char a(Ø),b(Ø)
        int n [
  MC(a,b,n,7)
]   /* In the block a...b count occurrences of character c.
countch char a(Ø),b(Ø),c [
  return MC(a,b,c,8)
]   /* Scan the block a...b for the n(Ø)th occurrence of char c.  Reduce n(Ø) for
    /*    every c found.  Return pointer to last character examined.
scann char a(Ø),b(Ø),c
      int n(Ø) [
  return MC(a,b,c,n,9)
```

```
]   /* Read the file 'name' into 'where', but take care not to read into bytes
    /*    beyond 'limit'.  Use unit to do the read.
readfile char name(Ø),where(Ø),limit(Ø)
          int unit [
int k,total
MC(1,name,Ø,unit,3)   /* Open
while(1) [
    k=MC(where,unit,4)   /* Read a block
    if(k== -1)return total   /* End of file
    if(k< -1)return k   /* Error
    total=total+k
    if((where=where+k)>limit) [
      pl"Too big"
      return -2
    ]
  ]
]
/* Write the block from...to as a file named 'name', using unit to do the writing.
writefile char name(Ø),from(Ø),to(Ø)
          int unit [
  int k,total,l
  MC(2,name,to-from+1,unit,3)   /* Open for writing.
  while(from<=to) [
    l=to-from
    if(l>255)l=255   /* Blocksize-1 for the installation
    k=MC(from,from+l,unit,5)   /* Write one block.
    if(k<Ø)return k   /* Error
    if(k>Ø)return -k   /* Also error, but must return negative.
    total=total+l+1
    from=from+l+1
  ]
  k=MC(unit,6)   /* Write end-of-file.
  if(k<Ø)return k   /* Error
  if(k>Ø)return -k   /* Error
  return total   /* No error, return amount written.
]
endlibrary
```

```
int err(Ø)   /* err returned by application program.
int cursor   /* Pointer into current line.
int lineno   /* Current line number
int progend   /* Pointer to last byte of program.
int lpr   /* length of pr
char line(64)   /* Input line
char pr(7ØØØ)   /* Program buffer
int lline   /* Length of input line
int lastline   /* Number of lines in pr
char ltext(2Ø)   /* Locate text
char totext(4Ø)   /* Change to text
int len,tolen   /* Lengths of ltext, totext
main [
  char c
  int val(1)
  lpr=55ØØ
  pr(Ø)=13   /* Create empty line Ø.
  lastline=cursor=lineno=progend=err(Ø)=ltext(Ø)=Ø
  while(1) [
  ps">"
    while((lline=gs(line))==Ø) []   /* Read non-empty line.
    c=line(Ø)
    if(c=='.') [
      if(num(line+1,val))goto(val)   /* .number
      else if((line(2)==Ø) + (alpha(line(2))==Ø)) [
        c=line(1)   /* One-letter commands
        if(c=='p')print
        else if(c=='d')dlines
        else if(c=='l')locline
        else if(c=='c')change
        else if(c=='?')facts
        else if(c=='r')progin
        else if(c=='w')progout
        else if(c=='x')return
        else [ps"???";pl""]
]else start   /* Multi-letter commands
    ]
    else if(c=='-')up   /* Not . try + -
    else if(c=='+')down
    else insert   /* Not . or + or -
  ]
]
```

```
/* Prints n lines, making the last current
plines int n [
  int fc,lc,val(Ø)
  val(Ø)=n
  fc=fchar   /* First char to print.
  lineno=lineno+val(Ø)-1
  lc=cursor+scann(pr+cursor,pr+progend,13,val)   /* Last char to print.
  cursor=lc
  lineno=lineno-val(Ø)
  MC pr+fc,pr+lc,13   /* This does the printing.
]
/* Returns pointer to first char of current line.
fchar [
  int k
  if((k=cursor)==))return Ø
  while(pr(k=k-1)!=13)if(k<Ø)break
  return k+1
]
/* Returns pointer to last char of current line.
lchar [
  int k
  k=cursor-1
  while(pr(k=k+1)!=13)if(k>=progend)break
  return k
]
/* Advances cursor to next line.
nl [
  if((cursor=lchar()+1)>progend) [
    cursor=progend
    return Ø
  ]
  return lineno=lineno+1
]
```

```
/* Backs cursor to previous line.
bl [
  if((cursor=fchar()-1)<Ø)cursor=Ø
  else lineno=lineno-1
]
/* Prints a set of lines.
print [
  int val(Ø)
  if(line(2))num(line+3,val)
  else val(Ø)=1
  plines(val(Ø))
]
/* Deletes a set of lines.
dlines [
  int fc,lc,val(1)
  if(cursor==Ø) [
    ps"cannot delete line Ø";pl""
    return
  ]
  if(line(2)==Ø)val(Ø)=1
  else num(line+3,val)   /* val is how many lines to delete.
  lastline=lastline-val(Ø)
  fc=fchar   /* First char to delete.
  lc=cursor+scann(pr+cursor,pr+progend,13,val)   /* Last char to delete.
  lastline=lastline+val(Ø)   /* In case val is too big, adjust lastline by the
                             /*    excess.
  lineno=lineno-1   /* Back up current line.
  cursor=fc-1
  if(lc<progend)movebl(pr+lc+1,pr+progend,-(lc-fc+1))   /* Closes the non-deleted
                                                        /*    text.
  progend=progend-(lc-fc+1)
]
```

```
/* Locates a line with given text.
locline [
  int k
  if(line(2)!=Ø) [
    len=move(line+3,ltext)   /* Set up locate text.
    if(ltext(Ø)==' ')ltext(Ø)=13
    if(ltext(len-1)==' ')ltext(len-1)=13
  ]
  if(len==Ø) [
    pl"locate what?";pl""
    return
  ]
  if(nl()!=Ø) [   /* Scan starts at next line.
    if(k=index(pr+cursor-1,progend-cursor+2,ltext,len)) [   /* index does the
                                                            /*    search.
      cursor=cursor-2+k   /* Found, set cursor and lineno.
      if(pr(cursor)==13)cursor=cursor+1
      lineno=countch(pr,pr+cursor-1,13)
      plines 1   /* Print found line.
    ]
    else[ps"?"; pl""]   /* Not found
  ]
  else[ps"at bottom"; pl""]   /* No next line, can't even start.
]
/* Changes text within current line.
change[
  char del
  int ptr,fc,lc
  if(line(2)!=Ø) [   /* Default is previous locate, to texts.
    del=line(2)   /* Delimits locate and to texts.
    ptr=2
/*  Locate middle delimiter
    while((line(ptr=ptr+1)!=del) [
      if(line(ptr)==Ø) [   /* No middle delimiter
        line(ptr+1)=Ø   /* Second null creates empty to text.
        break
      ]
    ]
    line(ptr)=Ø   /* Null in middle delimiter.
    len=move(line+3,ltext)   /* New locate text
    tolen=move(line+ptr+1,totext)   /* New to text
```

```
    if(tolen)if(totext(tolen-1)==del)tolen=tolen-1    /* Remove optional third
                                                      /*    delimiter.
  ]
  fc=fchar   /* Scan line for ltext.
  lc=lchar()-1
  int k
  if(k=index(pr+fc,lc-fc+1,ltext,len)) [
    cursor=fc+k-1   /* Found, set cursor to where.
    movebl(pr+cursor+len,pr+progend,tolen-len)   /* Move tail end text in or out.
    progend=progend+tolen-len
    if(tolen)movebl(totext,totext+tolen-1,pr+cursor-totext)
  ]
  plines 1   /* Print the result.
]
/* Inserts one line of text after current line.
insert [
  lline=lline+1
  if(progend+lline>lpr) [
    ps"won't fit";pl""
    return
  ]
  if(nl)movebl(pr+cursor,pr+progend,lline)   /* Move tail out.
  else[cursor=cursor+1; lineno=lineno+1]
  progend=progend+lline
  movebl(line,line+lline-1,pr-line+cursor)   /* Move in new line.
  pr(cursor+lline-1)=13   /* Put return at end.
  lastline=lastline+1
]
/* Gives facts about current line, including err code.
whathappened [
  int fc,lc,lcurs,blanks
  pn lineno; ps" --- err "; pn err(0); pl""
  lcurs=cursor
  fc=fchar
  blanks=lcurs-fc
  lc=lchar
  fc=fc-1
```

```
  while((fc=fc+1)<lc)putchar(pr(fc));pl""
  while((blanks=blanks-1)>=0)putchar(' ')
  putchar '<'; pl""
]
/* Moves cursor down.
down [
  int val(1)
  if(line(1)==0)val(0)=1
  else num(line+1,val)
  lineno=lineno+val(0)
  val(0)=val(0)+1
  cursor=cursor+scann(pr+cursor,pr+progend,13,val)
  lineno=lineno-val(0)
  plines(1)
]
/* Moves cursor up.
up [
  int lines,val(1)
  if(line(1)==0)lines=1
  else[ num(line+1,val); lines=val(0)]
  while((lines=lines-1)>=0)bl
  plines(1)
]
/* Move cursor to given line.
goto int l(1) [
  lineno=l(0)
  l(0)=l(0)+1
  cursor=scann(pr,pr+progend,13,1)
  lineno=lineno-l(0)
  plines(1)
]
/* Print some facts.
facts [
  pn lineno
  pn lastline
  pn progend
  pn lpr-progend
  pl""
]
```

```
/* Enter an application program.
start [
  while(lline<=64) [
    line(lline)=' '
    lline=lline+1
  ]
  MC(err,line+1,pr+progend,pr,11)
  if(cursor<Ø)cursor=Ø; if(cursor>progend)cursor=progend
  lineno=countch(pr,pr+cursor-1,13)
pl"";pl""
  if(err(Ø))
    if(err(Ø)==99) [ps"stopped";pl""]
    else whathappened
]
/* Read a file into pr.
progin [
  int k
  k=readfile(line+3,pr+progend+1,pr+1pr,1)
  if(k<Ø)return
  progend=progend+k
  lastline=countch(pr+1,pr+progend,13)
  pn k; pl""; facts
]
/* Writes all of pr to a file.
progout [
  facts
  pn writefile(line+3,pr+1,pr+progend,1)
  pl""
]
```

4.3.2 Comments on Style

The library routines are clean, and fairly straightforward. index requires study to understand. The clue is this: to get a modicum of speed, at least the first byte had to be matched at machine code speed. scann had the ability to do this, so it was adopted. scann is a technically difficult function to master; this is a good demonstration of that fact. The problem was broken into two parts: match the first byte and then test if the remaining bytes match. The second part is done by the if(ceqn(...)).

movebl, countch, and scann (and a few others) are examples of wrapping an MC in a nice package, and tying a bow on it. This should be done to all MCs. Sometimes a modest variation can be made in the packaging, as in putchar, where nulls are mapped to quotes before MC 1 is called. This is done to keep the MCs "pure", while, at the same time, incorporating a small variation in its predominant usage. Another function could still use the MC without the variation, or with yet another variation.

Students of software archeology will recognize the PPS code as a product of evolution. Originally, the only way to move the current line was with the functions nl and bl. The up, down, and goto functions all calculated an appropriate number of nls or bls and then did them. They worked, but slowly. So goto was recoded using scann; in fact, it was the motivation for scann. Next, down was recoded using scann, making it fast also. But up still uses bl and is still slow. A cleaner design now would be to eliminate nl and bl altogether, and code up and down in terms of goto. But even with this dichotomy of method, the code is well structured. Most of the routines are quickly read and understood. locate and change are (not surprisingly) the only difficult ones.

4.3.3 The Use of MC 11

A tiny-c program loaded via the loader is, by definition, a LEVEL ZERO PROGRAM. Usually this is PPS, but it could be any system-type program. When a program uses MC 11 to invoke (not call) another program, the invoked program is given a level one higher than the invoking program. Thus, when PPS invokes an application, that application runs at level one. An application is also allowed to use MC 11, creating levels higher than one.

The principal need for this is in using PPS to program and test new or modified PPSs. A working PPS is loaded and started at level zero. An experimental PPS is loaded as a level one application. It is edited, started and tested just like any other application. When the level one experimental PPS is running, it is preparing the text of still another program, which, if started, runs at level two.

In tiny-c/8080, the global variable APPLVL contains the current application level. An application can be stopped by pressing the ESCAPE key. This terminates the program, and causes the invoking MC 11 to return. The application level is reduced by one. The error 99 is returned in FACTS. ESCAPE only works to terminate applications. It cannot terminate a level zero program. The choice of ASCII character that stops an application can be changed. The global variable ESCAPE contains the ASCII character that causes an application stop. It can be changed to any value not used in programming or for data (see Section 6.5.4.2).

tiny-c/11 also contains a global variable called APPLVL which is incremented as MC 11 is entered, and decremented as MC 11 is left. This variable can be examined by interrupt routines to determine how to handle an interrupt.

4.4 Morse Code Generator

Do you have something on your computer that goes beep? For example, a printer that beeps when it's sent ASCII BELL? Some printers make a long continuous beep when sent several BELLs.

This program allows you to practice receiving individual letters in Morse code, or to send a Morse code message. Type

```
>.practice
```

to practice. Answer the four questions. (The last seeds the random number generator.) Then write down the letters beeped to you in Morse code. At the end, compare your answers with the ones displayed. Or have a friend type

```
>.morse "any message"
```

and listen to the message he is sending.

FIGURE 4-2

```
int SPEED
bell[
  MC 1ØØ2
  MC 7,1
  int k
  while((k=k+1)<3)[]
  MC 1ØØ3
]
lots[
  while(1)bell
]
dot[
  bell
  pause
]
dash[
  bell; bell; bell; pause
]
pause[
  int k
  while((k=k+1)<SPEED)[]
]
space[
  int r
  while((r=r+1)<2Ø)[]
]
letter char c [
  pause
  int k
  while((k=k+1)<SPEED)pause
  if(c=='a')[dot;dash]
  else if(c=='b')[dash;dot;dot;dot]
  else if(c=='c')[dash;dot;dash;dot]
  else if(c=='d')[dash;dot;dot]
  else if(c=='e')dot
  else if(c=='f')[dot;dot;dash;dot]
  else if(c=='g')[dash;dash;dot]
  else if(c=='h')[dot;dot;dot;dot]
  else if(c=='i')[dot;dot]
  else if(c=='j')[dot;dash;dash;dash]
  else if(c=='k')[dash;dot;dash]
  else if(c=='l')[dot;dash;dot;dot]
  else if(c=='m')[dash;dash]
```

```
  else if(c=='n')[dash;dot]
  else if(c=='o')[dash;dash;dash]
  else if(c=='p')[dot;dash;dash;dot]
  else if(c=='q')[dash;dash;dot;dash]
  else if(c=='r')[dot;dash;dot]
  else if(c=='s')[dot;dot;dot]
  else if(c=='t')dash
  else if(c=='u')[dot;dot;dash]
  else if(c=='v')[dot;dot;dot;dash]
  else if(c=='w')[dot;dash;dash]
  else if(c=='x')[dash;dot;dot;dash]
  else if(c=='y')[dash;dot;dash;dash]
  else if(c=='z')[dash;dash;dot;dot]
  else if(c==' ')[pause;pause;pause]
]
morse char s(0) [
  SPEED=12
  while(s(0))[
    letter s(0)
    s=s+1
  ]
]
practice [
  char c
  int k,n,r,rl
  ps"how many letters'
  k=gn
  ps"how many repeats"
  r=gn
  ps "speed"
  SPEED=gn
  ps "seed"
  seed=last=gn
  while((n=n+1)<=k)[
    c=random 'a','z'
    rl=0
    while((rl=rl+1)<=r)letter c
    letter ' '
    putchar c; putchar ' '
  ]
]
```

End of FIGURE 4-2

4.4.1 Comments on Style

You may have to replace dot and dash to conform to different hardware. You may also have to experiment awhile to get the timing correct. In bell, (which you may have to replace), two private MCs are used. 1002 enables the printer, and 1003 disables it. The MC 7,1 transmits ASCII BELL. Thus, nothing goes to the printer except BELLs. SPEED is set to a default value in line two of function morse. A lower value causes faster code generation.

4.5 A Tape-to-Printer Copy Utility

A handy utility is one that reads a file from a cassette or disk, and prints it. One would think that this is ordinarily an assembly language job. But here, written almost entirely in tiny-c, is the utility used to print Appendix A. The only parts not in tiny-c are two private MCs: 1002 enables the printer, and 1003 disables it.

The system for which this utility was written has a 300 baud printer and a 2400 baud cassette recorder. They both use the same USART, so only one device can be enabled at a time. This utility conforms to that restriction. It opens the file, reads one block of up to 256 bytes, and closes the file, again freeing the USART. Then it opens the printer, prints the block, and closes the printer, thus freeing the USART for use on the file.

FIGURE 4-3

```
/* Copies a file from cassette to printer.
pfile char name(Ø)[
  char a(256)
  int len,err
  while(1)[
    err=fopen(1,name,Ø,1)   /* Open to read a block.
    if(err)[
      pl"open err";pn err
      return
    ]
    len=fread(a,1)
    fclose(1)
    if(len<Ø)[
      if(len == -1) [pl"readblock err";pn len]
      return
    ]
    popen      /* Open the printer.
    pft(a,a+len-1)
    pclose
  ]
]
popen[MC 1ØØ2]
pclose[MC 1ØØ3]
```

End of FIGURE 4-3

4.6 TV Graphics Functions

There is a variety of devices available for plotting on a TV screen. Generally, they divide the screen into a rectangular grid and allow selective "painting" or "erasing" of any cell in the grid. Some provide for only black or white cells and some allow up to 16 colors. The tiny-c library does not include TV graphics functions because they are device-dependent.

Here are two function specifications for black and white TV graphics, but easily modified for color. We assume the device has R rows and C columns of cells. The rows are numbered top down from 0 to R-1, the columns left to right from 0 to C-1.

clean
The screen is erased.

plot int row, col, onoff
Tests if row and col designate an onscreen spot (i.e., 0<=row < R, and 0<=col < C). If this is NOT true, plot takes no action, and returns a 1. If it is true, and if onoff is nonzero, the spot designated by row and col is "painted" white or turned on; if onoff is zero, the spot is "painted" black or turned off. In either case, plot returns a 0.

The definition of plot can be extended to color grid cells by giving meanings to different nonzero values of onoff. Note also that this definition requires the plot function to accept ANY value for row and col, even one wildly off-screen. plot cannot abort or modify a random memory byte just because row and col are off the screen. It simply plots nothing and returns a 1.

The next program demonstrates the use of plot.

4.6.1 Meteor Shower

Meteor shower is a graphic display program consisting of a main program called "star" and a function called "shoot". star queries the user for a seed for random, and the number of fixed and of shooting stars. The first while statement puts down the field of fixed stars. The second while statement causes a series of shooting stars to cross the screen. Most of the work is done by the function shoot, which puts one shooting star across the screen. It chooses a random entry point along the top edge, but not too close to the corner. It finds its angle of descent by setting delta at random. delta is the amount of horizontal motion for each vertical step. delta ranges from -3 to +3, and it results in a size between 0 and 4. This is skewed to favor small sizes

DAD, ARE THERE SUCH THINGS AS PIRANHA FISH?

UH-OH! I PUSHED "DEL" BY MISTAKE.

OH NO...

I GET CONFUSED IS IT I to A, A to I, OR BOTH?

BEEP... BEEP! BE... BEEP!

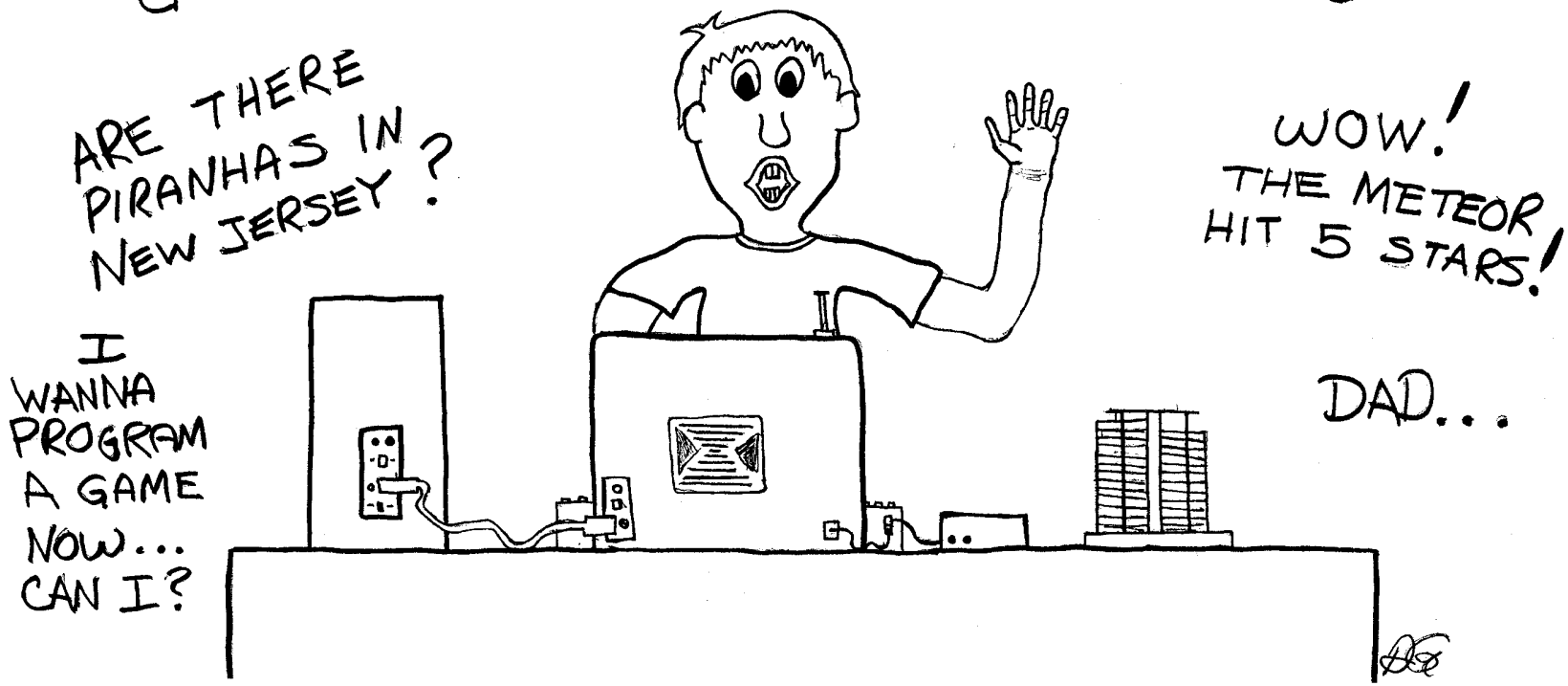
WHAT'S A CANNIBAL?

ARE THERE PIRANHAS IN NEW JERSEY?

WOW! THE METEOR HIT 5 STARS!

I WANNA PROGRAM A GAME NOW... CAN I?

DAD...



by multiplying two random numbers together. The while statement in shoot causes the motion. It repeatedly paints a new head, and erases the tail. How far back the erasing is done is determined by a variable called "big". For increasing values of k, the spot at row k and column start+delta*k is painted. This extends the shooting star down one step. Then the spot painted big steps ago is erased. The first big times this erasure is off screen, but that does no harm. When the erasure is off screen and k is larger than big, then the shooting star has completely traversed the screen. This causes a return, which leaves both the while, and shoot itself.

Notice that if a shooting star makes a direct hit on a fixed star, the fixed star is erased simultaneously with the tail of the shooting star.

For added interest, large shooting stars, called cruisers, erase not only direct hits, but also any fixed stars they merely approach. The last if statement does this with two plots.

