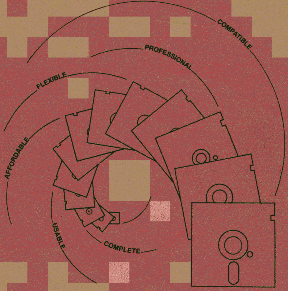
REFERENCE MANUALS

Optimized Systems Software, Inc.

tiny c

For Atari® Computers
©1982, OSS, Inc., ®©1981, Tiny C Associates

NOW WITH OS



Optimized Systems Software, Inc. 10379 Lansdale Ave., Cupertino, CA. 95014, (408) 446-3099

GETTING STARTED WITH OS/A+ AND tiny c

CONGRATULATIONS!!!

You have purchased what we believe is one of the most sophisticated sets of system tools available for Atari Home Computers*.

This package, tiny-c with OS/A+, is designed to be run on any Atari Computer with 48K bytes of RAM and at least one disk drive. Since no OSS software use any routines in any cartridge, you should REMOVE ALL CARTRIDGES.

HOW TO USE YOUR PACKAGE

1. Check the contents of your package. You should have one diskette, labeled "tiny c", and two manuals, "OS/A+" and "tiny c". The bulk of the tiny c manual was written by Tiny C Associates, but there should be a separate appendix, the "Tiny C Installation Guide", included at the end of the tiny c manual.

2. There should be a license agreement. FILL THIS OUT NOW AND RETURN IT TO US IMMEDIATELY!! Aside from its obvious purpose, the agreement is YOUR ticket to SUPPORT from OSS. Yes, we do answer phone questions. Yes, we do respond to bug reports. BUT ONLY for those persons who send back their license! Also, there is a special offer for tiny-c customers who return their license agreement (see below).

3. Turn on your disk drive(s) and screen, leaving the Atari Computer off. Insert the OS/A+ diskette in Drive 1 and turn on the computer. OS/A+ will boot, and you are ready to try tiny-c. Please refer to the installation guide.

4. We STRONGLY urge you to IMMEDIATELY use the DUPDSK program to obtain a working copy of your master disk. Then put your master disk someplace safe for emergencies!

5. Sit back and enjoy a REAL computer system.

SPECIAL OFFER !!!!

For purchasers of the tiny-c package only: When you return your license agreement, you may purchase a copy of the SOURCE AND LISTING of the tiny-c interpreter for only $25, including shipping! ($30 outside USA, please.) We realize that not everyone needs or wants this large (6K bytes of object code) program, but for those experimenters, etc., who do, this is a bargain. Please, at this price we can only accept pre-paid orders (checks or money orders, preferably no charge cards). [If you don't order when you send in your license agreement, you can get it later for $5 more--you must include your agreement number with the order.]

No order form is needed. Just write "SOURCE" in big letters on your agreement form and include the check. Two week delivery guaranteed.

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

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

Appendix:

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

Pages
vii through xi
intentionally [illegible]

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 state-
                         ment.  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>0) [
    while (x<limit) [
        if (case==1) [
            y = 0; w = 99
        ]
        if (case==2) [
            y = 99; w = 0
        ]
        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 nesting 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=0
     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,l) /* 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.

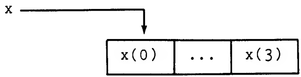
2.6 Pointers

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:

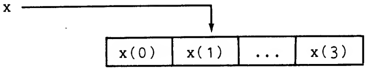


The arrow from x to x(0) indicates that the value in x is the address of x(0).

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,

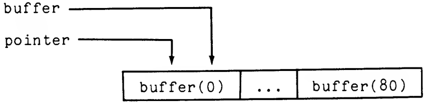
```
          x = x+1
```

results in:

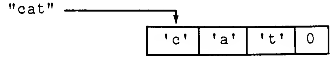


Or:

```
          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.

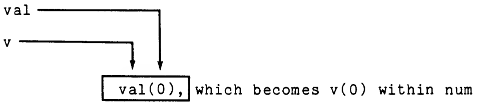
```
          num char b(5);int v(0) [
                .
                .
                .
             v(0) = 0
                .
                .
                .
             v(0) = expression
                .
                .
             return k
          ]
```

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
>.l random
```

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

```
>.l
```

matches line 19, the comment containing the word "random". The following .ls match line 19, line 21, and line 22, while the final .l 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
>.l number
```

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

```
>.l
```

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

```
>.c number num
```

Line 5 is changed. Continue with:

```
>.l
```

which makes line 6 current. Change it by typing:

