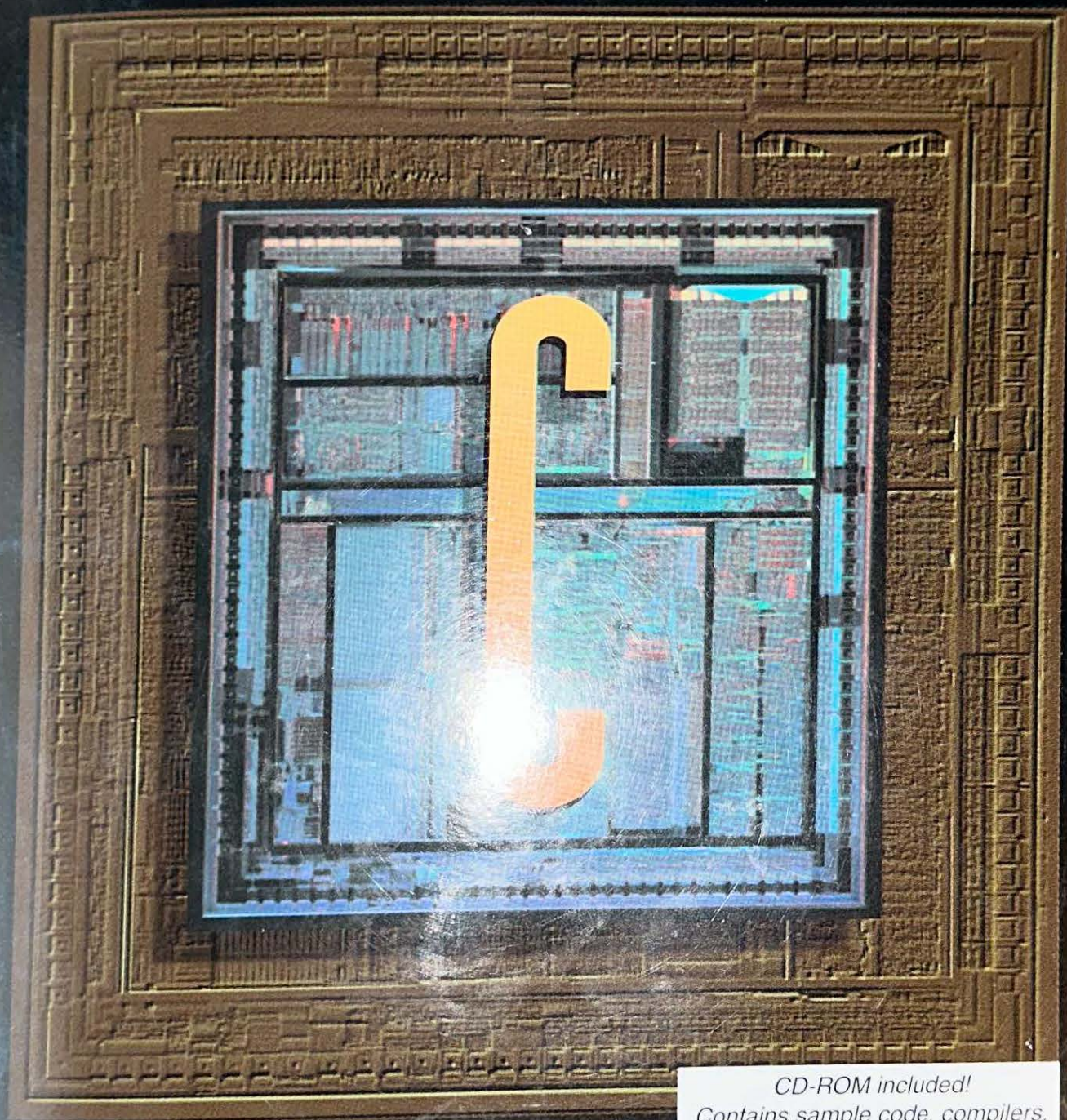


TB328

Programming Microcontrollers in



CD-ROM included!
Contains sample code, compilers,
and a complete searchable
version of the book!