```
/*
/* Meteor shower program, by T. A. Gibson, Nov. 1977.
/* Demonstrates use of random and plot.
star[
  clean
  pl"give a pattern number "
  seed=last=gn
  int k,stars
  int shoots
  ps" How many fixed stars "
  stars=gn
  ps" How many shooting stars "
  shoots=gn
  clean
  while((k=k+1)<stars)plot random(1,46),random(1,126),1
  k=Ø
  while((k=k+1)<shoots)shoot
]
shoot[
  int k
  int start
  start=random(2Ø,1Ø7)
  int big
  big=random(Ø,2)*random(Ø,2)
  int delta
  delta=random(-3,+3)
  while(1)[
    k=k+1
    plot k,start + delta*k , 1
    if(plot (k-big, start + delta*(k-big), Ø)) if(k>big) return
    if(big>3)[
      plot k-big, start+1+delta*(k-big),Ø
      plot k-big, start-1+delta*(k-big),Ø
    ]
  ]
]
```

End of FIGURE 4-4

V. INTERNAL OPERATION OF THE tiny-c INTERPRETER

For the student, the merely curious, or the software hack, this chapter describes the internal operation of the tiny-c interpreter. Software purchasers have a right to a thorough explanation of the operation of the product, INCLUDING commented source code.

Currently there are two implementations of tiny-c: the 8080 and the PDP-11. Both programs implement the same data structure and control flow. Both contain essentially the same set of subroutines.

In the narrative parts of this chapter, we describe the action of each of these subroutines. There are four parts to each description: arguments, results, errors, action.

ARGUMENTS describe data given to the subroutine through its calling sequence, e.g., via registers or the machine stack. We do not say in the narrative how arguments are passed, as that varies among different implementations. Refer to the listings for these details. Some routines will also use data in global cells such as CURSOR and PROGEND. These are not listed as arguments. Their use is described under the action part of the narrative.

RESULTS are data given back to the calling routine. This does not include modifications to the global cells. Many routines return no results to the calling routine. This does not mean they don't do anything! Don't confuse results (an explicit signal to the calling routine) with action (what the called routine does.)

ERRORS refers to errors detected by the interpreter in the tiny-c program being processed. All errors result in a call to ESET. Many routines detect no errors.

ACTION describes what the routine accomplishes.

The narrative has been kept general, so it will apply to all implementations, even improved ones. For example, a faster

search algorithm could be installed in ADDRVAL, but the narrative describing ADDRVAL would remain the same. The way we accomplish this is to say WHAT ADDRVAL does, NOT HOW it does it. The what-not-how approach to software description assumes the reader can read listings to determine the explicit algorithms used.

Section 5.12 is the sole exception. It describes subroutines needed only on eight-bit processors. Currently this discussion applies solely to the INTEL 8080. These routines are not in the PDP-11 version, nor are they likely to be necessary in any processor with full 16-bit arithmetic.

We begin this chapter with a discussion of tiny-c data objects.

5.1 Data Objects

There are two data declarations in tiny-c, int and char. They declare data objects which have five intrinsic properties: name, class, size of one datum, length, address where allocated.

5.1.1 Name

The name of an object is given in its declaration. For example, in

```
int X(10)
```

the object has name X. At most, 8 characters of a name are remembered, the first 7 and the last. Longer names may be used but their 8th through next-to-last character mean nothing. Thus

```
WHATHAPPENED
WHATHAPD
WHATHAPHAZARD
```

all appear to tiny-c as the same name. The second of these is the canonicalized eight-byte form.

5.1.2 Class

There is a difference between X and Y in:

```
char X, Y(0)
```

X is a character datum. Y is the address of a character datum.

The number of addresses one must go through to get an actual datum is the CLASS of the object. So X has class 0, and Y has class 1. You can think of class as the dimension of the object when viewed as a matrix. Thus X has dimension 0, and Y dimension 1.

Y(0) also has a class attribute. Y(0) is a character datum. So it has class 0. Consider

```
int VECTOR(80)
```

VECTOR has class 1. VECTOR(17) has class 0.

Classes greater than 1 theoretically exist but are not implemented in tiny-c.

5.1.3 Size

Size refers to the size of the ultimately referenced object. It does not refer to the size of an array. All character data has size 1. All integer data has size 2. Thus in

```
char LETTER, LINE (80)
```

both LETTER and LINE have size 1.

5.1.4 Length

The length of an object is measured in number of data items, not bytes. All class 0 objects have length 1. Class 1 objects have lengths equal to the declared size of the array.

Note that this is one larger than the number written, because element 0 must be counted. Thus,

```
char LINE (80)
int VECTOR (80)
```

both have length 81.

5.1.5 Address

The address of an object is the lowest memory byte address where the object's value(s) is(are) stored. An n byte (size * length = n) object has its value(s) stored in address, address + 1, ..., address + n-1.

5.2 The Variable Table

All functions and all currently active variables are described in the variable table. An entry in the variable table is 14 bytes:

BYTES	ENTRY
0-7	eight-byte name (canonicalized)
8	class
9	size (of one datum)
10-11	length (in data elements)
12-13	address of first value

A class value of ASCII 'E' for entry represents a function name. Size and length are ignored. The address value is where the cursor must be placed to begin execution of the function.

The variable table is allocated from the address stored in BVAR to that stored in EVAR inclusive. BVAR and EVAR are in the installation vector.

Currently active variables are:

-- all library symbols, including library functions
-- all global symbols, including functions
-- all local variables for currently active functions, including function arguments.

A function is active if it has been invoked, but has not completed. In the case of recursion, all incarnations count as active. All function names are either global or library, and all are entered in the variable table. Note that the function's name is entered without regard for its active or inactive status. The active function simply determines which local variables are currently active.

Suppose the following code is being interpreted, and control is currently in aaa:

```
char globalsymbol
main [
int localsymbol
   aaa 2,3
   bbb
]
aaa int x,y [
   int n
      .
      .     <----- current control point
      .
]
bbb [
char anothersymbol
   .
   .
   .
]
```

The global symbols are:

```
globalsymbol
main
aaa
bbb
```

The active functions are main and aaa. Their local variables are:

```
localsymbol
x
y
n
```

These eight variables are the ones allocated in the variable table. Later when aaa returns, x, y and n are de-allocated. When bbb is invoked, anothersymbol is allocated. At that point six symbols are in the table:

```
globalsymbol
main
aaa
bbb
localsymbol
anothersymbol
```

5.3 Layers in the Variable Table

The variable table is organized into layers so only those symbols accessible to a function will be searched. A symbol is accessible if it is

* local to the function, or
* global, or in the
* library.

This excludes variables local to active functions not currently executing. Thus:

```
int g
main [
   int x
   aaa
]
aaa [
   int y
   bbb
]
bbb [
   int z
      .
      .     <----- current control point
      .
]
```

The variables g and z are accessible to bbb. x and y, although active, are not accessible to bbb.

Layer 0 is reserved for library symbols. Globals are in layer 1 initially, but can occur in other layers (See Section 5.10.11). After layer 1, if there are n currently active functions, i.e. a main program, a function invoked by main, a function invoked by that function, etc., n levels deep, then those n functions are in layers 2, 3, ..., n+1. The locals for main are in layer 2. The locals for the currently executing function are in layer n+1. This layer is called CURFUN.

When a symbol is needed, it is searched as follows:

```
First, all symbols in the CURFUN layer are searched.
       If not found there, then
Second, all symbols in the global layer are searched.
       If not found there, then
Last, all symbols in the library layer are searched.
       If not found there, a symbol error is raised.
```

Notice this order of search guarantees the properties in Section 2.3.2.

5.4 The Function Table

The mechanism for layering is the function table, allocated from the address stored in BFUN to that stored in EFUN. BFUN and EFUN are in the installation vector. An entry in the function table has three addresses (six bytes).

BYTES	
0-1	The address of the first variable table entry for this function.
2-3	The address of the last variable table entry for this function.
4-5	The last variable table address allocated to this function.

The address in CURFUN points to the first byte of the function table entry of the currently executing function. The address in CURGLBL points to the first byte of the function table entry for current globals.

Library symbols are always in layer 0, and BFUN points to the first byte of that layer. So the implementation of the search order described above is simply to search symbols spanned by CURFUN, then CURGLBL, then BFUN.

5.5 Symbol Table Tools

The following subroutines manipulate and search the variable and function tables:

```
NEWFUN
FUNDONE
NEWVAR
ADDRVAL
```

These subroutines use the auxiliary subroutine

```
CANON
```

to canonicalize variable names to eight bytes.

5.5.1 NEWFUN

Arguments: None
Results: None
Errors: Too many functions
Action: Allocates a new layer in the function table, with zero entries in the variable table.

5.5.2 FUNDONE

Arguments: None
Results: None
Errors: None
Action: Deallocates a layer in the function table. Deallocates all variables in that layer. Deallocates all value space seized within that layer.

5.5.3 NEWVAR

Arguments: Canonicalized variable name, class, size, length, and value.
Results: Address of first byte of allocated space.
Errors: Too many variables, or too many values.
Action: Enters a new variable in the current layer of the variable table.

There are several considerations in allocating a value for a variable. If the variable is local or global, space is allocated and set to 0. If the variable is an argument to a function, and is class 0 (i.e. a single datum, not an array name) then space is allocated, and the passed value copied into the space. If the variable is an argument to a function, and is class greater than zero, then it is the address of an existing array. Space is not reallocated for these variables. The passed value - the array address - is entered as the new variable's address-of-first-value (bytes 12-13 of variable table). Thus, in effect, the array is passed.

Values are allocated in program space. PROGEND points to the last byte used for program. PRUSED points to the last byte

used for program and values. EPR points to the last byte of program space. EPR is in the installation vector.

5.5.4 ADDRVAL

Arguments: Canonicalized variable name.
Results: Address of first byte of allocated space, class, size, and length.
Errors: Variable name not found.
Action: Searches for the variable name and returns its properties. Searches locals, globals, and library symbols in that order.

5.5.5 CANON

Arguments: Addresses of first and last bytes of an uncanonicalized variable name and the address of an eight-byte buffer.
Results: Canonicalized variable name in buffer.
Errors: None
Action: Eight bytes are placed in the buffer. If the character string is shorter than eight bytes, it is padded on the right with nulls. If the variable name is longer than eight bytes, the first seven bytes and the last byte are placed in the buffer.

5.6 The tiny-c Stack

tiny-c has a stack for use in evaluating expressions. This stack is implemented in software, and is not to be confused with the 8080 or PDP-11 machine stack. This stack, called the tiny-c stack, is allocated from the address stored in BSTACK to the address stored in ESTACK. BSTACK and ESTACK are in the installation vector.

An entry on the tiny-c stack is 5 bytes:

BYTE	ENTRY
0	class, defined in Section 5.1.2
1	lvalue, defined below
2	size, defined in Section 5.1.3
4-5	stuff, defined below

An lvalue (left-side-value) is any symbol that can be written to the left of an equal sign. Allowable lvalues include variable names, subscripted array names, and unsubscripted array names.

The variable names and subscripted array names have class 0, and their value is a single character or integer datum. The unsubscripted array names have class 1, and their values are addresses.

Processing of an expression is done left to right. Whenever an lvalue is recognized, its attributes are put on the tiny-c stack. Bytes 4 and 5, stuff, are set to the address where the value is stored. An ASCII 'L' is put in byte 1, lvalue, signifying stuff is a REFERENCE to the actual value.

Whenever a constant is processed, or the results of arithmetic must be stacked, then the actual data value is put into stuff. An ASCII 'A' is put into lvalue, signifying stuff IS actual data.

Consider the processing of the following expression