```
>.c
```

and resume with:

```
>.l
```

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 now5Ø


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

>.l 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 now5Ø

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)
```

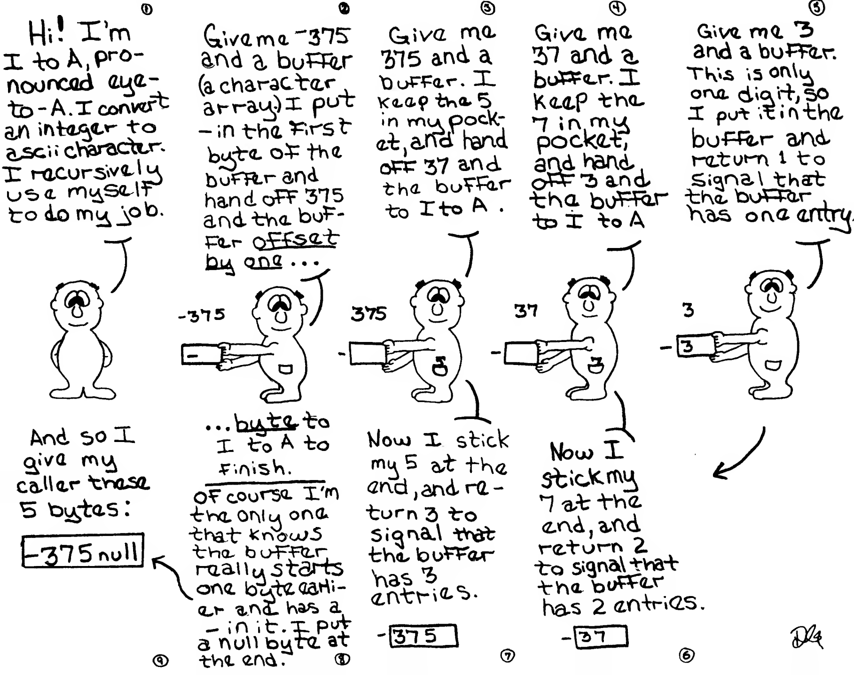
The integer n is converted to an ASCII representation of the integer: a minus (-) if needed, followed by 1 to 5 digits followed by a null byte. The string is put into b. The number of bytes of the string (excluding the null) is returned.

```
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.


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.
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 ...
-375
-
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.
375
-
5
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
37
-
7
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.
3
-3
⑥
Now I stick my 7 at the end, and return 2 to signal that the buffer has 2 entries.
-37
⑦
Now I stick my 5 at the end, and return 3 to signal that the buffer has 3 entries.
-375
⑧
... 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.
⑨
And so I give my caller these 5 bytes:
-375 null
DLG

GW
DG

FIGURE 4-1

```
/* Converts hex to integer.  Returns result in val(0).
/* Returns number of characters scanned as value of htoi.
/* First non-hex character stops the scan.
htoi char b(0)
     int val(0) [
  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(0)=0
  while(1) [
    if(b(0)<'0')break
    else if(b(0)<='9')val(0)=16*val(0)+b(0)-'0'
    else if(b(0)<'A')break
    else if(b(0)<='F')val(0)=16*val(0)+b(0)-'7'
    else break
    b=b+1
    n=n+1
  ]
  return n
]
/* Counts leading blanks in b.
blanks char b(0) [
  int n
  while(b(n)==' ')n=n+1
  return n
]
/* Tests if a is a leading substring of b.  Returns 1 on
/*    true, 0 on false.
ceq char a(0), b(0) [
  while(a(0) != 0) [
    if(a(0) != b(0)) return 0
    a=a+1; b=b+1
  ]
  return 1
]
/* Converts integer to hex.
itoh int n
     char b(4) [
  int k
  b(k=4)=0
  while((k=k-1)>=0) [
    b(k)=n%16+'0'
    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<0) [
    b(0)='-'
    return 1+itoa(-n,b+1)
  ]
  if(n<10) [
    b(0)=n+'0'
    return 1
  ]
  int k
  b(k=itoa(n/10,b))=n%10+'0'
  b(k+1)=0
  return k+1
]
/* Move n bytes from a to b.
moven char a(0), b(0)
      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).