Ted Van Sickle



MOTOROLA
Series in Solid State Electronics

Motorola reserves the right to make changes without further notice to any products herein to improve reliability, function, or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights nor the rights of others. Motorola products are not authorized for use as components in life support devices or systems intended for surgical implant into the body or intended to support or sustain life. Buyer agrees to notify Motorola of any intended use whereupon Motorola shall determine availability and suitability of its product or products for the use intended. Motorola is a registered trademark of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

Copyright © 1994 by LLH Technology Publishing. All rights reserved. No part of this book may be reproduced, in any form or means whatsoever, without permission in writing from the publisher.

The material in Appendix D is reprinted with permission of Motorola, Inc., Semiconductor Products Sector.

ISBN: 1-878707-14-0

Library of Congress catalog number: 94-075549

Motorola Series in Solid State Electronics number: TB328

Printed in the United States of America.

Cover design: Brian McMurdo, Ventana Studio, Valley Center, CA
Interior design and production services: Greg Calvert, Artifax, San Diego, CA

LLH

Technology Publishing

Eagle Rock, VA

www.LLH-Publishing.com

This program performs exactly the same function as the earlier one. The data are read in a character at a time as before. Here, however, the `switch` statement is used. The statement `switch(c)` causes the argument of the `switch` to be compared with the constants following each of the `case` statements that follows. When a match occurs, the next set of statements to follow a colon will be executed. Once the program starts to execute statements, all of the following statements will be executed unless the programmer does something to cause the program to be redirected. The `break` instruction does exactly this operation for us. When a C program encounters a `break`, it jumps to the end of the current block. Therefore, the `breaks` following the executable statements above will cause the program to jump out of the executing sequence and return to get the next character from the input stream.

When all options have been exhausted without a match, the statements following the default line will be executed. It is not necessary to have a default line.

EXERCISES

1. Write a program that counts the number of lines, words, and characters in an input stream.
2. Extend the program from the exercise above to calculate the percentage usage of each character in the alphabet.
3. A prime number is a number that cannot be evenly divided by any number. For example, the numbers 1, 2, and 3 are all prime numbers. Write a program that will calculate and print out the first 200 prime numbers.

Functions

The *function* is the heart of a C program. In fact, any C program is merely a function named `main`. The purpose of a function is to provide a mechanism to allow a single entry of a code sequence that is to be repeated many times. A function is the most reusable element in the C language. Properly written and debugged functions can be collected into a program when needed. Therefore, the use of functions will allow the programmer to write smaller programs and it is not necessary to rewrite common functions that are used often.

A function can have many arguments or none whatsoever. Function arguments are contained in parentheses following the function name. The values of the arguments are the parameters needed to execute the function. A function can return a value, or perhaps it will not have a return value. An example of a function that returns a value is `getchar()` which will return a character from the input stream. A function that will return no value is `putchar(c)` which puts the character `c` onto the output stream.

A function may not be nested inside another function. Therefore, any function must be created outside of the boundaries of any other function or program structure. Functions have only one entry point, and they return only one return item. The return can be of any type that C supports. Functions can have several arguments. The arguments can be of any valid C type.

In ANSI C, the use of a function requires the use of a *function prototype*. A function prototype is a statement of the following form:

```
type function_name(type, type, type,...);
```

The first type preceding the function name is the type to be returned from the function. The several types found in the argument are the types of the corresponding arguments that are sent to the function. Variable names may or may not be used for the arguments of a function prototype. The type list is the important item. The types in the argument list are separated by commas.

Thus far, it might seem that we have been blindly using functions like `printf()`, `getchar()`, and `putchar()` without the benefit of function prototypes. Not so! The header file `stdio.h` contains the function prototypes of all input/output related functions, so it is not necessary for you to put a function prototype in your code for these functions.