```
K = J + 1
```

where K and J are both integers. Suppose K has address 4000, J has address 4002, and J has value 7.

STEP	ELEMENT		tiny-c STACK CLASS	LVALUE	SIZE	STUFF
1	K	Top --->	0	L	2	4000
2	J	Top --->	0	L	2	4002
			0	L	2	4000
3	1	Top --->	0	A	2	1
			0	L	2	4002
			0	L	2	4000

At this point the plus operation is performed. First, the 1 is popped off the stack. It is an actual value, so it is left as a 1. Then the 4002 is popped. It is an lvalue so the contents of addresses 4002 and 4003 (two bytes because size is 2) are retrieved. This is the value of J, 7. They are added, and the sum pushed as an actual:

STEP		tiny-c STACK CLASS	LVALUE	SIZE	STUFF
4	Top --->	0	A	2	8
		0	L	2	4000

Now the equal operation is performed. The top entry, 8, is popped and held. The next entry is popped. It must be an lvalue but it is not converted to an actual, because the current value of K is irrelevant. Only the address of K is needed, and that is 4000. The held 8 is stored in bytes 4000 and 4001, thus performing the assignment.

The value of the expression is 8. (Recall that all expressions, including those with an equal sign, have values.) So the last step in processing this expression is to push the 8 back on the stack as an actual.

STEP		tiny-c STACK CLASS	LVALUE	SIZE	STUFF
5	Top --->	0	A	2	8

5.7 Stack Tools

The following subroutines manipulate the tiny-c stack:

PUSH
PUSHK
PZERO
PONE
POP
TOPTOI
POPTWO
TOPDIF
EQ

5.7.1 PUSH

Arguments: Class, lvalue, size, stuff
Results: None
Errors: Stack full
Action: The arguments are pushed onto the stack as its top entry.

5.7.2 PUSHK (8080) and PUSHINT (PDP-11)

Arguments: Stuff
Results: None
Errors: Stack full
Action: Stuff is pushed onto the stack with class 0, lvalue A, size 2.

5.7.3 PZERO, PONE (8080)

Arguments: None
Results: None
Errors: Stack full
Action: Same as PUSHK with stuff 0 for PZERO and 1 for PONE.

5.7.4 POP (8080)

Arguments: None
Results: Class, lvalue, size, stuff
Errors: None
Action: The stack is popped and the results returned.

5.7.5 POP (PDP-11)

Arguments: None
Results: Stuff
Errors: None
Action: The stack is popped and stuff returned.

5.7.6 TOPTOI

Arguments: None
Results: An actual value in integer form.
Errors: None
Action: The stack is popped. If the popped entry is an lvalue, it is converted to an actual. Otherwise, the datum in stuff is used. If the resulting datum is of size 2 or of class greater than 0 (i.e., a two-byte address) it is returned. If the datum is of size 1 and class 0, it is first converted to an equivalent two-byte integer and then returned.

5.7.7 POPTWO (8080)

Arguments: None
Results: Two integers
Errors: None
Action: TOPTOI is called twice and both of its results are returned.

5.7.8 TOPDIF (8080)

Arguments: None
Results: An integer
Errors: None
Action: The top two layers are popped and their difference (next-to-top minus top) returned.

5.7.9 EQ

Arguments: None
Results: None
Errors: Left-value error
Action: The top level is popped and converted to an actual if necessary. The next level is popped and must be an lvalue, or an lvalue error is raised. The value referenced by the lvalue is replaced by the actual. Eight bits are replaced if the lvalue represents a class 0 eight-bit datum (i.e. a character). 16 bits are replaced if the lvalue represents a class greater than 0 (i.e. an address) or a class 0 16-bit datum (i.e. an integer).

5.8 Scanning Tools

The scanning tools are subroutines used by the tiny-c interpreter to recognize tokens of the language. They do not themselves define the language, and in fact could be used in parsers of languages other than tiny-c.

Most of these tools make use of the contents of the global variable CURSOR. This is the address of the next character to be examined in the tiny-c program being interpreted. We

say that "cursor points to" that character. Some of these tools "advance the cursor", i.e., increment the address in CURSOR, so that it points to a character further on in the program.

5.8.1 LIT

Arguments: The address of the first byte of a character string which is to be matched. The string is terminated by a null byte.
Results: A flag signalling match or no match.
Errors: None
Action: The cursor is first advanced over blanks. If the next characters match the literal, then the cursor is advanced beyond them and a match is signalled. Otherwise, the cursor remains on the first non-blank character and no match is signalled.

5.8.2 SKIP

Arguments: A left delimiter and a right delimiter, each a single character.
Results: None
Errors: Cursor runaway
Action: The cursor is advanced beyond a balanced pair of left and right delimiters. It is assumed that one left delimiter has already been matched. Cursor runaway occurs when a character beyond PROGEND is being examined.

5.8.3 SYMNAME

Arguments: None
Results: A flag signalling match or no match.
Errors: None
Action: The cursor is first advanced over blanks. Then, if the next set of characters is a symbol, the cursor is advanced beyond it, the global variable FNAME is pointed to its first character, the global variable LNAME to its last, and a match is signalled. Otherwise, the cursor remains on the first non-blank character

and a no match is signalled. A symbol is an alphabetic character followed by an arbitrary number of alphanumeric characters.

5.8.4 CONST

Arguments: None
Results: A flag signalling match or no match and the type of constant matched.
Errors: Cursor runaway
Action: The cursor is first advanced over blanks. Then one of three types of constants is matched:

1. Number: an optional + or - followed by any number of digits.
2. Quoted String: "anything" or "anything null-byte
3. Primed String: 'anything'

If a number is matched, FNAME and LNAME are pointed to its first and last significant characters, including sign if any. The cursor is advanced beyond the last digit.

If a quoted string is matched, FNAME is pointed to the first byte AFTER the first quote. LNAME is not used. The second quote is replaced by a null byte. The cursor is advanced beyond this null. This makes string constants conform to the tiny-c convention that character strings end with a null.

If a primed string is matched, FNAME points to the first byte AFTER the first prime. LNAME is not used. The second prime is not modified. The cursor is advanced beyond the second prime.

5.8.5 REM

Arguments: None
Results: None
Errors: Cursor runaway
Action: Advances cursor past all remarks, carriage returns, and blanks.

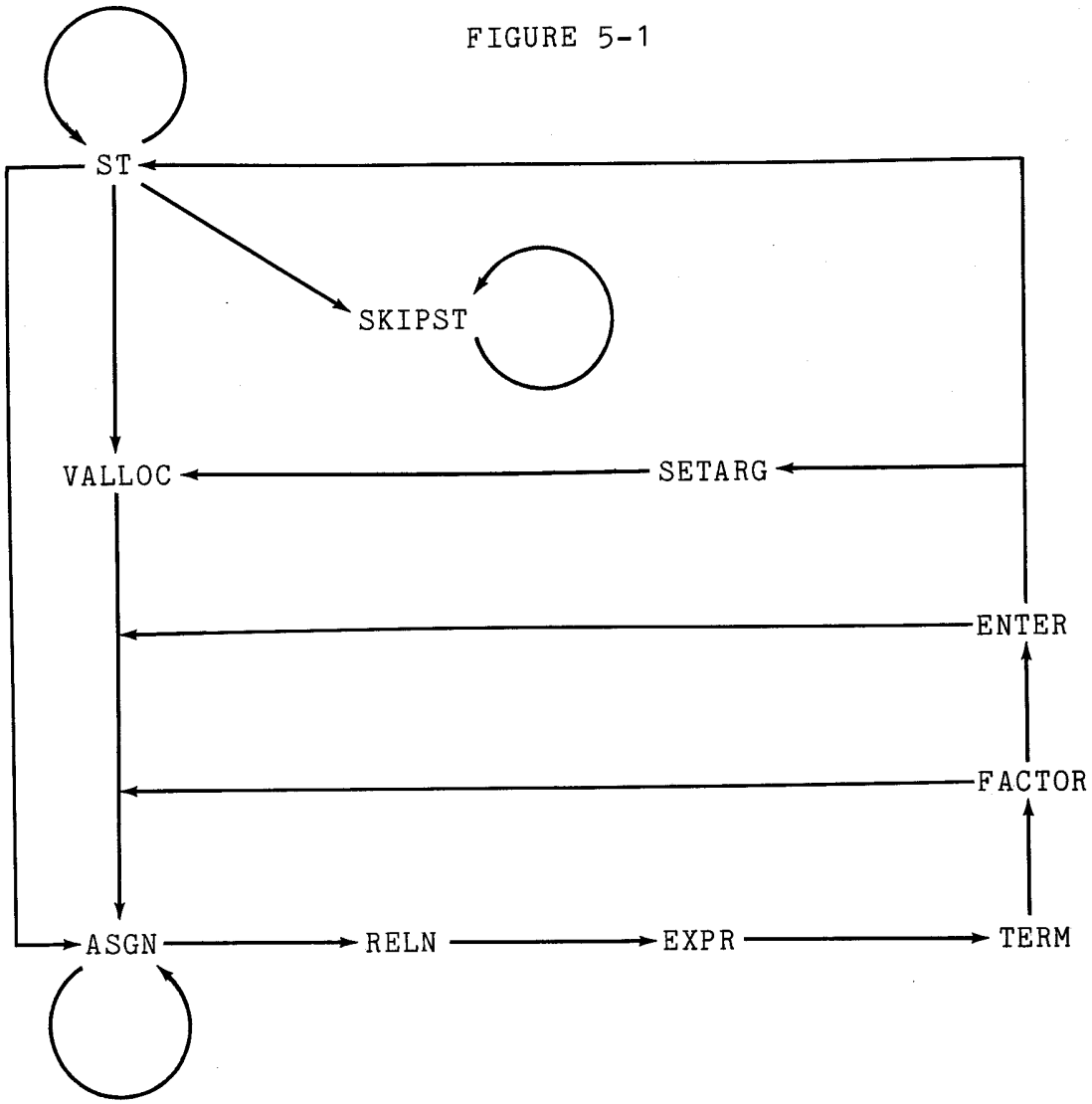
5.8.6 BLANKS

Arguments: None
Results: None
Errors: None
Action: Advances cursor beyond blanks.

5.9 The Statement Analyzer

The tiny-c statement analyzer is a collection of eleven subroutines which analyze and evaluate tiny-c statements. Figure 5-1 shows the calling relationship among the ten routines.

FIGURE 5-1



End of FIGURE 5-1

Briefly, the action of each of the routines is as follows:

ST -- determines the kind of statement currently being analyzed and keeps track of the level of statement compounding.

SKIPST -- moves the cursor beyond a complete, possibly compound, statement without interpreting it.

VALLOC -- parses variables in int and char statements, and calls NEWVAR to allocate them.

ASGN -- matches expressions and calls EQ to perform any assignments.

RELN -- evaluates an expression by calling EXPR. If a relational operator follows the expression, evaluates the next expression and then evaluates the relation between the two expressions.

EXPR -- evaluates unary plus and minus of a term and binary plus and minus between terms. Calls TERM to evaluate terms.

TERM -- evaluates multiplication, division, and remainder operators between factors. Calls FACTOR to evaluate factors.

FACTOR -- evaluates a factor which is an expression enclosed in parentheses, a constant, or a variable name including a function call. Evaluates function names by calling ENTER.

ENTER -- transfers control to a function by stacking the current position of the cursor and setting the cursor to the beginning of the function. Allocates a new layer (Sections 5.3 and 5.4) in the function table by calling NEWFUN and assigns argument values to local variables by calling SETARG.

SETARG -- assigns function argument values in a function call to the local variables used in the definition of the function. Enters local variable names in the variable table by calling VALLOC.

LINK -- enters global names in a new layer of the variable table.

Another view of the action of each statement analyzer routine is provided by examining the characters and character strings that each routine recognizes and past which it moves the cursor. Significant cursor movement routines, i.e., routines which move the cursor beyond things other than blanks and remarks, are LIT, CONST, SKIPST, and SYMNAME. Of these, only LIT can move the cursor beyond different strings of symbols depending on its argument. In Figure 5-2 we display the characters with which each statement analyzer routine calls LIT and if any of these routines calls CONST, SKIPST, or SYMNAME, we indicate this fact by enclosing the routine name in angle brackets, < >.

FIGURE 5-2

ST	int char if else while return
	break , ; [ ] ( ) <skipst>
ENTER	int char ( ) , ;
VALLOC	( )
ASGN	=
RELN	<= >= == != > <
EXPR	+ -
TERM	* / %
FACTOR	( ) MC <const> <symname>
SKIPST	if else while [(;

End of FIGURE 5-2

The statement analyzer routines communicate with each other by using the tiny-c stack almost exclusively. What is done with the contents of the stack is determined by the position of the cursor and, in the case of FACTOR and VALLOC, the setting of the global variables FNAME and LNAME. In the following descriptions, the term "matches" means the routine "recognizes and advances the cursor over the text of", while the term "evaluates" means the routine actually "computes the value of".

5.9.1 ST

Arguments: None
Results: None
Errors: Program segment encountered which is not a tiny-c statement.
Action: Matches a tiny-c statement. Calls VALLOC to enter global variables in variable table, ASGN to evaluate relational expressions and ST to evaluate components of compound statements. Uses SKIPST to skip statements which do not need to be evaluated.

5.9.2 VALLOC

Arguments: A class value and a value of that class.
Results: Address of first byte of allocated space.
Errors: Invalid symbol name or invalid array size defining expression.
Action: Matches a variable declaration and enters its name and attributes in the variable table by calling NEWVAR. The initial value of the variable name is the passed value of that class. In the case of arrays, evaluates the expression defining the size of the array.

5.9.3 ASGN

Arguments: None
Results: Success flag
Errors: None
Action: Matches an expression. In the case of an assignment operator calls EQ to make the assignment.

5.9.4 RELN

Arguments: None
Results: None
Errors: None
Action: Matches an expression and, if it is followed by a relational operator, matches another expression and evaluates the relational operator.

5.9.5 EXPR

Arguments: None
Results: None
Errors: None
Action: Matches an optional unary plus or minus followed by one or more terms separated by binary pluses or minuses. If any operators are matched, the expression is evaluated.

5.9.6 TERM

Arguments: None
Results: None
Errors: None
Action: Matches one or more factors separated by multiplication, division, or remainder operators. If any operators are matched, the term is evaluated.

5.9.7 FACTOR

Arguments: None
Results: None
Errors: Expressions without balanced parentheses; class 0 variables with subscripts; undeclared variable names, or unrecognized statement syntaxes.
Action: Matches constants, variable names including function calls, and expressions enclosed in parentheses. Evaluates constants and variables, and uses ENTER to evaluate function calls.

5.9.8 ENTER

Arguments: An address
Results: None
Errors: Number of arguments in function call not equal to number of arguments in function definition.
Action: Saves the cursor, allocates a layer in the function table by calling NEWFUN, calls SETARG to assign function argument values to function local variables, sets cursor to the passed address, matches parameter declarations and assigns them space and values using SETARG, pops all arguments off the stack, and calls ST. Upon return from ST, restores the cursor, and deallocates the layer from function table by calling FUNDONE.

5.9.9 SETARG

Arguments: The address of an entry in the tiny-c stack and a class value.
Results: None
Errors: None
Action: Converts the tiny-c stack entry to an actual and calls VALLOC with this value and the passed class value to enter an argument in the variable table as a local variable with an initial value equal to the value passed in the tiny-c stack.

5.9.10 SKIPST

Arguments: None
Results: None
Errors: Cursor runaway
Action: Advances the cursor to the end of a tiny-c statement and beyond any remarks which appear at the end of the statement. Leading remarks are also skipped.

5.9.11 LINK

Arguments: None
Results: None
Errors: Unidentifiable global declaration or function name not succeeded by a left bracket.
Action: Allocates a new layer of the variable table. Enters global variable names, including all function names, in the variable table. If an "endlibrary" statement is encountered, a second new layer is allocated and the entering of global variable names continues.

5.10 Standard Machine Calls

These Machine Calls are furnished with the tiny-c interpreter. They are always loaded and available for use. They are called as described in Section 2.10. The function number is the LAST argument. Thus MC 1 is called like this:

```
MC 'x', 1
```

Where no results are indicated, a 0 is returned as the value of the MC.

5.10.1 MC 1 ... PUTCHAR

Arguments: A character value
Results: None
Errors: None
Action: The character is transmitted to the console terminal.

5.10.2 MC 2 ... GETCHAR

Arguments: None
Results: A character value
Errors: None
Action: A character is retrieved from the console terminal and returned as the value of the function. The input port from the console terminal is cleared to receive another character.

5.10.3 MC 3 ... FOPEN

Arguments: A mode, filename, file size, and channel number.
Results: An open status indicator
Errors: None
Action: Same as fopen in the standard library, see Section 2.9.1.

5.10.4 MC 4 ... FREAD

Arguments: Pointer and a channel number
Results: A get status indicator
Errors: None
Action: Same as fread in the standard library, see Section 2.9.1.

5.10.5 MC 5 ... FWRITE

Arguments: Two pointers and a channel number
Results: A put status indicator
Errors: None
Action: Same as fwrite in the standard library, see Section 2.9.1.

5.10.6 MC 6 ... FCLOSE

Arguments: A channel number
Results: None
Errors: None
Action: Same as fclose in the standard library, see Section 2.9.1.

5.10.7 MC 7 ... MOVEBL

Arguments: Two pointers and an integer
Results: None
Errors: None
Action: Same as movebl in the standard library, see Section 2.9.1.

5.10.8 MC 8 ... COUNTCH

Arguments: Two pointers and a character value
Results: The number of appearances of the character between the two pointers inclusively.
Errors: None
Action: Same as countch in the standard library, see Section 2.9.1.

5.10.9 MC 9 ... SCANN

Arguments: Two pointers, a character value, and an integer
Results: A pointer to the last character examined
Errors: None
Action: Same as scann in the standard library, see Section 2.9.1.

5.10.10 MC 10 ... INTERRUPT

Arguments: None
Results: None
Errors: None
Action: A processor stop is executed, or a return to the processor's operating system.

5.10.11 MC 11 ... ENTER

Arguments: Four pointers, called FACTS, START, FIRST, and LAST in that order.
Results: None
Errors: None
Action: An application program is started. The text of the program is from FIRST to LAST inclusive. LAST should point to a statement terminator; a carriage return is recommended. START points to

a character string, which is a tiny-c statement. The statement can be within the program text, but need not be. FACTS points to a four-byte data area where facts are returned regarding why the application program stopped. The first two bytes of the area are the error code, the next two, the value of CURSOR when the program stopped.

The program text is linked (See Section 5.9.11 LINK). This causes a new layer of variables to be allocated for the variables defined by the link process. This layer is defined as the new Current Global area (See Sections 5.2, 5.3, 5.4). Another layer is allocated for the first layer of locals needed by the application.

The application level (global variable APPLVL) is incremented (See Section 4.3.3).

Then, if no errors occurred in these preliminary actions, the statement at START is executed. It has access to the newly defined globals. It follows that, if the statement is a function call to a program defined in the application text, that program will be called.

Upon completion, all global variables affected by MC 11 are restored to their previous values. If an error occurred in the application, that fact is captured in the bytes referenced by FACTS. The global variable ERR is set back to 0, reflecting the fact that no error has occurred in the program that called MC 11. The Current Global area and the Current Function layer are restored to their previous values. Several other pointers used in the interpretation process are restored, and the invoking function is resumed.

Values from the application are allocated from LAST+1 up.

5.10.12 MC 12 ... CHRDY (8080)

Arguments: None
Results: A character value
Errors: None
Action: Same as chrdy in the standard library, see Section 2.9.1.

5.10.13 MC 13 ... PFT

Arguments: Two pointers
Results: None
Errors: None
Action: Same as pft in the standard library, see Section 2.9.1.

5.10.14 MC 14 ... PN

Arguments: An integer
Results: None
Errors: None
Action: Same as pn in the standard library, see Section 2.9.1.

5.11 Other Tools

5.11.1 ALPHA

Arguments: An address
Results: A flag signalling valid or invalid alphabetic character.
Errors: None
Action: If the content of the address is a valid alphabetic character, the valid flag is returned. Otherwise the invalid flag is returned.

5.11.2 ALPHANUM

Arguments: An address
Results: A flag signalling valid or invalid alphanumeric character.
Errors: None
Action: If the content of the address is a valid alphanumeric character, the valid flag is returned. Otherwise the invalid flag is returned.

5.11.3 ATOI

Arguments: An address
Results: An integer value
Errors: None
Action: Starting at the argument, converts the string of numeric characters in sequentially higher addresses to the corresponding integer value. The string of numeric characters may be preceded by leading blanks and immediately preceded by a minus sign. Evaluation is terminated by the first non-numeric character. If no numeric characters are encountered, a value of 0 is returned.

5.11.4 BZAP (8080)

Arguments: Two memory addresses, the second larger than the first, and a one-byte value.
Results: None
Errors: None
Action: Like ZERO (see Section 5.11.14), except the one-byte value is put into memory instead of zeroes.

5.11.5 CEQ

Arguments: Two addresses
Results: A flag signalling match or no match
Errors: None
Action: Tests if the character string at the first address is a leading substring of the character string at the second address. Signals a match if it is, otherwise no match. The ends of the strings are signalled by a null byte.

5.11.6 CEQN

Arguments: Two addresses and an integer, n
Results: A flag signalling match or no-match
Errors: None
Action: Tests if the contents of the n sequentially higher addresses starting at the first address, are equal to the contents of the n sequentially higher addresses starting at the second address. Signals a match if all contents are equal.

5.11.7 CTOI (PDP-11)

Arguments: An address
Results: An integer value
Errors: None
Action: Returns contents of the address as an integer value.

5.11.8 ESET

Arguments: Error number
Results: None
Errors: None
Action: If and only if ERR is 0, the global variable ERR is set to the error number, and the global ERRAT to the value of CURSOR. Otherwise, no action is taken.

5.11.9 MOVN (PDP-11)

Arguments: Two addresses and an integer, n
Results: None
Errors: None
Action: Copies the contents of n sequentially higher addresses starting at the first address to the n sequentially higher addresses starting at the second address.

5.11.10 NUM

Arguments: An address
Results: An integer value
Errors: None
Action: If the content of the address is a numeric character, its integer value is returned. Otherwise, -1 is returned.

5.11.11 RET and RETS (PDP-11)

Arguments: None
Results: None
Errors: None
Action: RET restores registers 2, 3, 4, and 5 and branches to the address that was in 2(R5) before R5 was restored. When used in conjunction with SAVE, and JMPed to in place of an RTS subroutine return, subroutine calling can be recursive. RETS restores ONLY register 5, but is otherwise the same as RET.

5.11.12 SAVE (PDP-11)

Arguments: None
Results: None
Errors: None
Action: Saves registers 2, 3, 4, and 5 on the processor stack. Leaves R5 pointing to the saved value of R5. Arguments passed on the processor stack can be accessed as 4(R5), 6(R5), etc. SP, R2, R3, and R4 can be used freely. When used in conjunction with RETURN, subroutine calling can be recursive.

5.11.13 TCTOI (PDP-11)

Arguments: Two addresses
Results: An integer value
Errors: None
Action: Returns the contents of two bytes of memory as an integer.

5.11.14 ZERO (8080)

Arguments: Two memory addresses, the second larger than the first.
Results: None
Errors: None
Action: Memory is cleared from the first address to the second, inclusive.

5.12 16-Bit Arithmetic Tools (8080)

5.12.1 DNEG

BC <--- - (BC).

(Read this as BC is replaced by minus the contents of BC.) Uses A.

5.12.2 DENEG

DE <--- - (DE).

Uses A.

5.12.3 HLNEG

HL <--- - (HL).

Uses A.

5.12.4 BCRS

BC <--- (BC) >> 1.

(Read this as BC is replaced by the contents of BC right-shifted one bit.) High bit of B is replaced by a 0. Z is set if an only if result is 0. Uses A.

5.12.5 DELS

DE <--- (DE) << 1.

E's low bit <--- 0. Z is set if and only if the result is 0. Uses A.

5.12.6 RDEL

DE <--- (DE) << 1.

E's low bit <--- CY. Z is set if and only if the result is 0. Uses A.

5.12.7 HLLS

HL <--- (HL) << 1.

Low bit of L <--- 0.

5.12.8 DADD

DE <--- (DE) + (BC).

Flags set as in DSUB. Uses A.

5.12.9 DSUB

DE <--- (DE) - (BC).

Z is set if and only if the result is 0. CY is set if and only if the result is negative (high order bit of D becomes 1). Uses A.

5.12.10 DMPY

DE <--- (DE) * (BC).

Uses all registers.

5.12.11 DDIV

DE <--- (DE)/(BC).
HL <--- (DE) % (BC).

Uses all registers.

5.12.12 DREM

DE <--- (DE) % (BC).
HL <--- (DE)/(BC).

Uses all registers.

5.12.13 DCMP

The flags Z, S, and CY are set by usual 8080 conventions depending on the result of (DE) - (BC). Uses A. No other registers are changed.

VI. INSTALLING tiny-c ON YOUR 8080 SYSTEM

If you have an installed load-and-go version of tiny-c, you can skip this chapter. If you have an uninstalled version, then you must follow the installation procedure given here. It is a technical job. You must have an understanding of machine language, terminal input/output, and file input/output techniques. You must know the details of your operating system's terminal and file input/output subroutines (if they exist) or be ready to design and implement them if they do not. If you have this know-how already, the installation should take two or three days. Otherwise, the job may take some weeks.

The uninstalled tiny-c lacks seven subroutines for doing terminal and file input/output. The specifications for these routines are given in Section 6.1, but they must be coded by the installer. An installation vector in tiny-c contains jumps to these seven subroutines along with other installation dependent data. The addresses of the seven subroutines must be placed in these jump instructions.

Altogether, there are nine steps to installing a complete tiny-c on your 8080. They are:

1. Implement and test the four file interface routines and three terminal interface routines.

2. Load and relocate tiny-c.

3. Load the interface routines.

4. Link tiny-c to the interface routines using the installation vector.

5. Modify the other installation vector elements if necessary.

6. Write the results to your mass storage.

7. Test the results so far.

You now have a raw tiny-c which can execute programs, but not prepare them or modify them. To develop tiny-c programs, you may use either your own editor or the Program Preparation System (PPS) provided with tiny-c. To use the PPS,

6'. Load the results of step 6, if they're not already loaded.

8. Load tc.pps into the tiny-c program space. This file contains the PPS, a tiny-c program to prepare, test, and edit other tiny-c programs.

9. Write the results (including tc.pps) to mass storage.

You now have a tiny-c Program Preparation System.

The rest of this chapter expands on the nine steps. Section 6.10 gives suggestions on what to do if things go wrong. We urge you to carefully check each step of the installation before proceeding to the next step.

6.1 STEP 1 -- The Interface Routines

There are seven i/o interface routines: three to the terminal, and four to the file system. These must be implemented by the installer. In the happiest case, some or all exist already in your operating system. In an extreme case, you may have to code them yourself. Most likely, your operating system has subroutines that perform all the needed functions. All you will need to do is provide a little bit of code to adapt the seven interfaces to these operating system routines. An example of this is given later in Section 6.1.2.

Each routine is CALLed by tiny-c. Each must return control when finished by using the RET instruction.

6.1.1 Specification of the 8080 Terminal Interface

INCH

No arguments. This subroutine pauses until a character is available from the keyboard. It returns the ASCII character in register A.

CHRDY

Tests whether or not a character is ready from the terminal. It does NOT WAIT for a character. If no character is ready, a 0 is returned in A, and the Z flag is set. If a non-null character is ready, a copy of the character is returned in A, and the Z flag is not set. If a null character is ready, then a 1 is returned in A, and the Z flag is not set. Note that the character is not "read". To read the character (i.e., to remove it from the buffer, clearing the way for another character) INCH must be called. The fact that the ready flag happens to be a copy of the character should not be confused with the act of reading the character.

OUTCH

The A register contains an ASCII character. It is transmitted to the terminal. Nothing is returned. A, DE, and HL must not be modified.

6.1.2 Specification of the 8080 File Interface

FOPEN	HL	points to a character string of any length terminated by a null byte. This is the file name.
	DE	is an integer, specifying the file size in bytes. This is used to allocate space for newly created files.
	BC	is an integer, a unit (or device, or channel) number.
	A	1 for read, 2 for write access.

Prepare to read or write a file. If your OS needs to know whether this open is for reading or for writing, this infor-

mation is signalled in A. Otherwise, A can be ignored. If your OS supports multiple units, BC is a unit number. Otherwise it can be ignored. You may assume BC is a small integer (1, 2, 3, but not 197, 2727, 4793). Thus it can be used to address a file table if your design needs one. You may assume the unit designated in BC is not in use, i.e., if it was once opened it was subsequently closed before this call to FOPEN. If your OS allocates space for a file, and if A == 2, then DE can be used for this allocation. Otherwise, DE can be ignored. In any case DE is garbage if A == 1. If your OS supports file names, HL points to the beginning of a character string with the file name. Otherwise HL can be ignored. A result code must be returned in A and in the Z flag. 0 means FOPEN was successful. Nonzero means a problem has occurred. If a problem occurs, FOPEN must advise the user of the problem, because tiny-c will not.

FREAD	HL	points to first byte of the memory area to be read into.
	BC	is a unit number

One record is read from the file opened on unit BC. The record may be any size from 1 to 256 bytes, according to what was written. A result code must be returned in register A and in the Z flag. On a successful read A must be set to 0, and Z set, and the number of bytes read is returned in DE. HL is left unchanged. A result code of -1 in A, and Z reset, means an end-of-file was read, or an attempt was made to read beyond the file's end. This result is normal and must not cause fatal consequences, e.g., aborting tiny-c. If Z is reset and A is anything except 0 or -1, it means an error has occurred.

FWRITE	HL	points to first byte.
	DE	points to last byte.
	BC	is a unit number.

The bytes from (HL) to (DE) inclusive are to be written as a record on the file opened on unit (BC). You may assume (HL)<=(DE) and the length (DE)-(HL)+1, is less than or equal to 256. You must write the record so that the length can be determined by FREAD. If units are not supported, BC can be ignored. The result code returned in register A is 0 for a successful write and 1 otherwise.

FCLOSE BC is a unit number.

The file (connected to unit BC, if units are supported) is closed. You may assume no further reads or writes to this file or this unit will occur, unless named in a subsequent FOPEN. On a tape system, an end-of-file record is written.

6.1.3 File System Implementation Options

The file system has been specified to be as universally implementable as possible. Enough information and opportunity for control have been given to link to a sophisticated file manager. But the interface is uncomplicated enough to utilize even a very simple cassette recording system.

If you already have a tape operating system (TOS) or a disk operating system (DOS), you may have to implement only a small amount of linkage code. The operating system (OS) will do all the work.

FOPEN: If your OS has a file opening ability, you can connect with it here. If it needs to know whether the open is for read or write, the information is available. The file name is available if your OS uses file names. You may have to write code to modify the format and register assignments of this data before invoking your OS's file open. Units are for OSs supporting more than one open file.

If your OS doesn't have a file open, then FOPEN just returns 0. If reads and writes require file names, then FOPEN can simply capture the file name. An example of this is shown below.

FREAD: For cassette storage, FREAD should start the motor, read one record, then stop the motor. It must determine the number of bytes in the record, and return this number in DE.

FWRITE: Must record the record's length, probably in a record header (an "invisible" byte at the record's beginning). Some OSs do this already. FWRITE should also stop and start the motor for a cassette system.

FCLOSE: For a TOS, FCLOSE is where you write an end-of-file. Note that FCLOSE will be called at the end of a read-only access (A == 1 in FOPEN). You may need to make end-of-file

writing conditional on having a write access (A == 2 in FOPEN). On some file systems, such as a cassette system, an unconditional writing of an end-of-file in FCLOSE is satisfactory. For a DOS, FCLOSE is where you flush buffers and update the directory. Most DOSs have a close file routine to do this.

The tiny-c Program Preparation System never uses more than one file at a time and it always uses unit 1. Many applications will never use more than one file at a time. If your OS can only deal with one file, then ignore the unit number in BC; but, if you can handle multiple files, this number is the "key" that ties opens to reads, writes, and closes. You can assume it will be a small positive number. Thus you can use it, for example, to index into a table of currently open files. Or you can pass it on to your OS routines, if they use a similar unit, channel, or device number concept.

Note that returning an error code in A and Z is NOT an option. It must be done. Suppose there is no way in your FOPEN code to tell if an error occurred. Then always return 0. But don't return garbage. Also FREAD MUST return -1 for end-of-file. FWRITE MUST be able to write small (<256) records, and FREAD MUST return the number of bytes in the record in DE. tiny-c relies on this data for correct behavior.

This specification is a minimum specification. Your OS probably implements much more. For example, suppose your OS has a file write that can output large arrays. You can use this to implement FWRITE. tiny-c just won't make use of the full power of your file write. We have tried to make this specification a common denominator to all or most operating systems.

6.1.4 Example of a tiny-c Cassette File System Interface

Suppose your OS uses a cassette interface, and supports an array write, and a block read with filenames and a choice of recording modes (speeds and/or formats), with these specs:

TWRITE

Parameters

HL points to eight-byte filename
BC points to first byte of array
DE is length of array
A is recording mode (0,1,2,3)

Results

Writes array of (DE) bytes starting at (BC). Of the modes, 3 is the highest performance mode, 0 the lowest. Writes zero or more full blocks, and one partial block if needed. Starts and stops motor. Calculates and writes checksum. Writes block length in header of each block. Returns no errors, because it assumes recorder is working, etc. If DE is 0, a record of length 0 is written and can be read.

TREAD

Parameters

HL points to eight-byte filename
BC points to first byte of array
A is recording mode

Results

One block is read into memory starting at (BC). DE is set to the number of bytes read. (A) must be the same as when the record was written. Reading stops and A is set to -1 on checksum error. Otherwise a 0 is returned in A.

The problem is to realize AT LEAST the specs of FOPEN, FREAD, FWRITE and FCLOSE using TREAD and TWRITE.

The "T" routines require a mode. The "F" package has no provision for one. The simplest solution is to pick a mode, and always use it. Pick the highest performance mode consistent with your hardware.

The "T" routines have no end-of-file, but FREAD must detect one. We can use the zero length record as an end-of-file. FCLOSE will write such a record, and FREAD will test for this case.

The "T" routines require the filename at read/write time, but the "F" routines don't give it then. So the FOPEN routine will capture the filename in an eight-byte array. In

tiny-c, filenames can be any size. So FOPEN will truncate the name if it is longer than eight bytes, and pad it with blanks if shorter. TREAD and TWRITE then use this captured filename.

The registers and numbers aren't quite the same for the "T" and "F" routines. FWRITE (for example) gives the first byte's address in HL, and the last in DE. TWRITE wants the first in BC, and the array LENGTH in DE. The result code returned in A by TREAD is not what is expected by FREAD. All this can be resolved by some arithmetic and register moves.

The "T" routines don't support multiple files, so unit (in BC in the "F" routines) is ignored. Also at FOPEN time, the read/write switch in DE is ignored.

This example illustrates that implementing the file system interface routines on your computer may not require writing a file management system. Rather, you must carefully plan, and then write, a small amount of code which translates between the tiny-c file system specifications and the file system in your OS.

This example also shows that even though your OS doesn't have the four functions, open, read, write, and close, you can still implement the "F" routines. open and close should be viewed as an opportunity to arrange something, rather than a requirement to do something. Your goal is to realize at least the minimum specifications of the file system.

It is wise to thoroughly test the terminal and file subroutines to see that they conform to the specifications given. Then save them on your mass storage and proceed to Step 2.

6.2 STEP 2 -- Load and Relocate tiny-c

As furnished, an uninstalled tiny-c has two files:

```
tiny-c
tc.pps
```

Step 2 deals with the first file, tiny-c.

6.2.1 Reading the tiny-c File

With your machine-readable copy of tiny-c are instructions on how to read it. Follow these instructions until you have a good copy of the file "tiny-c" loaded into memory. Load it at 2000 hex, even if you intend to install it at another origin.

6.2.2 Address Adjustment

As delivered, tiny-c has an origin of 2000 hex. Suppose you intend to install it with an origin of 100 hex. First, load it as described in Section 6.2.1. Then all the addresses in tiny-c must be adjusted by the difference between 2000H and 100H, namely -1F00H.

The program in Figure 6-1 does this.

Some three-byte instructions have data, not addresses. For example:

```
LXI     D,-C
LXI     H, 0
```

In tiny-c these data are always of small magnitude. Addresses are always of the form 2XXX or 3XXX (hex). So the relocation program tests for this, and does not adjust data.

First, load tiny-c at 2000H. Then load the ADJADDR program including its subroutines and data tables. Load this anywhere outside the tiny-c area (2000-3100 hex). It is shown compiled at 6000H. If this area is available, use ADJADDR as compiled in Figure 6-1. Otherwise, its 20 addresses can be relocated by hand. Then ADJADDR can be loaded somewhere else.

Set BC to the adjustment: -1F00H = E100H. In general it is:

```
yourorigin - 2000H.
```

Start execution at ADJADDR. When the computer halts, the addresses have been adjusted.

Examine memory to see if the addresses make sense. The listings of tiny-c are in Appendix A. Is the first address correctly relocated? Is the last? If so the rest are probably correct.

The address at 2D3A is (-BUFF-1). The relocator will not modify this address correctly. It must be manually modified by subtracting the adjustment. For example:

D267 - E100 = D267 - (-1F00) = D267 + 1F00 = F167

Manually enter 67 into 2D3A, and F1 into 2D3B.

Record the results (2000-3100 hex) and reload them at yourorigin. Examine the addresses again to see if they make sense.

FIGURE 6-1

```
6000                    0000    ;Adjusts the address of tiny-c, version 80-01-01. The table
6000                    0010    ;  TCADDS separates data from program areas.  However
6000                    0020    ;  certain data (like error codes) which fortuitously
6000                    0030    ;  look like one-byte 8080 instructions are not shown
6000                    0040    ;  in the table.  So if you change the error codes, the
6000                    0050    ;  table must be changed.
6000                    0060    ;
6000 21 06 60           0070    ADJADDR LXI     H,TCADDS
6003 C3 80 60           0080            JMP     ADJUST
6006 00 20              0090    TCADDS  DW      2000H
6008 08 20              0100            DW      2008H
600A 2B 20              0110            DW      202BH
600C 33 20              0120            DW      2033H
600E C0 20              0130            DW      20C0H
6010 C5 23              0140            DW      23C5H
6012 C7 23              0150            DW      23C7H
6014 04 24              0160            DW      2404H
6016 0F 24              0170            DW      240FH
6018 D1 24              0180            DW      24D1H
601A E1 24              0190            DW      24E1H
601C 68 25              0200            DW      2568H
601E F3 27              0210            DW      27F3H
6020 27 28              0220            DW      2827H
6022 2C 28              0230            DW      282CH
6024 88 28              0240            DW      2888H
6026 8F 28              0250            DW      288FH
6028 1E 2A              0260            DW      2A1EH
602A 25 2A              0270            DW      2A25H
602C 7A 2A              0280            DW      2A7AH
602E 80 2A              0290            DW      2A80H
6030 3A 2C              0300            DW      2C3AH
6032 3C 2C              0310            DW      2C3CH
6034 FF 2C              0320            DW      2CFFH
```

```
**********************************
* See Appendix A for changes in *
* Relocation code  for  version *
* 80-01-02.                     *
**********************************
```

```
6036 12 2D          0330          DW      2D12H
6038 97 2D          0340          DW      2D97H
603A C0 2D          0350          DW      2DC0H
603C 09 2E          0360          DW      2E09H
603E 48 2E          0370          DW      2E48H
6040 86 2E          0380          DW      2E86H
6042 89 2E          0390          DW      2E89H
6044 DF 30          0400          DW      30DFH
6046 6B 25          0410          DW      256BH
6048 F1 27          0420          DW      27F1H
604A 00 00          0430          DW      0       ;end of table
604C                0440          DS      52      ;to expand the table
6080                0630    ;
6080                0640    ;
6080                0650    ;
6080                0660    ;Adjusts the addresses of a program so it will run at
6080                0670    ;  a different origin.  (HL) points to a table of
6080                0680    ;  FROM-TO addresses giving the range where adjustments
6080                0690    ;  are to be applied.  (BC) is the adjustment to be
6080                0700    ;  added to the addresses.  Only addresses in the range
6080                0710    ;  2000 through 3fff are adjusted.  A 0000 FROM address
6080                0720    ;  signals the end.
6080 5E             0730    ADJUST  MOV     E,M     ;from address
6081 23             0740            INX     H
6082 56             0750            MOV     D,M
6083 23             0760            INX     H
6084 7B             0770            MOV     A,E     ;test if done
6085 B2             0780            ORA     D
6086 CA 96 60       0790            JZ      HALT
6089 D5             0800            PUSH    D
608A 5E             0810            MOV     E,M     ;to address
608B 23             0820            INX     H
608C 56             0830            MOV     D,M
608D 23             0840            INX     H
608E E3             0850            XTHL            ;from -> HL, pointer -> stack
608F CD 9A 60       0860            CALL    ADJFRTO ;adjust this range
6092 E1             0870            POP     H       ;pointer -> HL
```

```
6093 C3 80 60          0880            JMP     ADJUST
6096 76                0890    HALT    HLT
6097 C3 96 60          0900            JMP     HALT
609A                   0910    ;Adjusts address of instructions from (HL)...(DE)
609A                   0920    ;  inclusive by adding (BE).  An address is modified
609A                   0930    ;  only if it is from 2000 to 3FFF inclusive.
609A 7B                0940    ADJFRTO MOV     A,E     ;done if HL > DE
609B 95                0950            SUB     L
609C 7A                0960            MOV     A,D
609D 9C                0970            SBB     H
609E D8                0980            RC
609F CD CA 60          0990            CALL    INSIZE  ;Get size of instruction at
60A2 FE 01             1000            CPI     1       ;  (HL)
60A4 CA BD 60          1010            JZ      SKIPONE
60A7 FE 02             1020            CPI     2
60A9 CA BC 60          1030            JZ      SKIPTWO
60AC 23                1040            INX     H       ;3 byter, point to hi byte
60AD 23                1050            INX     H       ;  of its address
60AE 7E                1060            MOV     A,M     ;test for page 2x or 3x
60AF FE 20             1070            CPI     20H
60B1 DA BD 60          1080            JC      SKIPONE
60B4 FE 40             1090            CPI     40H
60B6 DC C1 60          1100            CC      ADJONE  ;adjust this one
60B9 C3 BD 60          1110            JMP     SKIPONE
60BC 23                1120    SKIPTWO INX     H
60BD 23                1130    SKIPONE INX     H
60BE C3 9A 60          1140            JMP     ADJFRTO
60C1                   1150    ;Adjusts one address.  (HL) points to its hi byte.
60C1 2B                1160    ADJONE  DCX     H       ;lo byte
60C2 79                1170            MOV     A,C
60C3 86                1180            ADD     M
60C4 77                1190            MOV     M,A
60C5 23                1200            INX     H       ;hi byte
60C6 78                1210            MOV     A,B
60C7 8E                1220            ADC     M
60C8 77                1230            MOV     M,A
60C9 C9                1240            RET
```

```
60CA                    1250  ;Determine size of instruction at (HL).  Restores all
60CA                    1260  ;  registers except A, where it returns a 1, 2, or 3.
60CA E5                 1270  INSIZE  PUSH    H
60CB 7E                 1280          MOV     A,M     ;inst byte
60CC 21 00 61           1290          LXI     H,THREEBT
60CF CD EA 60           1300          CALL    LOOKUP
60D2 CA D9 60           1310          JZ      IN2
60D5 3E 03              1320          MVI     A,3     ;is three bytes
60D7 E1                 1330          POP     H
60D8 C9                 1340          RET
60D9 21 17 61           1350  IN2     LXI     H,TWOBT
60DC CD EA 60           1360          CALL    LOOKUP
60DF CA E6 60           1370          JZ      INONE
60E2 3E 02              1380          MVI     A,2     ;is two bytes
60E4 E1                 1390          POP     H
60E5 C9                 1400          RET
60E6 3E 01              1410  INONE   MVI     A,1     ;is one byte
60E8 E1                 1420          POP     H
60E9 C9                 1430          RET
60EA                    1440  ;One-byte table lookup subroutine.  Byte is in A.
60EA                    1450  ;  (HL) points to beginning of table, ended with a
60EA                    1460  ;  null byte.  Returns NZ on found, Z on found.  All
60EA                    1470  ;  registers, including A, are restored.
60EA E5                 1480  LOOKUP  PUSH    H       ;only flags can be changed.
60EB C5                 1490          PUSH    B
60EC 47                 1500          MOV     B,A     ;lookup byte -> B
60ED 7E                 1510  LK2     MOV     A,M     ;next byte of table
60EE B7                 1520          ORA     A
60EF CA FC 60           1530          JZ      LK4     ;end of table, byte not found
60F2 90                 1540          SUB     B
60F3 CA FA 60           1550          JZ      LK3     ;found
60F6 23                 1560          INX     H
60F7 C3 ED 60           1570          JMP     LK2
60FA DE 01              1580  LK3     SBI     1       ;A has 0, so this sets NZ
60FC 78                 1590  LK4     MOV     A,B     ;this restores A without
60FD C1                 1600          POP     B       ;  affecting flags
60FE E1                 1610          POP     H
```