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 1ØØ-dist; ps" yards to go, and is"
      pn full; ps"% full"
      if(random(1,4)==1)onefish
    ]
  ]
  i=Ø      /* 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=Ø   /* 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(Ø,1))[
      pl"Unfortunately he made a hole in the"
      pl"boat, increasing its sink rate 1Ø%."
      sinkrate=sinkrate+sinkrate/1Ø
    ]
  ]
  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 boooooat!"
      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)==0)[]    /* All done if healthy.
    else if(random(0,2))[]   /* All done for .67 of sick.
    else if(health(p)<3)[    /* But some get sicker.
      if(random(0,2)==0)[
        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=1000      /* Perfect score.
  while((p=p+1)<=9)[
    h=health(p)
    if(h==5)s=s-100
    if(h==4)s=s-30
    if(h==3)s=s-15
    if(h==2)s=s-10
    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.

4.3.1 Program Preparation System Code

```
/* 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 l
  while((b(l)=MC(2))!=13) [   /* Do until carriage return.
    if(b(l)==24) [   /* line kill
      l=Ø; pl""
    ]
    else if(b(l)==127) [   /* char kill
      if(l>Ø)l=l-1
    ]
    else l=l+1
  ]
  b(l)=Ø   /* Put null at line's end.
  return l
]
/* Prints a string.
ps char b(Ø) [
  int l
  char c
  l=-1
  while((c=b(l=l+1))!=Ø)MC c,1
  return l
]
```

```
/* 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-lfind,find(Ø),left)
    if(left(Ø))return Ø   /* If no first char, then fail.
    if(ceqn(in+at,find+1,lfind-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)[illegible]
  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(Ø); 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)>=Ø)putchar(' ')
  putchar '<'; pl""
]
/* Moves cursor down.
down [
  int val(1)
  if(line(1)==Ø)val(Ø)=1
  else num(line+1,val)
  lineno=lineno+val(Ø)
  val(Ø)=val(Ø)+1
  cursor=cursor+scann(pr+cursor,pr+progend,13,val)
  lineno=lineno-val(Ø)
  plines(1)
]
/* Moves cursor up.
up [
  int lines,val(1)
  if(line(1)==Ø)lines=1
  else[ num(line+1,val); lines=val(Ø)]
  while((lines=lines-1)>=Ø)bl
  plines(1)
]
/* Move cursor to given line.
goto int l(1) [
  lineno=l(Ø)
  l(Ø)=l(Ø)+1
  cursor=scann(pr,pr+progend,13,l)
  lineno=lineno-l(Ø)
  plines(1)
]
/* Print some facts.
facts [
  pn lineno
  pn lastline
  pn progend
  pn lpr-progend
  pl""
]
```

```
/* Enter an application program.
start [
  while(llline<=64) [
    line(llline)=' '
    llline=llline+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+lpr,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 1002
  MC 7,1
  int k
  while((k=k+1)<3)[]
  MC 1003
]
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)<20)[]
]
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(Ø) [
    SPEED=12
    while(s(Ø))[
      letter s(Ø)
      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=Ø
      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.

DAD, ARE THERE SUCH THINGS AS PIRANHA FISH?
WHAT'S A CANNIBAL?
ARE THERE PIRANHAS IN NEW JERSEY?
I WANNA PROGRAM A GAME NOW... CAN I?
UH-OH! I PUSHED "DEL" BY MISTAKE.
BEEP... BEEP! BE... BEEP!
OH NO...
I GET CONFUSED IS IT I to A, A to I, OR BOTH?
WOW! THE METEOR HIT 5 STARS!
DAD...

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 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

INSTALLATION GUIDE

OSS tiny-c for Atari Computers

OPTIMIZED SYSTEMS SOFTWARE

January, 1982

TABLE OF CONTENTS

SECTION Page

1. INTRODUCTION

The OSS tiny-c for Atari Computers is an implementatation of the tiny-c language interpreter developed by tiny-c Asscociates. tiny-c was originally developed for the 8080 microprocessor and has been since migrated to many other microcomputers. The tiny-c owner's manual (included with your OSS tiny-c pcakage) is the prime source of information about the tiny-c language for all implemenatations. The Installation Guide contains information which is specific to the OSS version of tiny-c for Atari computers.

2. DISK CONTENTS

Your disk contains two sets of files. The first set of files pertain to the OSS OS/A+ Operating System. These files are DOS.SYS, DUPDSK.COM. INIT.COM, HELP.COM, COPY.COM, SYSEQU.ASM, and RS232.OBJ. The OS/A+ User's Manual (included with your OSS tiny-c package) describes OS/A+ and elucidates the usage of these files. The second set of files pertains to tiny-c. These files ate TC.COM, PPS.C, and LIST.C. The TC.COM file is the tiny-c interpreter. The xxx.C files are tiny-c source programs.

3. STARTING and RUNNING

In order to run tiny-c, you must first insure that your Atari Computer is properly configured. This requires:

1) 48K of RAM installed.
2) Both the A and B ROM cartrdige slots to be empty.
3) An 810 disk drive installed.

When your system is properly configured, boot the tiny-c disk in the normal manner. The first message that will appear on your screen is the OSS OS/A+ message:

```
OSS OS/A+ - AATARI version 1.2
Copyright (c) 1981  OSS

D1:[cusror]
```

The next step is to load and execute the tiny-c interpreter. This is done by typing:

```
TC[return]
```

The tiny-c interpreter greeting message will then appear:

```
*** TINY-C+ Version 1.1 ***
Copyright (c)  1979 - TINY-C Associates
Copyright (c)  1981 - OSS
>[cursor]
```

The '>' is the tiny-c interpreter prompt. The following list is the commands available to the interpreter:

```
.x            exit the interpreter and return to OS/A+
.g            link and run the loaded program
.r [filspec]  load the [filespec] tiny-c program
```

The .r command should be used at this time to load the Program Preparation System (PPS). To do this, enter:

```
.r D:PPS.C[return]
```

Note: All input to tiny-c is via the Atari screen editor, thus the normal edit features such as back arrow, delete, etc. are available.

When the PPS has been loaded (the > prompt returns), begin executing PPS by entering:

```
.g[return]
```

The screen will then be cleared and the tiny-c greeting message will appear as before - except that the prompt character will now be '#'. At this point you can begin to use PPS to enter and execute tiny-c programs.

WARNING! - Any time you hit RESET, your machine will take you back to OS/A+. To get back into tiny-c, you must repeat the entire process of loading PPS and restarting it. You will lose any program that may have been in memory before the RESET key was hit.

4. PPS DIFFERENCES

The OSS version of the tiny-c PPS has an expanded set of commands intended to provide access to DOS, to facilitate the use of the Atari Screen Editor, and make the PPS generally easer to use.

File Load and Save:

The loading and saving of tiny-c source programs prepared under PPS is accomplished with the .r and .w commands. The command formats are:

```
.w [filespec]
.r [filespec]
```

The [filespec] is the standard Atari DOS file specification for a disk file:

```
D1:LIST.C
D1:PPS.C
D1:MYPROG.C
```

The [filespec] must be specified in capital letters. The '.C' appendage is recommended for all tiny-c source programs.

Statistics:

The orignial PPS './' statistics command presents the information is a rather cryptic format. The OSS PPS presents the information in a self explanatory manner.

DOS Commands:

The OSS PPS has two useful DOS type commands.