Compilers will differ. If a programmer attempts to send the wrong type of data to a function through its argument, the compiler might consider it an error or it might well convert the argument to the correct type prior to calling the function. In either case, the compiler will not let a program use the wrong type of data as an argument to a function.

Data returned from a function will always be converted to the correct type before it is passed to the program. If you wish to have a different type returned, the cast operator can be used to change the return data type to any type desired.

ANSI C defined the type `void`. This type is used in several different ways. If there is no function return, the prototype must identify the function as type `void`. Also when there are no function arguments, the argument list must contain the type `void`. This use of the keyword `void` will prevent problems in function calls.

Note that the function prototype above is terminated with a semicolon. The semicolon is needed in the function prototype, but it is not to be used after the name of the function in the code where the function is defined.

The philosophy in C is to use functions with little provocation. Using many functions produces code that is easy to read and follow. Often it is easier to debug many small functions rather than a larger program. One must temper these ideas somewhat when writing code for small microcontrollers. Calling a function requires some overhead that is repeated each time the function is accessed. If the total overhead is more than the length of the function, it is better to use *in-line* code. In-line code implies that the function code is repeated in-line every time that it is needed. If the function code is much greater than the calling overhead, the function should be

used. In between these limits, it is difficult to determine a hard-and-fast rule. In microcontroller applications, it is probably best to use function calls to a single function if there is a net savings of memory as a result. This savings is calculated by first determining the code needed prior to calling the function, the code needed to clean up the process after the function call, the number of times the function is called, and the length of the function. The in-line code will be smaller than the corresponding function code. Therefore, if the total code for the number of function calls listed first exceeds the total in-line code required to accomplish the same operations, then use the in-line code. Otherwise, use function calls.

The above argument is valid for microcontroller applications code. It does not necessarily follow for code written for large computers. When writing for a large computer, there are usually few memory constraints. In those cases, it is probably best to use more function calls and not be worried about the memory space taken up by function calls unless there is a serious speed constraint. When speed is a problem, the programmer must go through an analysis similar to that above with the dependent parameter being time rather than memory space. In small computers where several registers can be saved and restored when a function is entered and exited, single instructions can require many clock cycles. When deciding whether to use a function or in-line code, the programmer must assess the total time lost to entering and exiting a function each time it is entered, and weight that time lost as a fraction of the total time the program resides in the function. If this time is large, and the program requires too much execution time, consider the use of in-line functions.

It is always good to write small functions and create simple calling programs to exercise the small functions. These programs are used to debug the functions, and they are discarded after the functions are debugged. If later, the program constraints dictate that in-line code should be used, the essential code of the function can be written into the program wherever it is needed. Another approach that will be discussed in the next chapter is to use a macro definition to specify a small function. With a macro definition, the function code is written in-line to the program whenever the function is invoked.

Let us revisit an example used earlier. Write a program to calculate and display the square root of each integer less than 11:

```
/* Calculate and display the square roots of numbers
1 <= x <=10 */

#include <stdio.h>

#define abs(t) (((t)>=0) ? (t) : -(t))
#define square(t) (t)*(t)

double sqr ( double );
```

```

void main(void)
{
    int i;
    double c;

    for(i=1;i<11;i++)
    {
        c=sqr(i);
        printf("\t%d\t%f\t%f\n",i,c,square(c));
    }
}

double sqr( double x )
{
    double x1=1,x2=1,c;

    do
    {
        x1=x2;
        x2=(x1 + x/x1)/2;
        c=x1-x2;
    }
    while(abs(c) >= .00000001);
    return x2;
}

```

The result of this calculation is shown below:

1	1.000000	1.000000
2	1.414214	2.000000
3	1.732051	3.000000
4	2.000000	4.000000
5	2.236068	5.000000
6	2.449490	6.000000
7	2.645751	7.000000
8	2.828427	8.000000
9	3.000000	9.000000
10	3.162278	10.000000

The second line of code

```
#define abs(t) ((t)>=0) ? (t) : -(t))
```