```
60FF C9              1620             RET
6100 C3              1630   THREEBT   DB      0C3H
6101 C2              1640             DB      0C2H
6102 CA              1650             DB      0CAH
6103 D2              1660             DB      0D2H
6104 DA              1670             DB      0DAH
6105 F2              1680             DB      0F2H
6106 FA              1690             DB      0FAH
6107 CD              1700             DB      0CDH    ;calls
6108 C4              1710             DB      0C4H
6109 CC              1720             DB      0CCH
610A D4              1730             DB      0D4H
610B DC              1740             DB      0DCH
610C F4              1750             DB      0F4H
610D FC              1760             DB      0FCH
610E 22              1770             DB      022H    ;xHLD
610F 2A              1780             DB      02AH
6110 32              1790             DB      032H    ;xDA
6111 3A              1800             DB      03AH
6112 01              1810             DB      01H     ;LXI
6113 11              1820             DB      11H
6114 21              1830             DB      21H
6115 31              1840             DB      31H
6116 00              1850             DB      0       ;end of three-byters
6117 06              1860   TWOBT     DB      06H     ;MVI
6118 0E              1870             DB      0EH
6119 16              1880             DB      16H
611A 1E              1890             DB      1EH
611B 26              1900             DB      26H
611C 2E              1910             DB      2EH
611D 36              1920             DB      36H
611E 3E              1930             DB      3EH
611F C6              1940             DB      0C6H    ;xyI
6120 CE              1950             DB      0CEH
6121 D6              1960             DB      0D6H
6122 DE              1970             DB      0DEH
6123 E6              1980             DB      0E6H
```

```
6124 EE          1990          DB      0EEH
6125 F6          2000          DB      0F6H
6126 FE          2010          DB      0FEH
6127 DB          2020          DB      0DBH    ;in/out
6128 D3          2030          DB      0D3H
6129 00          2040          DB      0       ;end of two-byters
```

End of FIGURE 6-1

6.3 STEP 3 -- Load the Interface Routines

In Section 6.1.4 you saved these routines on your mass storage. Now reload them, if they are not still in memory.

6.4 STEP 4 -- Link tiny-c to the Interface Routines

The tiny-c transfer vector starts at ORG+0A hex. There are seven JMP statements. Fill their address fields with the addresses of the seven interface routines:

```
ORG+0A      JMP      INCH
ORG+0D      JMP      OUTCH
ORG+10      JMP      CHRDY
ORG+13      JMP      FOPEN
ORG+16      JMP      FREAD
ORG+19      JMP      FWRITE
ORG+1C      JMP      FCLOSE
```

Note that the addresses given are for the JMP instruction; C3 hex must be in ORG+0A, ORG+0D, ORG+10 ... The next two bytes are the address field.

6.5 STEP 5 -- Modify the Other Installation Vector Elements

Sections 6.5.1, 2, and 3 describe those additional entries in the installation vector that are essential to an installation. There are several optional entries that give additional facilities. These are described in Section 6.5.4.

6.5.1 Upper Case Option

tiny-c is at its best with a full ASCII keyboard and display. As delivered, it assumes full upper and lower case operation. In particular, the keywords in the keyword table are all lower case. When you use lower case keywords, you

will see it makes programs more readable, more pleasant to the eye.

If you have full upper/lower case for both your keyboard and display, skip this subsection.

If your keyboard is upper case only, then your tiny-c programs will have upper case keywords ("WHILE" instead of "while", etc.). There is one situation to watch out for: if your keyboard happens to be full upper/lower, but your display is upper case only, it may seem safe to leave tiny-c in its lower case mode. The program text is entered in lower case, it runs in lower case, and it is stored in lower case. Only the display maps it to upper case, and that is readable enough. But, what if you accidentally enter "whiLe"? The program won't run. When you display it, it reads WHILE, giving you no clue to the problem. We recommend two modifications if you have this kind of terminal:

1. Implement the upper case mod, given below.

2. Also implement case fold logic in INCH. Lower case alphabetics arriving from the keyboard should be mapped to upper case by software in INCH. (See Section 6.1.1.)

This will give you a quality interface to the terminal.

To implement upper case:

The table of keywords is located at ORG+63 hex through ORG+93 hex. Modify NONZERO bytes of this table by subtracting hex 20. So 69H becomes 49H. Leave the zero bytes as 0. Starting at ORG+0D09H are four bytes that must be similarly adjusted. Finally, all alphabetics in the file "tc.pps" must be case-folded to upper case. This must be carefully done so that non-alphabetics are not modified.

6.5.2 Allocation of Memory

After tiny-c and the interface routines and operating system routines are loaded, you should have a large block of memory left over. This must be allocated into four working areas for tiny-c. They are:

	Recommended Allotment
STACK	128 bytes (decimal)
FUN	128 bytes (decimal)
VAR	15% of the remainder
PR	all the rest

The STACK is for tiny-c calculations, and should not be confused with the 8080 stack. The space allotments are for "average" programs, and can be rounded to convenient numbers. Assign the beginning addresses for the blocks, and enter these values in the locations:

```
BSTACK      ORG+36H
BFUN        ORG+3AH
BVAR        ORG+3EH
BPR         ORG+42H
```

Calculate the end addresses, and put their NEGATIVE value in the locations:

```
ESTACK      ORG+38H
EFUN        ORG+3CH
EVAR        ORG+40H
EPR         ORG+44H
```

EXAMPLE: Assume an origin of 0. The area to be allocated is from 1200H to 3FFFH. There are

```
3FFF - 1200 = 2DFFH = 11775 dec
```

bytes available. Allocate 128 for STACK and 128 for FUN. That leaves 11519. 15% of this is 1728 for VAR, leaving 9791 for PR. We will round these to convenient hex quantities.

	allot dec	rounded allot hex	begin	end+1	-end-1
STACK	128	80	1200	1280	ED80
FUN	128	80	1280	1300	ED00
VAR	1728	700	1300	1A00	E600
PR	9791	2600	1A00	4000	C000

End+1 is calculated as begin + rounded allotment. This is the next item's begin. The boxed quantities are entered as the B and E addresses for STACK, FUN, VAR and PR respectively.

	address*	contents (all hex)
	=======	========
BSTACK	36	1200
ESTACK	38	ED80
BFUN	3A	1280
EFUN	3C	ED00
BVAR	3E	1300
EVAR	40	E600
BPR	42	1A00
EPR	44	C000

*origin 0 assumed. Otherwise add your origin to the addresses shown.

Note that all addresses must actually be entered in the usual 8080 byte-reversed order. This example is NOT in that order to make the contents readable and the principles clear.

Anytime BPR is changed, PPS must be reloaded as described in Section 6.8, and the load-and-go tiny-c re-recorded as described in Section 6.9.

6.5.3 8080 Stack

If your operating system allocates an 8080 stack, then the two bytes starting at ORG+46H, must have the value 0. (The delivered tiny-c has this value.) If your system will start up tiny-c without having allocated an 8080 stack, then tiny-c will allocate it for you on a COLD START. (All tests through Section 6.8 use cold starts. Section 6.9 explains hot starts.) Determine a block of memory to use. Put its HIGHEST address into ORG+46H. Use the usual 8080 byte-reversed order.

EXAMPLE: The block from 1000H to 11FFH will be used for the stack, and tiny-c will allocate it. (The origin is 0.)

	address	contents (NOT shown byte-reversed)
	=======	========
S8080	46H	11FFH

6.5.4 Optional Entries in the Installation Vector

6.5.4.1 User Machine Call Jump Address

If you program a user Machine Call you must allocate some memory for its code. Do this by carefully accounting for all the memory space used:

1. by tiny-c from ORG through ORG+1100.
2. by your own installation code.
3. in allocating the four working areas (Section 6.5.2) and possibly the 8080 stack (Section 6.5.3).

Items 1 and 2 are probably bolted down in certain addresses of memory and not easily changed. Item 3 is easily changed. So by changing item 3 allocations you can make room for code for private (user) Machine Call code. Note that changing the allocations means PPS must be reloaded (see Section 6.8).

Load the Machine Call program in the space freed up. Now link it to tiny-c by putting an address in the jump instruction at ORG+1F hex:

```
(ORG+1F)  C3 XX XX  <--- fill in this address
```

Note that this jump instruction already has an address (not 0000). If you have an uninstalled tiny-c, it jumps to an error routine called MCESET, (see Section 2.10). Check your installation document if your tiny-c is installed, since it may already have private MCs. If so, the JMP at ORG+1F is already patched to the correct address and should not be changed. Your installation document will tell you:

1. What MC numbers are already used, and which you may use for additional MCs.
2. How to patch them in.

Finally, capture all this work by doing the procedure in Section 6.9.

6.5.4.2 Choice of Escape Character

You may halt an application by typing ASCII ESC. If this choice of character is unsuitable for your terminal it can be changed. At ORG+35 hex is the character which, when typed, causes an application halt. You can change it to any value not needed for programming or for data.

6.5.4.3 Statement Control Opportunities

tiny-c will CALL external subroutines at three points:

-- When any program is begun,
   including PPS and applications.
-- At the start of each statement.
-- When any program completes,
   including PPS and applications.

(Note that by "program" we do not mean function. We mean a main program, i.e., all of PPS, or all of an application.)

There is room for three jumps in the installation vector at ORG+22, ORG+25, and ORG+28 hex. As furnished, each has the three one-byte instructions:

```
NOP
NOP
RET
```

These instructions do nothing except assure the continued running of tiny-c.

The opportunity to link to external subroutines is provided for several purposes; for example, to create

1. a debugger, with single step
   and variable display capability,
   and even breakpoints.

2. a profiler, which counts how
   often each statement gets
   executed.

Use of this capability requires knowledge of Chapter V, plus study of the listings in Appendix A, plus imagination.

6.6 STEP 6 (and 6') -- Write to Disk Or Tape

Step 5 concluded the installation of raw tiny-c. This should be written to your mass storage before testing. If your OS allows, set it up so loading will automatically cause execution to begin at ORG.

Step 6' is simply the converse of 6. Make sure this can be done successfully before going to Step 7.

6.7 STEP 7 -- Test the Results So Far

Load tiny-c and start it at ORG. The prompter

```
>>>
```

should appear. This indicates you are in the loader.

If the prompter does not appear, check that your OUTCH routine is loaded, linked properly at OTG+0D, and working properly. If you get one > and then either garbage or nothing, check that your OUTCH restores register A. Finally, trace the code in Appendix A from ORG to the first CALL INCH in the subroutine LOADER. Check that your loaded tiny-c has good bytes on the path.

The loader has three commands:

```
.r  filename
.g
.x
```

The first reads a file containing a tiny-c program or part of a program. The .r command can be used more than once if a program has parts stored in several files.

The second command, .g, causes the load program to start at the function "main". First, the entire program is linked (see Section 5.9.11). Several errors can be detected here, indicating a bad tiny-c program or an improperly functioning file read routine (see Section 6.1.2).

The third command, .x, causes a jump to location 0, which usually returns control to your OS. The jump instruction (actually CA hex, Jump on Zero) is located at ORG+10F1 hex, and can be given another address if needed.

6.7.1 Null Program Test

The first test is to give the .g command, i.e., try to run a null program. The diagnostic

```
DONE 26 aaaaa
```

will be printed. aaaaa is the address where the program looked for "main". Its value is BPR+5, and it is printed in decimal.

6.7.2 Simple Program Test, Hand Loaded

In Section 6.5.2, memory was allocated. A value for the beginning of program space (BPR) was determined. The example derived 1A00 hex for BPR. In this test we will hand load 14 bytes, a very small tiny-c program, into this space. We will set a pointer to the first byte BEYOND the program. And we will run the program.

Load tiny-c and do a cold start (i.e., start it at ORG). When the >>> appears, interrupt it using your operating system's usual method. Then, starting at BPR+9, put these bytes into memory:

address	ASCII value	hex value
BPR+9	m	6D
BPR+A	a	61
BPR+B	i	69
.	n	6E
.	[	5B
.	M	4D
	C	43
	'	27
	x	78
	'	27
	,	2C
	1	31
	]	5D
BPR+16 hex	null	00

Now (in hex) calculate the value of -(BPR+16). Put that value in ORG+60 and ORG+61 in the usual 8080 byte-reversed order. For example, if BPR is 1A00 hex, then

```
-(BPR + 16)

 = -1A16

 = E5E9 + 1     (hex complement + 1)

 = E5EA
```

Put EA into 2060, E5 into 2061, i.e., low-order byte first.

You have now loaded a program. Be sure the 8080 stack is exactly the way it was when you interrupted tiny-c. Warm start tiny-c by starting it at ORG + 3. (If you start it at ORG, the loaded program gets erased.) The triple prompter >>> should appear. Give the command .g. The copyright message should be printed, followed by the single character 'x' on a line by itself. Next, DONE should appear, also on a line by itself. Then the triple prompter >>>. If all this happens, you have just run your first tiny-c program.

If it doesn't work don't look in your interface routines for the problem. Only your OUTCH program was used by this test, and that had to work to produce the >>> . Most likely the problem is in the setup of the test. Go over it carefully. The series BPR through BPR+8 should contain

```
[main();]
```

Does it? If not, then you forgot the cold start. If it does, check that you entered the 14 bytes correctly and calculated the negative of BPR+16 correctly. Did you use upper case for MC? It must be upper case. The bytes "main" at BPR+1, ..., BPR+4 must be exactly the same as at BPR+9, ..., BPR+C. They can be upper or lower case, but must be the same. Be sure you did not confuse BPR (the address of the program space) with ORG+42, a word in which that address must be stored.

6.7.3 Simple Program Test, File Loaded

Prepare a file with the same 14-byte program as in Section 6.7.2. Make sure the file is readable by your FREAD routine (see Section 6.1.2), and that it returns decimal 14 (hex 0E) in DE as the number of bytes read. Make sure that a second call to FREAD returns an end-of-file signal.

Now cold start tiny-c (start it at ORG). Give .r filename. The file should read into BPR+9, ..., BPR+C.

Interrupt and examine the program buffer. Its first nine bytes should be [main();]. Its next 14 bytes should be the test program. Examine ORG+60. The same end pointer you calculated for Section 6.7.2 should be there.

Resume execution of tiny-c (making sure the 8080 stack is unmodified). Give the .g command. The program should run. If it doesn't, make all the checks given in Section 6.7.2.

6.8 STEP 8 -- Prepare PPS for Loading

If you intend to use an external editor to prepare tiny-c programs, then your installation is complete. The external editor must, of course, write files compatible with your

FREAD program. And every program you prepare must have a main program called "main".

If you intend to use PPS, you must complete Step 8. PPS is a tiny-c program. You must prepare it for reading, read it, and start it just as you did the sample program in Section 6.7.3.

First PPS must be written on a file compatible with your FREAD program. PPS is the second file of your machine-readable media. It is also shown in Appendix C. With your machine-readable copy are instructions on how to read the second file. Study these until you understand them.

Write a program that reads all of PPS into memory, then (using your FWRITE and FCLOSE) writes it to your own media in your own format. Run this program. Check to see that you have a good copy. If you have a "file-examine" capability, use it to make sure the output file has a good copy of PPS, and will be readable by your FREAD. In any case, write a program using your FOPEN and FREAD to read the file. Does it return the correct number of bytes in DE for each record read? Does it detect an end-of-file correctly?

When you are sure your copy is correct, cold start tiny-c and read PPS using .r filename. Interrupt tiny-c and examine memory to see if a good copy of PPS is in BPR+9, ... Find its end. There should be a null byte (hex 00) at the very end. Calculate the negative of the address of this null byte. That value should be in ORG+60. If you find a problem, it's probably because this is the first multi-record program loaded. Check that FREAD returns 100 hex for 256-byte records. If everything seems in order, start PPS with .g.

The tiny-c copyright message should print. Then, on a line by itself, the single prompter

```
>
```

will appear. This was produced by a tiny-c statement in PPS!

If a > does not appear, a DONE message should appear:

```
DONE  ee  aaaaa
```

where ee is a decimal error code, and aaaaa is a DECIMAL (not hex) address. This is a tiny-c diagnostic, error ee, at address aaaaa. (The error codes are in Section 3.5.) Convert

the address to hexadecimal, and examine memory there to find the problem.

6.8.1 Possibly Required PPS Changes

The block size of the file management system was determined by the installer in Step 1 of the installation process. That size, less one, is coded into the writefile function of PPS in the statement

```
if(l>255)l = 255
```

(This is the statement that implements the "portability view" for PPS. See Section 2.9.2). writefile is about one-third from the top of PPS, just before the endlibrary statement.

These two integers are encoded in ASCII, so the line appears in hex as:

```
69 66 28 6C 3E 32 35 35 20 29 6C 3D 32 35 35 20 0D
i  f  (  l  >  2  5  5  sp )  l  =  2  5  5  sp ret
```

If the design block size is not 256, then the size-less-one must be put into this statement, in two separate places. The space (20 hex) at the end of each such place allows up to four digits for this.

6.8.2 Optional PPS Changes

The ASCII character CAN (meaning cancel, hex 18) is the "line kill" character. This is encoded as the decimal number 24 in the function gs, near the beginning of PPS (See Appendix C). The digits of the number are encoded in two ASCII bytes: 32 34 hex. Similarly, the ASCII character DEL (hex 7F) is the "character kill". It is encoded as decimal 127, and is written as three ASCII bytes: 31 32 37 hex. These can be changed to suit your habits, or the limitations of your keyboard.

6.9 Write the "Load-And-Go" tiny-c to Mass Storage

You can always go through the three (or four) step sequence:

```
load tiny-c
(start at ORG)
>>>.r tc.pps
>>>.g
```

to start the Program Preparation System. It is more convenient to do a single load with an auto start, if you have one. To prepare for this, write out all of tiny-c, the interface routines, any system routines they need, and 5500 bytes starting at BPR (the tc.pps text) as one file on your mass storage. Arrange an automatic execution at ORG+6, if you have a way of doing this. This is a "hot start" which starts not only tiny-c, but any system program it has loaded. In this case, the Program Preparation System will be started. Now loading (and starting) this file gets you going in only one or two steps, because the .r and .g steps are "automated".

If you implemented the stack allocation option (Section 6.5.3) and want to do a hot start, then put this program in your installation code:

```
HOTSTART    LHLD    ORG+46
            SPHL
            JMP     ORG+6
```

Then HOTSTART is your hot start address.

6.10 If Things Go Wrong

It should be clear from reading this that to install tiny-c, you must, at least, know how to use a file system and your i/o interface routines. You might even have needed to write a file system. So you're not a software novice.

If things blow up, review your i/o interface routines. Do they work as required? Test them again. Are they correctly linked? Do you have an 8080 stack? Are the addresses correctly relocated?

A crucial step is in Section 6.7 when you start tiny-c at ORG. This is the first attempt to execute tiny-c. Trace the code in Appendix A. It is a short sequence from ORG to the first CALL INCH in subroutine LOADER. If you get there, and if INCH works, you should be able to type in a line of text. Type a few characters. Then interrupt and examine BUFF. If characters are going into BUFF, and are in the right case, and if the first two are ".r", then a carriage return should cause a call on FOPEN.

Are you getting there? If so, is a 0 returned in A? Is the Z flag set? The JNZ LOADER after the CALL FOPEN should not branch! If it does you have an FOPEN problem. Next, FREAD should get called. It should return 0 with Z set on the first call. Is that happening? If so, you've read one block of tc.pps. Check BPR (ORG+42 hex). Is it where you want tiny-c programs loaded? If so, go look there and see if a block of tc.pps is really there. If it isn't, you have FREAD problems.

Let the read loop (L2) run to completion. An end-of-file should be detected by FREAD, a -1 returned, and Z not set. This causes a jump back to LOADER. Now interrupt and examine the program area. Is all of tc.pps there? Are there gaps? Are there missing bytes? If so, FREAD is not returning correct byte counts in DE. Is PROGEND pointing to the last byte of tc.pps? (PROGEND is stored negative. Write out its 16 bits. Then complement them. Then add one. That address should contain the last byte of tc.pps, which is a null byte after the last carriage return.)

If all this is correct, then the go command ".g" should cause LOADER to return to the location HOT. From then on, you're in tiny-c code, except for calls to INCH and OUTCH. If it still doesn't work, then print a hex dump of ORG through ORG+1100 hex. Sit down with a good friend and read every byte, and compare with the listings in Appendix A.

6.11 Key Points of a tiny-c Installation

In this documentation several references are made to installer-determined parameters. Any tiny-c installation should document the following key points:

1. The Storage Map parameters. These are:

   ORG
   Where private MCs can start
   Where space starts

2. Private MCs furnished. These should be documented. We recommend you edit the standard library to incorporate these MCs with descriptive names, as described in Section 2.9.

3. Line kill and character kill characters, as described in Section 6.8.2. State what they are, even if not changed.

   Similarly, define the application halt (ESCAPE) key chosen in Section 6.5.4.

4. Design decisions on the interface routines:

   -- whether or not filenames are used, and if so, how long they can be.
   -- whether or not filesize is relevant in FOPEN.
   -- whether or not multiple units are supported, and restrictions if they are.
   -- whether or not read/write access modes are required.
   -- block size.

5. Files on the machine-readable copy, their names, recording modes, and software to load them.

6. Other installation options:

   -- upper case, or upper/lower case;
   -- allocation of memory;
   -- how the 8080 stack is allocated and where.

7. PPS options, particularly the size of program space. It is also convenient to know:

   -- the exact bytes to modify for larger or smaller program space. These are the ASCII encoded decimal digits DDDD in the PPS text:

```
char ln(64), pr(DDDD)
```

   and four lines later

```
lpr = DDDD
```

   -- The exact byte address which is line zero of an application.

8. We recommend that an emergency utility be built to save your work in case of a system crash. Assuming your operating system allows you to browse through memory looking at the software corpse, it is not unlikely you can find the first and last bytes of your valuable but not yet recorded work. The utility accepts two addresses, and records everything from the first to the second (inclusive) as a file with a standard filename.

6.12 Modifying the Program Preparation System

The PPS can be used to edit itself. PPS is stored in a file called "tc.pps". This is the file produced by Step 8 of the installation procedure (Section 6.8). Read it in by using the .r command, edit it, and write the edited version using .w. Start it and test it with .main (the name of the main function is "main"). In this way you can

-- add functions to the standard library, and
-- add or modify features of PPS.

In the middle of the file is an "endlibrary" statement. This unique statement separates library symbols from global symbols. All global symbols that occur before the endlibrary statement are library symbols, and can be accessed by both

PPS and applications. All that occur after the endlibrary are globals to PPS, and are not accessible to applications. There must be one and only one endlibrary statement.

Once you are satisfied that your new version of PPS is working, go to Step 9 (Section 6.9) and re-record tiny-c with an imbedded new PPS.

6.12.1 The System Loader

To load an edited PPS, interrupt or halt your 8080. If the tiny-c interpreter is not loaded, then load it. Start it at ORG (See Section 6.2.2). This is a cold start. It erases any PPS already loaded. Then the system loader is entered. The prompter:

```
>>>
```

is issued. The system loader accepts three commands:

```
>>>.r filename
```

Reads a file from cassette or floppy disk, appending it to what was read previously. [Note: Some installations of tiny-c may not use the filename.] The set of files read using this command becomes the system level program (level 0 as defined in Section 4.3.4).