```
.f n          Display the filenames on the disk which is in drive n. If n is omitted, it will default
              to 1.
.e [filespec] Erase (delete) the given [filespec].
```

Modify Command:

The OSS tiny-c interpreter gets its keyboard input from the powerful Atari Screen Editor. The .m command has been added to PPS to facilitate the use of this screen editor. To use the .m command:

- display the current line using the .p command (see tiny-c User's Guide).
- modify the line using the edit keys.
- hit return
- enter .m[return].

The line will then be displayed as it was before the change, and then as it is now after modification. The current line will be the modified line.

5. LANGUAGE DIFFERENCES

The OSS tiny-c interpreter has been modified to permit the use of hexidecimal constants. This change was made to allow easy access to the many and wonderful graphics and sound features of the Atari computer. A hexidecimal constant may be used anywhere an integer or character constant is used. The form of a hexidecimal constant is $hhhh. The '$' indicates that the constant is hexidecimal. The hhhh characters represent a string of one to four hexidecimal digits (0 through 9, and A through F).

Error codes returned to tiny-c from the ATARI operating system are single byte negative numbers. The tiny-c interpreter will extend these values to be a negative integer. The values that you are used to seeing are the positive form of the signle byte value. For example, the error code $81 can be error 129 or error -127. If you print the value in tiny-c you will see -127. To translate this value to the familiar 129, you must add the (negative) error code to 256. If the variable 'errcode' contained the $81 error, then you should print 'errcode+256'.

6. MC and LIBRARY DIFFERENCES

The OSS tiny-c MC routines and its associated library of tiny-c functions have a number of modifications and extensions from those shown in the tiny-c User's Guide. The entire OSS tiny-c library is presented below.

putchar char c

Puts one character onto the sceen via the E: device.

getchar

Gets the next character from the current screen input line via the E: device. The value returned by the function is the character received. Implements MC #2.

pft char from(0),to(0)

Displays all characters from the 'from' pointer to and including the 'to' pointer via the E: device. The displayed string may contain zero or many EOL ($9B) characters. The value returned is the total number of characters displayed. Implements MC #13

gs char buff(0)

Reads the next input line from the E: device. The EOL character is changed to a null ($00) character. The value returned is the number of characters in the line excluding the EOL. Implements MC #4.

ps char buff(0)

Displays on the screen via the E: driver the characters from the character pointed to by 'buff' to (but not including) the first null character encountered. The value returned is always zero.

pl char buff(0)

Writes an EOL to the screen via the E: device, then displays the string pointed to by 'buff' via the 'ps' function.

num char buff(5); int value(0)

Converts the ATASCII numeral characters pointed to by 'buff' to an integer and places the integer in the element pointed to by value. The conversion ends when either 5 digits have been converted or a non-numeric character is is encountered.

atoi char buff(0); int value(0)

Converts the ATASCII numeral characters pointed to by 'buff' to an integer and places the integer in the element pointed to by value. The conversion will skip leading blanks and will recognize and implement plus (+) and minus (-) signs.

itoa int num; char buff(0)

Converts the integer 'num' to an unsigned ATASCII decimal string and places the result in the character elements starting at the address pointed to by 'buff'. The ATASCII string will be terminated with a null byte. The value returned is the number of digits in the number. Implements MC #14.

htoa int num; char buff(0)

Converts the integer 'num' into an ATASCII hexidecimal string and places the result in the character elements starting at the address pointed to by 'buff'. The ATASCII string will be terminated with a null byte. The value returned is the number of digits in the number and is always four. Implements MC #15

pn int num

Displays the integer 'num' on the screen via the E: driver. The number will be preceeded by a single blank character.

gn

Gets a numeric string from the next input line (via the E: driver) and converts the number to an integer. The value returned is the integer read.

ceqn char str1(0),str2(0);int length

Compares the character elements pointed to by 'str1' to the character elements pointed to by 'str2' for a length of 'length' charactes. If all the elements are equal a true value (1) is return - otherwise a false value (0) is returned.

index char str1; int ls1; char str2; int ls2

Searches the first 'ls2' elements for 'str2' of the 'ls1' characters of 'str1'. If 'str1' is not found in 'str2' then a zero value is returned. If 'str1' is found in 'str2', then the value returned is the number of characters into 'st2' where the first character of 'st1' was found.

move char dest(0),src(0)

Moves the source string 'src' to the destination string 'dest'. The move ends after a null byte has been moved.

gc

Gets the next non-EOL character from the current input line on the screen via the E: device. The value returned is the value of the character read.

movebl char first(0),last(0); int num

Moves the block of memory pointed to by 'first' and terminated by the element pointed to by 'last' up or down in memory. If (num>0) then the block is moved up num bytes higher (towards $FFFF) in memory. If (num<0) then the block is moved num bytes down in memory. The move in either direction is non-destructive. Implements MC #7.

countch char first(0),last(0), c

Counts the number of times the character 'c' appears in the block of memory starting at 'first' and ending at 'last'. The value returned is the count.

scann char first(0),last(0),c; int count(0)