```
>>>.g
```

The system level program is started at the function named "main". main has no arguments.

```
>>>.x
```

A jump is made to the OS. See Section 6.7 for how this jump is done.

NOTE: You may use .r repeatedly to load more than one file. In this case library routines must be loaded first, and the system programs last. Within the entire set of files must be one and only one endlibrary statement partitioning the library from the system programs.

6.12.2 Debugging a New PPS

There are no editing tools in the system loader. The debugging aid is a single diagnostic: either

```
DONE
```

which means the function main completed without error; or

```
DONE  ee  aaaaa
```

indicating error number ee occurred at byte address aaaaa.

The error codes are listed in Section 3.5. The address is given in decimal, and is the memory address of the byte being interpreted when the error was detected. This is the sole clue to a problem in a new PPS. For this reason, a new or modified PPS should be debugged while operating as an application under the standard PPS, which gives better clues and allows online editing.

VII. INSTALLING tiny-c ON YOUR PDP-11 SYSTEM

As furnished, the PDP-11 version of tiny-c, tiny-c/11, is ready to use with any DEC™ monitor which handles the following programmed requests:

TTYIN
TTYOUT
LOOKUP
ENTER
READ
WRITE
CLOSE

The RT-11, RSX and RSTS monitors all satisfy this requirement. Since all device communication takes place through the tiny-c installation vector, users of CAPS-11, Paper Tape Software and IOX can assemble and install tiny-c after placing IOTs in the seven input/output entries in this vector and writing code to translate between the tiny-c peripheral specifications and IOX.

7.1 Logical Division of tiny-c/11

tiny-c/11 is divided logically into four parts: the interpreter, the interfaces, the MCs, and the Program Preparation System (PPS). The first three parts are collections of PDP-11 assembly language programs and subroutines. The last part, the PPS, is a tiny-c program together with a collection of tiny-c functions. The seven programmed requests mentioned above are found in the interface part of tiny-c/11.

™ DEC is a trademark of Digital Equipment Corporation.

The interpreter consists of:

1. a main program which initiates the tiny-c interpreter and includes

   a. assembly constants which size tiny-c

   b. global declarations

   c. tiny-c literals;

2. the tiny-c interpreter subroutines; and,

3. a data block consisting of tiny-c global variables and the installation vector.

The interfaces consist of all of tiny-c's input/output access routines and, hence, all monitor calls and programmed requests. The access routines are linked to the tiny-c interpreter through the installation vector. Users who have written their own input/output routines -- whether simple specification translation routines or full-fledged device drivers -- should place the addresses of these routines in the appropriate JMP instructions in the installation vector.

7.2 The Installation Vector

The tiny-c/11 installation vector is divided into three parts: seven input/output calls; four user option calls; and eight tiny-c/11 working area addresses.

7.2.1 Input/Output Routine Calls

The first seven entries in the tiny-c/11 installation vector are JMPs to the seven input/output routines used by tiny-c/11:

Installation Vector Entry	Subroutine Entry Point	Action
INCH	GETCHR	return a character from the console terminal right-justified (even-byte) in R0.
OUTCH	PUTCHR	transfer the character in the even-byte of R0 to the console terminal.
FLOAD	LOAD	load the file TCLOAD.TCF into the tiny-c program space. Squeeze out nulls (ASCII 000) and carriage returns (ASCII 015).
FOPEN	OPEN	open the file described by the arguments placed on the machine stack as follows: 2(SP) — 1 read, 2 write 4(SP) — address of 14-byte, fully-qualified file name 6(SP) — file size 10(SP) — channel or unit number return the status of the open in R0 as follows: 0 or greater — file opened, return size of file in bytes -1 — unit already open -2 — file not found

FREAD GETBL read a block of data from the file system according to the arguments placed on the machine stack as follows:

2(SP)	low order address of a 512-byte data buffer
4(SP)	channel or unit data is to be read from

return the status of the read in R0 as follows:

0 or greater	block read
-1	end-of-file
-2	read error

FWRITE PUTBL put a block of data to the file system according to the arguments placed on the machine stack as follows:

2(SP)	address of first byte of the block
4(SP)	address of last address of the block
6(SP)	channel or unit to receive the data

return the status of the write in R0 as follows:

0 or greater	block written
-1	write error

FCLOSE	CLOSE	close the file described by the arguments placed on the machine stack as follows:
		2(SP) channel or unit to be closed

FLOAD is intended to "bootstrap" the tiny-c system. It reads a specific file, TCLOAD.TCF, from the system device. Since this file would typically have been prepared using the system editor, FLOAD provides a translation service between the output of this editor and the internal form required by the tiny-c interpreter. For most DEC systems this means simply eliminating carriage returns and nulls. FREAD and FWRITE, on the other hand, are meant to be complementary. That is, FREAD returns exactly what FWRITE has written.

7.2.2 User Option Routines

There are four opportunities for user routine activity during the execution of the tiny-c interpreter. JMP instructions to user routines to be called at each of these opportunities are the next four entries in the tiny-c/11 installation vector. The installation vector and corresponding opportunity description are as follows:

USERMC If an MC call with function number greater than 1000 is encountered in the tiny-c program being executed, its arguments are placed on the machine stack and a JSR to USERMC is executed as described in Section 2.10.

PRBEGN Before the standard Machine Call eleven, MC 11, enters an application program, a JSR to PRBEGN is executed.

STBEGN Before the interpretation of each tiny-c statement, a JSR to STBEGN is executed.

PRDONE After a return from an application program entered through the standard Machine Call eleven, MC 11, but before the tiny-c interpreter continues to process the program, a JSR to PRDONE is executed. which entered the application.

7.2.3 Working Areas

The four tiny-c working areas described in Chapter V have the addresses of their first and last bytes recorded in the next eight entries in the installation vector, as follows:

BSTACK	beginning of tiny-c stack
ESTACK	end of tiny-c stack
BVAR	beginning of variable table
EVAR	end of variable table
BFUN	beginning of function table
EFUN	end of function table
BPR	beginning of program space
EPR	end of program space

A ninth installation parameter is found in the system generation file. It is called

MSTACK The beginning (high-order) address of the machine stack space to be used by tiny-c. This address is loaded into SP when tiny-c is started.

7.3 The Sizing Constants

There are six assembly constants which can be used to control the size of the tiny-c/11 working areas. Their names, default values, and descriptions are as follows:

Name	Default Value	Description
MAXSTACK	30	the maximum number of entries on the tiny-c stack.
MAXVAR	100	the maximum number of active variable and function names in a tiny-c program; includes both PPS and application program variables.
MAXFUN	30	the maximum number of functions which can be active (not defined) in a tiny-c program. Each recursion instance counts as another active function.
MAXPR	20000	the size in bytes of the tiny-c program space.
MAXDEPTH	50	the maximum subroutine call depth of the tiny-c interpreter. Twelve times this value is the size of the machine stack allocated to tiny-c.
VLEN	8	the size in bytes of canonicalized variable and function names.

7.4 Subroutine Call and Processor Stack Regimen

tiny-c/11 uses the same subroutine call and processor stack regimen as that produced by the C compiler. This regimen is implemented by two subroutines, SAVE and RETURN. SAVE is called upon entry to any subroutine as follows:

```
JSR     R5,SAVE
```

and RETURN is called to leave a subroutine as follows:

```
JMP     RETURN
```

This method of subroutine entry and exit has the following advantages:

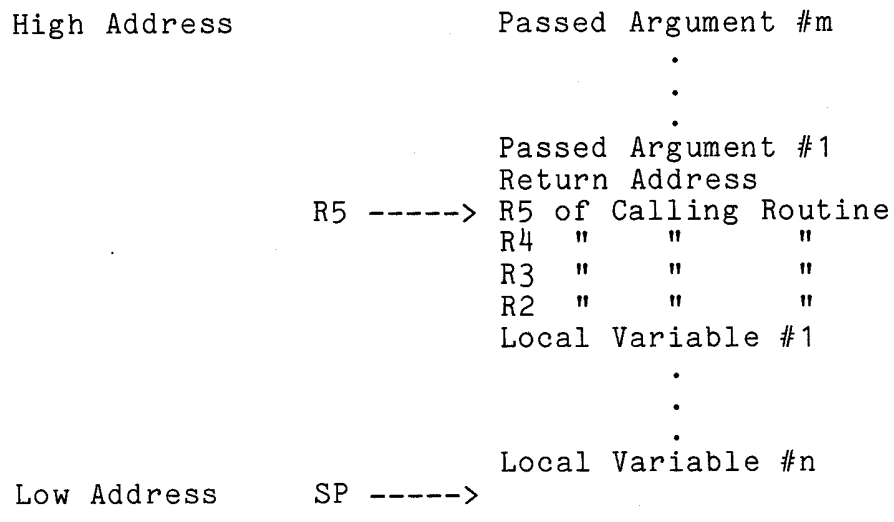
1. safety of registers R2, R3, R4, and SP over all subroutines which follow the regimen;
2. allocation, preservation, and deallocation of local variables on the processor stack;
3. recursive subroutine calling;
4. consistent handling of subroutine arguments; and,
5. a stack which contains much helpful debugging information.

It also has the following disadvantages:

1. reservation of register R5 as an additional stack pointer;
2. an unnecessarily large stack area; and
3. potentially wasted save, access, and restore cycles.

7.4.1 Local Stack Structure

Locally -- that is, within a subroutine -- the structure of the stack maintained by SAVE and RETURN looks like this:



Thus, arguments passed into the called subroutine are addressed as 4(R5), 6(R5), ... and local variables are addressed as -10(R5), -12(R5), ... Upon return from SAVE, the stack pointer points to the address under the saved R2 value. The called subroutine makes room for its own local variables by simply subtracting their number from SP.

7.4.2 Subroutine Calling

A subroutine is called by simply placing its arguments on the stack and executing a JSR to it. Thus, for example, to push a 0 on the tiny-c stack, the following code is executed:

```
CLR     (SP)
MOV     #2,-(SP)
MOV     #101,-(SP)
CLR     -(SP)
JSR     PC,@#PUSH
```

In order to conserve on stack space, SP should be reset upon return from the called subroutine. To continue our example,

```
        ADD     #6,SP
```

after the JSR instruction to "deallocate" the argument space.

The first thing the subroutine does is execute a

```
        JSR     R5,SAVE
```

which places R5 on the stack, moves PC (the address of the instruction after the JSR) into R5, and the entry point to SAVE in PC. Let's look at SAVE:

```
SAVE:   MOV     R5,R0           ;hold return address
        MOV     SP,R5           ;R5 points to stack
        MOV     R4,-(SP)        ;save registers
        MOV     R3,-(SP)
        MOV     R2,-(SP)
        JSR     PC,(R0)         ;go back to (R0)
```

The JSR to SAVE pushed the calling routine's R5 reference pointer and the new R5 points to it in the stack.

After the JSR to SAVE, the subroutine can freely use R2, R3, R5, and SP. To return control to the routine that called this subroutine, execute a

```
        JMP     RETURN
```

to the following code:

```
RETURN: MOV      R5,R2          ;temporary copy of R5
        MOV     -(R2),R4        ;restore registers
        MOV     -(R2),R3
        MOV     -(R2),R2
        MOV      R5,SP          ;point to old R5
        MOV      (SP)+,R5       ;restore R5 and point to
                                ; return
        RTS      PC             ;go to return and bump SP
```

The calling routine gets registers R2, R3, R4, R5, and SP back in exactly the same shape it left them. The called routine can pass values back to the calling routine either in R0 and R1 or in the arguments. Typically, a called subroutine will not alter the values of a calling subroutine's local variables unless it is given a pointer argument. Thus, the relationship between local variables and arguments which is enforced in the tiny-c language is echoed in the tiny-c interpreter subroutines.

7.4.3 Interpreter Subroutines Following the Regimen

One could, of course, require that all subroutines observe the SAVE/RETURN regimen. Subroutines, such as ST, which can indirectly call themselves, clearly MUST follow the regimen. Others, such as BLANKS, which don't call any other routines, simply waste cycles doing needless saves and restores. The following tabulation separates those subroutines which need the regimen from those which don't:

FIGURE 7-1

SUBROUTINES WHICH

DO USE SAVE and RETURN		DO NOT	
ADDRVAL	LINK	ALPHA	NEWFUN
ASGN	NEWVAR	ALPHAN	NUM
CANON	RELN	ATOI	POP
CONST	SETARG	BLANKS	PUSH
ENTER	SKIPST	CEQ	REM
EQ	ST	CEQN	SKIP
EXPR	TERM	ESET	SYMNAM
FACTOR	VALLOC	FUNDON	TCTOI
		LIT	TOPTOI
		MOVN	

End of FIGURE 7-1

Further improvements are certainly possible. While tiny-c/11 has made some concessions to speed and efficiency, program consistency and clarity have not been sacrificed to obtain them. Owners will find that it is easier to optimize an unoptimized program than it is to reoptimize a misoptimized one.

BIBLIOGRAPHY

Feurzeig, W., et al., "Programming Languages as a Conceptual Framework for Teaching Mathematics," BBN Report No. 2165 (June 1971).

Duncan, Ray, "tiny-c Interpreter on CDOS," DR. DOBB'S JOURNAL, Vol. 4, Issue 5 (May 1979).

Gibson, Paul, "Stopwatch: A tiny-c Program," CREATIVE COMPUTING, Vol 5, No. 3 (March 1979).

Gibson, Tom, "Caution: Structured Programming Can Be Habit-Forming," CREATIVE COMPUTING, Vol. 5, No. 1 (January 1979).

Gries, David, "On Structured Programming - A Reply to Smoliar," CACM, Vol. 17, No. 11 (November 1974).

Hancock, L., "Growing, Pruning & Climbing Binary Trees with tiny-c," DR. DOBB'S JOURNAL, Vol. 4, Issue 6 (June/ July 1979).

Kemeny, J. G. and Kurtz, T. E., BASIC PROGRAMMING, Wiley, 1967.

Kernighan, Brian W. and Plauger, P. J., SOFTWARE TOOLS, Addison-Wesley, 1967.

Kernighan, Brian W. and Ritchie, Dennis M.,THE C PROGRAMMING LANGUAGE, Prentice-Hall Software Series, 1978.

Madden, J. Gregory, "C: A Language for Microprocessors?," BYTE, Vol. 2, No. 10 (October 1977).

Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System," CACM, Vol. 17, No. 3 (March 1974).

Ritchie, D. M., et al., "The C Programming Language," Computer Science Technical Report No. 31, Bell Telephone Laboratories, Murray Hill, N. J. 07974 (October 1975).

Snyder, Alan, "A Portable Compiler for the Language C," MIT Project MAC Technical Report 149, May 1975.

Zahn, C. T., C NOTES: A Guide to the C Programming Language, Yourdon Press, 1979.

Appendix C -- Crunched PPS Code

```
/* <CR>=1Ø, PDP-11 file names, console line=132
[main();]
putchar char c
[if(c==Ø)c='""'
return MC c,1
]
getchar
[return MC 2
]
gs char b(Ø) [
return MC(b,15)
]
pft char f(Ø),t(Ø)[
  MC(f,t,13)
]
ps char b(Ø)
[int 1
char c
1=-1
while((c=b(1=1+1))!=Ø)MC c,1
return 1
]
pl char b(Ø)
[MC 1Ø,1
ps b
]
alpha char a
[
if((a>='a')*(a<='z'))return 1
if((a>='A')*(a<='Z'))return 1
]
num char b(5)
int v(Ø)
[int k
v(Ø)=Ø
while(k<5)
[if((b(k)<'Ø')+(b(k)>'9'))return k
v(Ø)=1Ø*v(Ø)+b(k)-'Ø'
k=k+1
]
return k
]
```

```
atoi char b(Ø)
int v(Ø)
[int k,s
char c
s=1
c=b(Ø)
while((c==' ')+(c=='-')+(c=='+'))
[if(c=='-')s=-1
c=b(k=k+1)
]
k=k+num(b+k,v)
v(Ø)=s*v(Ø)
return k
]
pn int n
[
MC ' ',1
MC n,14
]
gn
[char b(2Ø)
int v(Ø)
while(1)
[gs b
if(atoi b,v)return v(Ø)
ps"number required "
]
]
ceqn char a(Ø),b(Ø)
int n
[int k
k=-1
while((k=k+1)<n)if(a(k)!=b(k))return Ø
return 1
]
index char i(Ø)
int l
char f(Ø)
int n
[
if(n<=Ø)return l
if(l<=Ø)return Ø
int a,d(Ø)
while(a+n<=l)[
d(Ø)=1
a=a+1+scann(i+a,i+1-n,f(Ø),d)
if(d(Ø))return Ø
if(ceqn(i+a,f+1,n-1))return a
]
]
```

```
move char a(Ø),b(Ø)
[int k
int l(Ø)
l(Ø)=l
k=scann(a,a+32765,Ø,l)
movebl(a,a+k,b-a)
return k
]
gc
[char f
f=MC 2
while(MC(2)!=lØ)[]
return f
]
movebl char a(Ø),b(Ø);int n
[MC(a,b,n,7)]
countch char a(Ø),b(Ø),c
[return MC(a,b,c,8)]
scann char a(Ø),b(Ø),c;int n(Ø)
[return MC(a,b,c,n,9)]
filename char fin(Ø),fout(Ø)[
int j(Ø),k,m
if(fin(3)!=':') moven("dk :",fout,4)
else [moven(fin,fout,4);fin=fin+4;]
if((alpha(fin(Ø))+('Ø'<=fin(Ø))*(fin(Ø)<='9'))==Ø) return -l
j(Ø)=l;m=scann(fin,fin+99,Ø,j)
if(j(Ø)!=Ø)[pl"name too long";return;]
j(Ø)=l;k=scann(fin,fin+m,'.',j)
moven("          ",fout+4,lØ)
if(j(Ø)!=Ø)[if(m<=6)moven(fin,fout+4,m)
            else    moven(fin,fout+4,6)
            moven(".tcf",fout+lØ,4);]
else[if(k<=6)moven(fin,fout+4,k)
     else    moven(fin,fout+4,6)
     if(m-k<=4)moven(fin+k,fout+lØ,m-k)
     else      moven(fin+k,fout+lØ,4);]
]
moven char f(Ø),t(Ø)
       int n [
  if(n)movebl(f,f+n-l,t-f)
]
readfile char n(Ø),w(Ø),l(Ø)
int u
[int k,t
char fi(13)
if(filename(n,fi)<Ø) [
   pl"Invalid file name";pl""
   return -l
]
```

```
if(MC(1,fi,0,u,3)<0) [
   pl"File not found"
   MC(u,6)
   return -1
   ]
while(w+511<=1)
[
k=MC(w,u,4)
if(k== -1) [MC(u,6);return t;]
if(k<  -1) [MC(u,6);return k;]
w=w+k
t=t+k
]
pl"Too big"
MC(u,6)
return -2
]
writefile char n(0),b(0),e(0)
int u
[int k,t,1
char fi(13)
if(filename(n,fi)<0) [
   pl"Invalid file name";pl""
   return -1
]
if(MC(2,fi,e-b+1,u,3)<0) [
   pl"Unit 1 open"
   MC(u,6)
   return -1
   ]
while(b<=e)
[
l=e-b
if(l>511)l=511
k=MC(b,b+1,u,5)
if(k<0) [MC(u,6);return k;]
t=t+1+1
b=b+1+1
]
MC(u,6)
return t
]
fopen int rw
      char n(0)
      int s,u [
return MC(rw,n,s,u,3)]
fread char a(0)
      int u [
return MC(a,u,4)]
fwrite char f(0),t(0)
      int u [
return MC(f,t,u,5)]
fclose int u [
```

```
int er(Ø),cu,lo,pe,lp
int ll,la
char ft(4Ø),tt(8Ø)
int fl,tl
char ln(133),pr(12ØØØ)
main
[char c
int v(1)
lp=12ØØØ
pr(Ø)=1Ø
while(1)
[ps">"
while((ll=gs(ln))==Ø)[]
c=ln(Ø)
if(c=='.')
[if(num(ln+1,v))go(v)
else if((ln(2)==Ø)+(alpha(ln(2))==Ø))
[c=ln(1)
if(c=='p')pt
else if(c=='d')dl
else if(c=='l')oi
else if(c=='c')ch
else if(c=='/')fa
else if(c=='r')gi
else if(c=='w')gu
else if(c=='x')return
else [ps"???";pl""]
]else st
]
else if(c=='-')up
else if(c=='+')do
else in
]
]
pi int n
[int f,l,v(Ø)
v(Ø)=n
f=fc
lo=lo+v(Ø)-1
l=cu+scann(pr+cu,pr+pe,1Ø,v)
cu=l
lo=lo-v(Ø)
MC pr+f,pr+l,13
]
fc
[int k
if((k=cu)==Ø)return Ø
while(pr(k=k-1)!=1Ø)if(k<=Ø)break
return k+1
]
```

```
lc
[int k
k=cu-1
while(pr(k=k+1)!=1Ø)if(k>=pe)break
return k
]
nl
[if((cu=lc()+1)>pe)
[cu=pe
return Ø
]
return lo=lo+1
]
bl
[if((cu=fc()-1)<Ø)cu=Ø
else lo=lo-1
]
pt[
int v(Ø)
if(ln(2))num(ln+3,v)
else v(Ø)=1
pi(v(Ø))
]
dl
[int f,l,v(1)
if(cu==Ø)
[ps"cannot delete line Ø";pl""
return
]
if(ln(2)==Ø)v(Ø)=1
else num(ln+3,v)
la=la-v(Ø)
f=fc
l=cu+scann(pr+cu,pr+pe,1Ø,v)
la=la+v(Ø)
lo=lo-1
cu=f-1
if(l<pe)movebl(pr+l+1,pr+pe,-(l-f+1))
pe=pe-(l-f+1)
]
```

```
oi
[
int k
if(ln(3)==Ø)return
if(ln(2)!=Ø)
[fl=move(ln+3,ft)
if(ft(Ø)=='^')ft(Ø)=1Ø
if(ft(fl-1)=='^')ft(fl-1)=1Ø
]
if(fl==Ø)
[pl"locate what?";pl""
return
]
if(nl()!=Ø)[
if(k=index(pr+cu-1,pe-cu+2,ft,fl)) [
cu=cu-2+k
if(pr(cu)==1Ø)cu=cu+1
lo=countch(pr,pr+cu-1,1Ø)
pi 1
]
else[ps"?";pl""]
]
else[ps"at bottom";pl""]
]
ch[
char d
int p,f,l
if(ln(2)!=Ø)[
d=ln(2)
p=2
while((ln(p=p+1)!=d)[
if(ln(p)==Ø)[
ln(p+1)=Ø
break
]
]
ln(p)=Ø
fl=move(ln+3,ft)
tl=move(ln+p+1,tt)
if(tl)if(tt(tl-1)==d)tl=tl-1
]
f=fc
l=lc()-1
int k
if(k=index(pr+f,l-f+1,ft,fl)) [
cu=f+k-1
movebl(pr+cu+fl,pr+pe,tl-fl)
pe=pe+tl-fl
if(tl)movebl(tt,tt+tl-1,pr+cu-tt)
]
pi 1
]
```

```
in
[
ll=ll+l
if(pe+ll>lp)
[ps"won't fit";pl""
return
]
if(nl)movebl(pr+cu,pr+pe,ll)
else[cu=cu+l;lo=lo+l]
pe=pe+ll
movebl(ln,ln+ll-l,pr-ln+cu)
pr(cu+ll-l)=lØ
la=la+l
]
wh
[int f,l,u,b
pn lo;ps" --- err ";pn er(Ø);pl""
u=cu
f=fc
b=u-f
l=lc
f=f-l
while((f=f+l)<l)putchar(pr(f));pl""
while((b=b-l)>=Ø)putchar(' ')
putchar '<';pl""
]
do
[int v(l)
if(ln(l)==Ø)v(Ø)=l
else num(ln+l,v)
lo=lo+v(Ø)
v(Ø)=v(Ø)+l
cu=cu+scann(pr+cu,pr+pe,lØ,v)
lo=lo-v(Ø)
pi(l)
]
up
[int v(l)
if(ln(l)==Ø)v(Ø)=l
else num(ln+l,v)
if((v(Ø)=lo-v(Ø))<Ø) v(Ø)=Ø
go(v)
]
```

```
go int l(l)
[lo=l(Ø)
l(Ø)=l(Ø)+l
cu=scann(pr,pr+pe,lØ,l)
lo=lo-l(Ø)
pi(l)
]
fa
[pn lo;pn la;pn pe;pn lp-pe;pl""]
st
[ln(ll)=' '
while((ll=ll+l)<=133)ln(ll)=' '
MC(er,ln+l,pr+pe,pr,ll)
if(cu<Ø)cu=Ø;if(cu>pe)cu=pe
lo=countch(pr,pr+cu-l,lØ)
pl"";pl""
if(er(Ø))
if(er(Ø)==99)[ps"stopped";pl""]
else wh
]
gi
[int k
if(ln(2)==Ø) ln(3)=Ø
pn k=readfile(ln+3,pr+pe+l,pr+lp,l)
pl ""
if(k<Ø)return
pe=pe+k
la=countch(pr+l,pr+pe,lØ)
fa
]
gu
[fa
if(ln(2)==Ø) ln(3)=Ø
pn writefile(ln+3,pr+l,pr+pe,l)
pl""
]
```

INDEX

tiny-c "Load and Go" Version

for the

Heath* HDOS* Disk Operating System

NOTE: This is not a tiny-c Manual. This is a supplement to the tiny-c OWNER'S MANUAL, which is available separately.

TABLE OF CONTENTS

Installed by:
J.J. Thompson
281 Warren Ave.
Kenmore, N.Y. 14217
Phone-716-873-0380

HD-1 HARDWARE:

The hardware requirements to implement this version of tiny-c are a Heath computer with disk system containing at least 16K of memory.It is preferable that you have 24K of memory.

HD-2 PREPARATION:

It is recommended that before you try to run tiny-c you thoroughly read the tiny-c OWNER'S MANUAL. It is assumed that you are familiar with the HDOS operating system.

The tiny-c distribution disk is a non-bootable disk.We strongly recommend that you make a copy as outlined below, and put the distribution disk away in a safe place.

HD-3 IMPLEMENTATION:

The distribution disk you recieved contains the four files TINYC.ABS; PPS.TNC; TINYC.ASM; HDOS.ACM; AND TINYC.DOC. The first is the binary file object code for tiny-c. The second is the Program Preparation System and is an ASCII file. The next two are the assembler source code and the assembler common deck. These can be submitted to the Heath disk assembler (ASM) to generate the binary file. The last is a copy of this installation guide.

The distribution disk does not contain HDOS and is, therefore a non-bootable disk. To implement tiny-c carry out the following procedure:

1) Initialize a fresh disk using the INIT17 program as described in the HDOS manual.
2) SYSGEN this disk as described in the HDOS manual.
3) Delete all files from this SYSGENED disk except the file ERRORMSG.SYS. In most cases you will have to change flags on the files to allow them to be erased. If you do not want tiny-c to print error messages you can also delete the file ERRORMSG.SYS.
   If you want to use a line printer or alternate terminal include the proper device driver in your disk.
4) You should now have a SYSGENED disk containing only the system files HDOS.SYS; HDOSOVL.SYS; PIP.SYS; SYSCMD.SYS; and ERRORMSG.SYS; and possibly a device driver.
5) Copy to this new disk the files TINYC.ABS, and PPS.TNC from the tiny-c distribution disk.

You now have a bootable disk containing tiny-c and the program preparation system. To load tiny-c simply boot the disk using the normal HDOS procedure. When HDOS prompts for a command with ">" type "TINYC"(cr) and tiny-c will be loaded. You will see the copyright message and then the triple prompt ">>>". Now type ".R PPS"(cr) and the program preparation system will be loaded. When the triple prompt ">>>" shows again type ".G"(cr) and PPS will start and give its prompt "#".

HD-4 MEMORY ALLOCATION:

The Heath system uses split octal notation for memory

references. therefore all memory references will be in split octal with hex values given in parenthesis.

In the heath system useable memory starts at 040-000(the system monitor uses the area from 040-000(2000H) to 040-100(2040H). HDOS uses the area from 040-100(2040H) to 041-146(2166H) for system cells and system work area. The 8080 stack is usually assigned to the 282 byte region from 041-146(2166H) to 042-200(2280H), and application programs start at 042-200(2280H). Because tiny-c may require more machine stack space than the 282 bytes normally allocated the origin of this version of tiny-c is at 045-000(2500H) with the area below this down to 041-146(2166H) allocated to the 8080 machine stack (922 bytes).

The interpreter itself resides in the 5207 byte area from 045-000(2500H) to 071-127(3957H).the tiny-c stack is located in the 128 byte area from 071-127(3957H) to 071-327(39D7H), and the function stack is located in the 128 byte area from 071-347(39E7H) to 072-147(3A67H). The disk operating system and its overlay reside in the upper 4760 bytes of memory. The initialization routine of this version of tiny-c automatically allocates 20% of the remaining memory to the variable stack, all the rest being allocated to program storage.

HD-5 MACHINE CALLS (MC'S):

At this time no special machine calls have been implemented,however the 128 byte area from 070-324(38D4H) to 071-124(3954H) has been left for their use. Currently there is simply a jump to the MC error routine at location 0-324(38D4H). The first 20 user machine calls (1001-1020) are reserved for future implementation.

HD-6 SPECIAL CHARACTERS:

This implementation supports upper case only. Any lower case input is masked to upper case before being used.

The character delete and line delete characters are control-H and control-U respectively, as used by HDOS. The HDOS console routine is configured to accept character mode input with no echo.

The program halt character is control-C. If this key is depressed during an application program return will be made to the systems program (normally PPS). If control-C is depressed when in a systems program return is to the warm start location of tiny-c from which entry can be made back into the systems program by entering ".G"(cr).

The HDOS system uses the ASCII newline character (0AH) as a line terminator. This version of tiny-c has been modified to allow the new line character to be the line terminator instead of the ASCII carriage return, which is normally used by tiny-c. This makes HDOS files compatible with tiny-c, and allows the use of the HDOS text editor "EDIT"" to edit tiny-c files.

HD-7 FILES:

Files are accessed in tiny-c just as they are in HDOS. All that is needed is a file descriptor (DEV:FNAME.EXT) where DEV is the device designation, FNAME is an 8 character or less file name and EXT is a three character extension. The default

extension for all tiny-c files is TNC. As with HDOS a line printer or auxiliary terminal can be specified, and output will be to that device. Thus after entering a program in PPS typing .W LP: will cause the program to be listed on the line printer. Remember to achieve this capability the correct device driver must be resident on the disk. This implementation supports only one channel of file access at a time, this is channel 1. This channel is automatically specified to the disk operating system by the FOPEN, FCLOSE, FREAD, and FWRITE routines.

File error messages are presented as they are in HDOS programs. That is if the file ERRORMSG.SYS is present a descriptive error message will be given. If that file is not present only an error number will be given.

HD-8 PROGRAM PREPARATION SYSTEM (PPS):

The program preparation system (file PPS.TNC) is pretty much identical to the crunched version given in appendix C of the OWNERS MANUAL. The line and character delete characters used by the function getchar have been changed to reflect those used by this implementation. All functions which look for the end of a line have been changed to look for the HDOS newline character instead of a carriage return. The function READFILE has been slightly modified to check for an error on file opening (usually caused by trying to open a non-existant file). If an error is reported on file opening the function returns without trying to read the file. This modification prevents HDOS from giving a double error report if an attempt is made to read a non-existant file. All corrections mentioned in newsletters 1 and 2 have been incorporated. The line length is set at 64, the program buffer is set at 7000, and the length of the program is set at 7000. The PPS prompter for this implementation is "#" to avoid confusion with the HDOS prompter. The PPS can easily be edited or changed by using the HDOS text editor EDIT.

HD-9 SOURCE CODE AND ASSEMBLY:

The installation source code is given in the appendix at the end. The complete tiny-c source listing is given as file TINYC.ASM on the distribution disk. This file along with the file HDOS.ACM can be submitted to the Heath disk assembler ASM to give a complete source listing. Alternatively a complete assembly listing can be obtained for $10.00 from the installer. Orders should be placed to the address given in the table of contents.

The installation code makes use of system calls (SCALLS) to the HDOS operating system. Further details on syscalls are given in the HDOS SYSTEM PROGRAMMER'S REFERENCE MANUAL which is available from the Heath Users Group. Also several calls are made to routines in the disk system controller ROM. These routines are listed in the HDOS system manual.

HD-10 IN CASE OF PROBLEMS:

If you encounter any installation related problems, write to the installer at the address given in the table of contents. Make sure you give a full description of the problem. Every effort will be made to try to assist you with the problem.

TABLE OF IMPORTANT LOCATIONS

LOCATION	ADDRESS
ORIGIN	045-000(2500H)
BSTACK	045-066(2536H)
ESTACK	045-070(2538H)
BFUN	045-072(253AH)
EFUN	045-074(253CH)
BVAR	045-076(253EH)
EVAR	045-100(2540H)
BPR	045-102(2542H)
EPR	045-104(2544H)
USERMC	070-324(38D4H)
COLD	061-130(3158H)
WARM	061-176(317EH)
HOT	061-201(3181H)
START OF INSTALLATION CODE	066-075(363DH)
START OF HDOS INITIALIZATION	070-134(385CH)

SPECIAL CHARACTERS

CHARACTER DELETE	CONTROL-H
LINE DELETE	CONTROL-U
ESCAPE CHARACTER	CONTROL-C
PPS PROMPTER	#

INSTALLATION SOURCE CODE

```
066.075 345          CHRDY.    PUSH   H       SAVE HL
066.076 041 135 066    LXI     H,TEMPC
066.101 176            MOV     A,M     SEE IF CTLC SAVED BY CTLC ROUTINE
066.102 376 003        CPI     003
066.104 312 114 066    JZ      CHRDY2  YES PROCESS IT
066.107 377 001        SCALL   .SCIN   NO CHECK FOR OTHER CHARACTER INPUT
066.111 332 130 066    JC      NOCHR   NO INPUT
066.114 167          CHRDY2    MOV     M,A     SAVE CHAR IN TEMPC FOR INCH
066.115 376 000        CPI     0
066.117 312 125 066    JZ      CHRDY3  HAVE NULL
066.122 303 157 066    JMP     MLCU    NO MASK TO UPPER CASE IF NECESSARY
066.125 074          CHRDY3    INR     A       A=1 FOR NULL
066.126 341            POP     H
066.127 311            RET
066.130 066 000      NOCHR     MVI     M,0     CLEAR TEMPC
066.132 257            XRA     A
066.133 341            POP     H
066.134 311            RET
066.135 000          TEMPC     DB      0
066.136 345          INCH.     PUSH    H
066.137 041 135 066 INCH1      LXI     H,TEMPC
066.142 176            MOV     A,M     GET CHAR FROM TEMPC
066.143 066 000        MVI     M,0     MAKE SURE TEMPC IS CLEARED
066.145 376 000        CPI     0       WAS CHAR LEFT IN TEMPC?
066.147 302 157 066    JNZ     MLCU    YES USEIT
066.152 377 001        SCALL   .SCIN   NO GET CHAR FROM TERMINAL
066.154 332 137 066    JC      INCH1   NO CHAR YET TRY AGAIN
              *                ROUTINE TO MAKE A LOWER CASE CHAR UPPER CASE
066.157 376 141      MLCU      CPI     141Q
066.161 332 173 066    JC      MLCU1
066.164 376 173        CPI     173Q
066.166 322 173 066    JNC     MLCU1
066.171 326 040        SUI     040Q
066.173 341          MLCU1     POP     H
066.174 311            RET
066.175 365          OUTCH.    PUSH    PSW
066.176 377 002        SCALL   .SCOUT
066.200 361            POP     PSW
066.201 311            RET
066.202 315 054 031 ERROR      CALL    $SAVALL SAVE ALL REGISTERS
066.205 365            PUSH    PSW     SAVE ERROR NUMBER
066.206 041 223 066    LXI     H,ERRORM        POINT TO MESSAGE
066.211 377 003        SCALL   .PRINT  PRINT IT
066.213 361            POP     PSW     GET ERROR NUMBER BACK
066.214 046 012        MVI     H,0AH
066.216 377 057        SCALL   .ERROR
066.220 303 047 031    JMP     $RSTALL GET REGISTERS AND RETURN
```