Scans the block of memory from 'first' to 'last' for the character 'c'. Each time 'c' is found, the integer pointed to by 'count' is decremented by one. When count(0)=0, or when last(0) is examined, the scan stops. The value returned is the number of characters scanned.

readfile char name(0),start(0),end(0);int iocb

The file whose filespec (ie. "D:LIST.C") is pointed to by 'name' is read into memory starting at the address pointed to by 'start' using the IOCB specified by 'icob'. The address pointed to by 'end' is the upper limit of the read. The value returned is the number of bytes read from the file. In no case will more then (end-start+1) bytes be read. The function will open and close the file.

writefile char name(0),start(0),end(0);int iocb

The data starting at the memory address pointed to by 'start' and ending at the address pointed to by 'end' will be written to the file whose filespec (ie. "D:TEST.C") is pointed to by 'name' using IOCB #'iocb'. The value returned is the total number of bytes written. This value should be (end-start+1). The function will open the file (mode 8) and close it.

fdel char name(0)

The file whose filespec is pointed to by 'name' will be deleted. IOCB #7 is used. The value returned is the condition code returned by CIO.

flock char name(0)

The file whose filespec is pointed to by 'name' will be locked. IOCB #7 is used . The value returned is the condition code returned by CIO.

funlock char name(0)

The file whose filespec is pointed to by 'name' will be unlocked. IOCB #7 is used. The value returned is the condition code returned by CIO.

finit char name(0)

The disk drive whose filespec is pointed to by 'name' will be caused to perform a disk format operation. IOCB #7 is used. The value returned is the condition code returned by CIO.

fdir int n

The directory of the diskette in drive 'n' will be displayed upon the screen. IOCB #7 is used.

fopen int mode; char name(0); int aux2; int iocb

The file whose filespec is pointed to by 'name' will be opened in the mode specified by 'mode' (input=4, output=8, input/output=12). The 'aux2' value will be placed in the IOCB auxillary #2 cell. IOCB #'iocb' will be used. The value returned is the condition code returned by CIO. Note: if the most significant byte of 'aux2' is non-zero, then the most significant byte of 'aux2' is used as the IOCB command byte rather then the OPEN command thus allowing the user to implement device dependent ("XIO") command calls. In either case, the 'mode' value is placed in the IOCB auxillary #1 cell. Implements MC #3.

fclose int iocb

The IOCB indicated by 'iocb' will be closed. The value returned is always zero.Implements MC #6.

fwrite char from(0),to(0);int iocb

Data bytes starting at the memory location pointed at by the 'from' pointer are written to the previously opened file in IOCB #'iocb'. If the value of 'to' is non-zero, then (to-from+1) bytes will be written including all EOL characters. If the value of 'to' is zero, then the write operation terminates after the first EOL character is written. The value returned is the number of bytes written. If the value returned is less then zero, then an error has occured and the value is the error code returned by CIO. Implements MC #5.

fread char from(0),to(0); int iocb

Data bytes are read from the previously opened file in IOCB #'iocb' to the address starting at 'from'. If the value of 'to' is zero, then the read will terminate after the first EOL character is read. If the value of 'to' is non-zero, then (to-from+1) bytes will be read unless the end of file is encountered. In either case, the value returned is the number of bytes actually read. If the value returned is less then zero, then an error has occurred and the value is the error code returned by CIO. Implements MC #4 dos

Tiny-c will pass control to OS/A+ via DOSVEC in location $0A. Implements MC #10.

mcall char adr(0);int regs(0)

The machine language subroutine located at 'adr' will be called via the 6502 JSR instruction. The 6502 registers will be loaded from the values pointed to by 'regs' before the call is made. (regs(0)=ACU; reg(1)=X; reg(2)=Y). Upon return from the subroutine, the returned values in the 6502 registers will be stuffed into their respective locations in 'regs'. The value returned will be flag bits in the P register upon return from the subroutine. Implements MC #12.

7. MEMORY MAP

Zero Page	$0080-$00B1	Variables and Registers
OS/A+	$0700-$1EFF	OS/A+ Operating System
User Memory	$1F00-$A5FF	tiny-c User Memory Area
Interpreter	$A600-$BBFF	tiny-c Interpreter

The tiny-c interpreter configures the tiny-c User Area starting at the location pointed to by the Atari system MEMLO vector (at $2E7). The Atari 850 driver routine (RS232.OBJ) will (if loaded) change MEMLO to $25E2, thus making the tiny-c user area smaller.

8. MACHINE LANGUAGE ROUTINES

The OSS tiny-c Source Package (available from OSS) has a detailed explanation of the tiny-c interpreter and a copy of the interpreter in machine readable form. It is reccommended that you obtain this package if you wish to create special MC functions. In lieu of this, you can use the MCALL function (MC 14) to access any user or system machine language subroutine. If you wish to do this, the routines should be loaded at MEMLO ($1F00 or $25E2) and MEMLO adjusted above your routines before executing the interpreter.

The tiny-c interpreter is located just below the Atari screen memory. If you wish to use an Atari Graphics mode that will use over 1k of screen RAM, you must relocate the screen memory somewhere below the tiny-c interpreter. The way to do this is to set RAMTOP ($6A) to $A4 and then open the screen in the graphics mode you wish to use.

Example:

```
gr7()
    [
    /* poke RAMTOP with $A4
    char ramtop (0); int errcode
    ramtop= $6A; ramtop(0)=$A4
    /* open screen (S:) on IOCB #6 (Atari standard)
    /* for read/write (12) with mixed char/graphics (+16)
    /* in mode 7 graphics
    /* equivalent to GR.7 in Atari BASIC and BASIC A+
    errcode=fopen (12+16, "S:", 7, 6)
    if (errcode)<0 pn errcode+256
    return errcode
    ]
```