```
066.223 012 105 122 ERRORM    DB      0AH,'ERROR-',' '+2000Q
066.233 000         EOFFLG    DB      0       FLAG FOR END OF FILE ON READ
066.234             FBUFF     DS      256     BUFFER FOR FILE READS AND WRITES
067.234 123 131 060 DEFALT    DB      'SY0TNC',0      DEFAULT VALUES FOR DEVICE AND EXT
067.243 345         FOPEN.    PUSH    H
067.244 041 233 066   LXI     H,EOFFLG
067.247 066 000       MVI     M,0     MAKE SURE EOFFLG IS CLEARED
067.251 341           POP     H
067.252 021 234 067   LXI     D,DEFALT
067.255 376 002       CPI     2       IS IT OPEN FOR WRITE
067.257 076 001       MVI     A,1     CHANNEL 1
067.261 312 273 067   JZ      OPENW   YES IS WRITE OPEN
067.264 377 042       SCALL   .OPENR  NO OPEN FOR READ
067.266 332 302 067   JC      OERR    GO IF OPENING ERROR
067.271 257           XRA     A       OPENED OK
067.272 311           RET
067.273 377 043     OPENW     SCALL   .OPENW  OPEN FILE FOR WRITE
067.275 332 302 067   JC      OERR
067.300 257           XRA     A       NO ERROR
067.301 311           RET
067.302 315 202 066 OERR      CALL    ERROR
067.305 076 001       MVI     A,1     FLAG THAT ERROR OCCURRED
067.307 267           ORA     A       SET FLAGS
067.310 311           RET
067.311 076 001     FCLOSE.   MVI     A,1     CHANNEL 1
067.313 377 046       SCALL   .CLOSE
067.315 322 325 067   JNC     CLOSE2  OPERATION OK NO ERROR
067.320 376 011       CPI     EC.FNO  DID WE TRY TO CLOSE AN UNOPEN CHANNEL?
067.322 302 202 066   JNZ     ERROR   IF NOT IT IS A SERIOUS ERROR REPORT IT.
067.325 311         CLOSE2    RET
067.326 345         FREAD.    PUSH    H
067.327 041 233 066   LXI     H,EOFFLG        CHECK EOFFLG
067.332 176           MOV     A,M
067.333 066 000       MVI     M,0     MAKE SURE IT'S CLEARED
067.335 037           RAR
067.336 322 351 067   JNC     FREAD2  OK TO READ NO EOF YET
067.341 021 000 000   LXI     D,0     NO BYTES READ THIS TRIP
067.344 341           POP     H
067.345 076 377       MVI     A,-1    SHOW THAT EOF WAS SEEN
067.347 267           ORA     A       SET FLAGS
067.350 311           RET
067.351 041 234 066 FREAD2    LXI     H,FBUFF
067.354 006 000       MVI     B,0
067.356 315 212 031   CALL    $ZERO   ZERO THE BUFFER
067.361 076 001       MVI     A,1     CHANNEL 1
067.363 001 000 001   LXI     B,256   256 BYTES
067.366 021 234 066   LXI     D,FBUFF TO THE BUFFER
067.371 377 004       SCALL   .READ   GET IT FROM THE DISK
067.373 332 034 070   JC      RERR    PROCESS ERROR
067.376 341         FREAD3    POP     H       GET DESTINATION
067.377 345           PUSH    H       AND REFRESH IT
070.000 021 000 000   LXI     D,0     DE COUNTS THE BYTES
070.003 001 234 066   LXI     B,FBUFF   FBUFF HAS THE DATA
070.006 172         RLOOP     MOV     A,D
070.007 376 001       CPI     1       HAVE WE READ 256 BYTES YET
070.011 312 031 070   JZ      RDON    IF SO THIS TRIP IS DONE
070.014 012           LDAX    B       NO GET NEXT BYTE
```

```
070.015  376 000        CPI     0       IS IT AN EOF?
070.017  312 031 070    JZ      RDON    IF SO THIS TRIP IS DONE
070.022  167            MOV     M,A     NOT DONE YET SAVE CHAR
070.023  043            INX     H
070.024  003            INX     B
070.025  023            INX     D
070.026  303 006 070    JMP     RLOOP   GO BACK AND GET ANOTHER
070.031  341          RDON      POP     H
070.032  257            XRA     A       READ SUCCESSFUL
070.033  311            RET
070.034  376 001      RERR      CPI     EC.EOF  IS IT JUST EOF
070.036  302 051 070    JNZ     RERR2   NO ITS MORE SERIOUS
070.041  076 001        MVI     A,1     ONLY EOF FLAG IT
070.043  062 233 066    STA     EOFFLG
070.046  303 376 067    JMP     FREAD3
070.051  021 000 000 RERR2      LXI     D,0     NO BYTES READ
070.054  341            POP     H GET H BACK AND CLEAR STACK
070.055  303 302 067    JMP     QERR    REPORT ERROR
070.060  345          FWRITE.   PUSH    H
070.061  325            PUSH    D
070.062  041 234 066    LXI     H,FBUFF
070.065  006 000        MVI     B,0
070.067  315 212 031    CALL    $ZERO   ZERO THE BUFFER
070.072  321            POP     D       GET LAST BYTE BACK
070.073  341            POP     H       GET FIRST BYTE BACK
070.074  001 234 066    LXI     B,FBUFF   BC POINTS TO FILE BUFFER
070.077  053            DCX     H
070.100  043          WLOOP     INX     H       POINT TO NEXT BYTE
070.101  176            MOV     A,M     GET IT
070.102  002            STAX    B       PUT CHAR INTO FILE BUFFER
070.103  003            INX     B       POINT TO NEXT BUFFER LOCATION
070.104  315 216 030    CALL    $CDEHL  WAS IT THE LAST BYTE TO SAVE?
070.107  302 100 070    JNZ     WLOOP   NOT DONE GET NEXT
070.112  076 001        MVI     A,1     CHANNEL 1
070.114  001 000 001    LXI     B,256   256 CHARACTERS
070.117  021 234 066    LXI     D,FBUFF
070.122  377 065        SCALL   .WRITE  WRITE THE BUFFER
070.124  332 302 067    JC      QERR
070.127  257            XRA     A       WRITE OK
070.130  311            RET
070.131  257          EXIT      XRA     A
070.132  377 000        SCALL   .EXIT   GO BACK TO MDOS
070.134  041 377 377 INIT       LXI     H,-1
070.137  377 052        SCALL   .SETTOP SEE HOW MUCH MEMORY IS AVAILABLE
070.141  353            XCHG
070.142  052 324 040    LHLD    S.OMAX  GET OVERLAY SIZE
070.145  001 012 000    LXI     B,10    10 BYTES FOR SLOPOVER
070.150  011            DAD     B       HL=OVERLAY SIZE PLUS SLOP
070.151  315 224 030    CALL    $CHL    HL=-HL
070.154  031            DAD     D       HL= AMOUNT OF MEMORY AVAILABLE WITH OVERLAY PRESENT
070.155  377 052        SCALL   .SETTOP TELL MDOS WHERE END OF MEMORY WILL BE
070.157  052 322 040    LHLD    S.USRM  GET END ADDRESS OF USEABLE MEMORY
070.162  353            XCHG
070.163  041 167 072    LXI     H,VAR   GET END ADDRESS OF TINY C
070.166  315 224 030    CALL    $CHL    HL=-HL
070.171  031            DAD     D       HL= BYTES AVAILABLE FOR VARIABLES AND PROGRAM
070.172  353            XCHG            PUT THIS IN DE
```

```
070.173  001 005 000     LXI     B,5
070.176  315 157 046     CALL    DDIV    FIND OUT WHAT 20% OF THIS IS
070.201  041 167 072     LXI     H,VAR
070.204  031             DAD     D       HL=END ADDRESS FOR VARIABLES
070.205  345             PUSH    H       SAVE IT
070.206  315 224 030     CALL    $CHL    MAKE NEGATIVE
070.211  042 100 045     SHLD    EVAR
070.214  341             POP     H       GET END ADDRESS BACK
070.215  021 020 000     LXI     D,16
070.220  031             DAD     D       ADD 16 BYTE GUARD BAND
070.221  042 102 045     SHLD    BPR
070.224  052 322 040     LHLD    S.USRM  GET PROGRAM END ADDRESS
070.227  315 224 030     CALL    $CHL    MAKE NEGATIVE
070.232  042 104 045     SHLD    EPR
070.235  257             XRA     A       SET CONSOLE MODE FLAGS
070.236  006 201         MVI     B,201Q  SET CHARACTER MODE WITH NO ECHO
070.240  016 201         MVI     C,201Q  EFFECT BOTH BITS
070.242  377 006         SCALL   .CONSL  SET UP CONSOLE FOR CHARACTER MODE WITH NO ECHO
070.244  076 003         MVI     A,003   CONTROL C FLAG
070.246  041 261 070     LXI     H,CTLC  ADDRESS OF CONTROL C ROUTINE
070.251  377 041         SCALL   .CTLC   SET UP ROUTINE TO PROCESS CONTROL C
070.253  332 202 066     JC      ERROR   IF      ERROR
070.256  303 130 061     JMP     COLD
                *  THIS ROUTINE PROCESSES CONTROL C IF APPLICATION LEVEL IS   0
                *  RETURN IS MADE TO LOADER AT WARM. IF APPLICATION LEVEL IS > 0
                *  RETURN IS MADE TO LEVEL     0.
070.261  365         CTLC    PUSH    PSW
070.262  345             PUSH    H
070.263  041 320 070     LXI     H,CCMSG
070.266  377 003         SCALL   .PRINT  SHOW THAT CONTROL C HIT
070.270  341             POP     H
070.271  072 142 045     LDA     APPLVL  FIND OUT WHAT LEVEL WE'RE AT
070.274  267             ORA     A
070.275  302 311 070     JNZ     CTLC2   NOT LEVEL 0
070.300  377 007         SCALL   .CLRCO  ARE LEVEL 0 CLEAR CONSOLE BUFFER AND FLAGS
070.302  361             POP     PSW
070.303  301             POP     B       CLEAN UP STACK
070.304  301             POP     B       "
070.305  301             POP     B       "
070.306  303 176 061     JMP     WARM
070.311  076 003     CTLC2   MVI     A,003
070.313  062 135 066     STA     TEMPC   SHOW THAT CONTROL C HIT
070.316  361             POP     PSW
070.317  311             RET
070.320  012 136 103 CCMSG   DB      0AH,'^C',0AH+200Q
070.324  303 125 064 USERMC.  JMP     MDESET
070.327                 DS      128
071.127             PSTACK  DS      144     128 BYTES + 16 BYTE GUARD
071.347             FUN     DS      144     128 BYTES + 16 BYTE GUARD
072.167  000          VAR     DB      0       START OF VARIABLE TABLE
072.170  000             END     ENTRY
```

tiny-c Owner's Manual
2nd Printing

ERRATA

Appendix A

1. On page A-45, thirteen lines after EN11, delete the RET and insert the following lines in its place:

```
        LHLD    TOP     ;assume returned
        INX     H       ;  result is an
        MOV     A,M     ;  actual
        CPI     'A'
        RZ
        CALL    TOPTOI
        JMP     PUSHK
```

The new version number is 80-01-03.

Appendix B

1. On page B-3, change the third line from

```
typedef char TEXT;
```

to

```
typedef char TINY, TEXT;
```

2. On page B-19, replace line 29,

```
else leave = NO;
```

with the following lines:

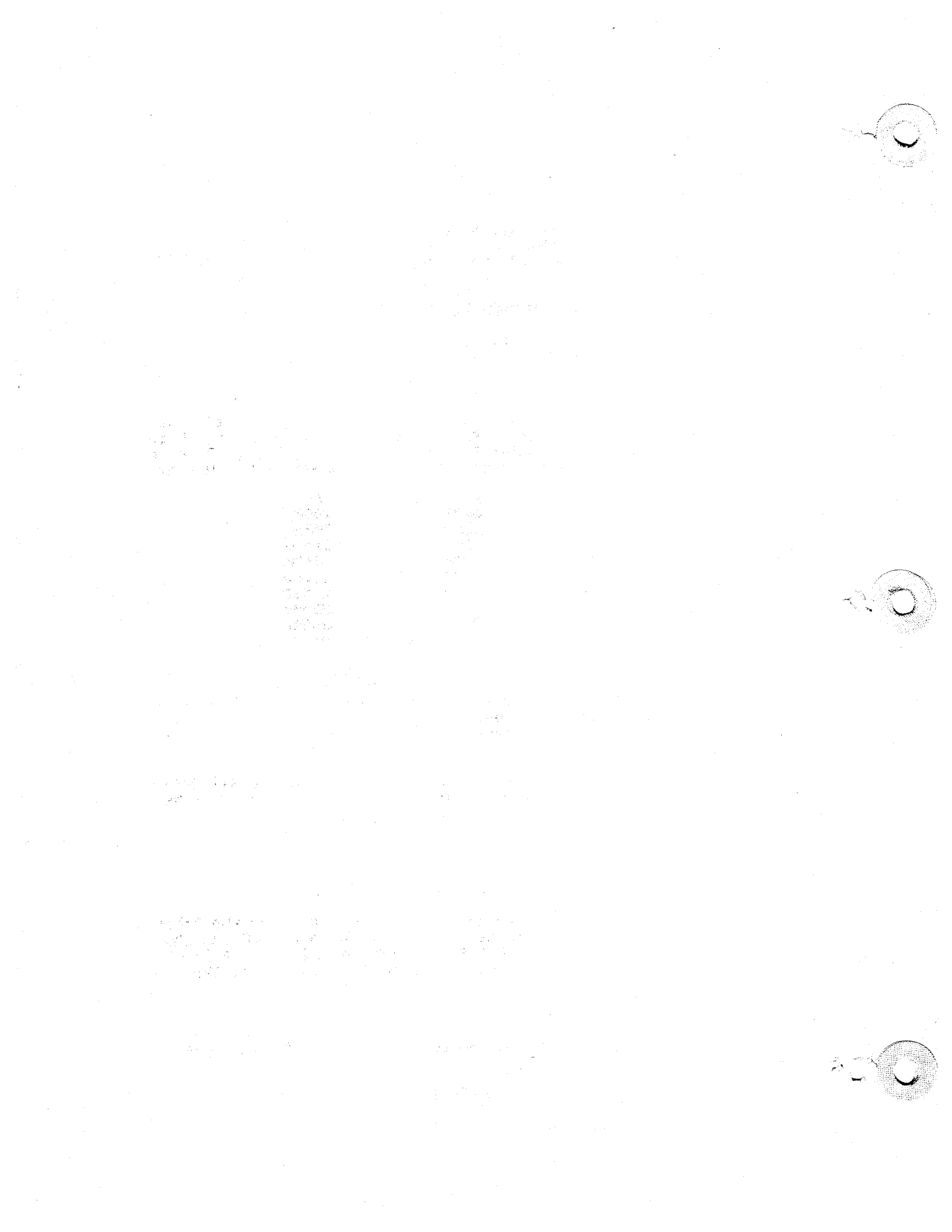
```
/* otherwise, make it an actual */
else    {
        pushint(toptoi());
        leave = NO;
        }
```

3. On page B-58, after line 156,

```
$101: ;276
```

insert the lines

```
JSR %7,TOPTOI
MOV %0,(%6)
JSR %7,PUSHINT
```



Received Nov 24 79

PATCHES FOR HEATH TINY-C

If the version of tiny-c you have is HDOS version 1.02 or later the changes given below for the CHRDY routine have already been made. Also your PPS is provided with LP and PR set at 5000.

PATCHES TO PPS

As provided on the installation disk PPS will not load properly on systems having less than 28K of memory. This is due to the values given in PPS for PR and LP, also systems with larger amounts of memory may not make adequate use of the memory with the values used.

To change the values use the text editor EDIT to change the values to those given in the table below. PR is 1 line before MAIN and LP is 3 lines after MAIN. As distributed PR has a value of 7000 and LP has a value of 5500. The table below gives values of PR and LP for various memory configurations.

MEMORY SIZE	PR	LP
20K	1000	1000
24K	5000	5000
32K	11500	11500
36K	14900	14900
40K	18200	18200
44K	21400	21400
48K	23500	23500
52K	26500	26500
56K	29400	29400

The PPS is not necessary to use tiny-c. A program can be prepared using EDIT and run directly by tiny-c. To do this you must first prepare a library file by using EDIT to delete everything in PPS after the ENDLIBRARY statement. The resulting file can be given the name LIB.TNC. You then write your program using EDIT making sure that the first statement is MAIN.

To run the program you load tiny-c as usual then type .R LIB after the tripple prompt appears type .R FNAME where FNAME is the name you gave to your program. Now typing .G will initiate your program.

PATCHES TO INTERPRETER FOR CHRDY

Two changes have to be made to the interpreter code for the CHRDY routine to work properly. The changes can be made in one of two ways 1) change the source code file TINYC.ASM and reassemble it or 2) use the utility program PATCH.ABS provided by Heath to alter the code on the disk.

1) Changing the source code:

The new code for the CHRDY routine is given below with the changes underlined.

```
CHRDY.  PUSH    H               SAVE HL
```

```
            LXI     H,TEMPC         SEE IF A CHAR ALREADY SAVED
            MOV     A,M
            CPI       0               IF 0 NO CHAR SAVED
            JNZ   CHRDY2              A CHAR WAS SAVED PROCESS IT
            SCALL   .SCIN           CHECK FOR CHAR INPUT
            JC      NOCHR           NO INPUT
CHRDY2      MOV     M,A             HAVE CHAR SAVE IN TEMPC
            CPI     0
            JZ      CHRDY3          HAVE NULL
            JMP     MLCU            MAKE SURE CHAR IS UPPER CASE
CHRDY3      INR     A               A=1 FOR NULL
            POP     H               GET BACK HL
            RET
NOCHR       MVI     M,0             CLEAR TEMPC
            XRA     A
            POP     H
            RET
```

To summarize you want to change the CPI 003 to CPI 0 , and the JZ CHRDY2 to JNZ CHRDY2.

To maintain consistency the version number for the header should also be changed to 1.02.

2) Using the PATCH utility

Heath has provided no instructions on using the PATCH utility, but it is not difficult. Below is listed the dialogue with PATCH when making the necessary changes in tiny-c. It is assumed that the disk on drive 0 containes the file PATCH.ABS and that the disk on drive 1 has a copy of the file TINYC.ABS. The operator responses to the program's querys are underlined. <CR> stands for carriage return and <CTRL-D> means to press the control key and the D key simultaneously.

```
>PATCH <CR>
PATCH Issue #50.00.00.
File Name? SY1:TINYC.ABS <CR>
Address? 066104 <CR>
066104=003/000 <CR>
006105=312/302 <CR>
006106=115/<CTRL-D>
Address? 063125 <CR>
063125=061/ 062 <CR>
063126=040/ <CTRL-D>
Address? <CTRL-D>

PATCH Issue #50.00.00.
File Name? <CTRL-D>
>
```

To summarize you want to change data at address 066104 from 003 to 000, data at address 066105 from 312 to 302, and data at address 063125 from 061 to 062. This last change is simply to change the version number from 1.01 to 1.02 and is not necessary for proper operation of tiny-c.

PATCHES FOR HEATH TINY-C 2 11/28/79

If the version of HDOS tiny-c you have is 1.02 or later the modified GU routine given below has been incorporated in your distribution disk.

ADDITION TO GU ROUTINE OF THE PPS

In order for the .W command to work properly several lines must be added to the GU routine in the PPS. This is the last routine in the PPS. The text of the new GU routine is given below. These changes can be made with the HDOS editor.

```
GU
[
[INT L,K
L=PR+1
K=0
WHILE(L <= (PR+PE))[
IF(PR(K)==0) PR(K)=34
L=L+1
K=K+1
]
]
[FA
FN WRITEFILE(LN+3,PR+1,PR+PE,1)
PL""
]
]
```

This change is necessary because when tiny-c executes a program trailing quotes are changed to nulls. Without the modification given above the nulls are written to the file, and when the file is read they are interpreted as eof characters. The new routine is somewhat slower at writing the buffer to a file than was the original routine.

TABS AND BACKSPACING

Tiny-c does not recognize TAB characters. It is recommended that spaces be used if indenting is desired. It would be possible to add a routine to the GS routine of the PPS that would replace a TAB with a series of spaces, but this would not be effective for programs prepared using the HDOS editor.

The present character removal routine in the PPS is not a true CRT type backspace as provided by HDOS because the system was designed to be as universal as possible. Below is listed a modified GS routine

that will give a true backspace for those having CRT terminals.

```
GS CHAR B(0)
[INT I; CHAR C
I=-1
WHILE (1)
[C=B(I=I+1)=MC 2
IF(C==10) [B(I)=0; RETURN I]
ELSE
IF(C==21) [PS" <<DEL<<"; PL""; I=-1]
ELSE
IF(C==8)
IF(I >=1)
[PUTCHAR(8)
PUTCHAR(' ')
PUTCHAR(8)
I=I-2
]
ELSE I=I-1
]
]
```

If you want the DELETE key to be the character delete instead of CTRL-H Just replace the statement IF(C==8) with IF(C==127).

My apologies for sending this so late!