The Small-C Handbook

James E. Hendrix

The Small-C Handbook

James E. Hendrix
Office of Computing and Information Systems
The University of Mississippi

A Reston Computer Group Book
Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia

A copy of the compiler described in this book may be obtained by sending $25 (check or money order) to:

J. E. Hendrix
Box 8378
University, MS 38677-8378

Distribution is on standard 8-inch SSSD CP/M diskettes containing all source and object files for Small-C version 2.1 implemented for use with Microsoft's MACRO-80 assembler package. Add $3 for orders outside of the United States.

ISBN 0-8359-7012-4

A Prentice-Hall Company
Reston, Virginia 22090

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

Preface

We tend to like the programming language we learn first and to accept new ones with great reluctance. It seems we would rather suffer than learn a better way, especially if it means learning a new language. So it is noteworthy when a new programming language overcomes these tendencies and receives enthusiastic acceptance from experienced programmers. Such is the case with the C language. Its popularity is booming, and many microcomputer programmers are looking for low-cost ways of getting started in C. Since 1980, the Small-C compiler has satisfied that desire. It offers a respectable subset of the C language at a rock-bottom price. For applications not requiring real numbers, it offers clear advantages over BASIC and assembly language. And programs written for Small-C are upward-compatible with full-C compilers.

Anyone using or considering Small-C will find here a valuable resource of information about the language and its compiler. This material should appeal primarily to three classes of readers:

1. Small-C programmers who need a handbook on the language and the compiler.

2. Assembly language programmers who wish to increase their productivity and to write portable code.

3. Professors and students of computer science. Its small size and the fact that it is written in C instead of assembly language make Small-C an ideal subject for laboratory projects. Here is a real compiler that is simple enough to be understood and modified by students.

Without much difficulty, it may be transformed into a cross-compiler or completely ported to other processors. Additional language features may be added, improvements made, etc. Any number of projects could be based on the little compiler.

This book is not an introductory programming text. Instead, it is a description of the Small-C language and compiler for people who are already programming in other languages. Commensurate with that objective, the text is brief and to the point. Section 1 covers program translation concepts. Beginning with the CPU, it presents a survey of machine-language concepts, assembly language, and the use of assemblers, loaders, and linkers. Enough information is given to permit a complete understanding of the assembly language code generated by the compiler. This material is based on the 8080 CPU since the compiler was originally written for that processor and it is still the most popular implementation of Small-C. Chapter 1 describes the 8080 CPU, and Chapter 3 presents the 8080 instruction set. These two chapters are required reading for anyone not already familiar with 8080 assembly language programming.

Section 2 introduces the Small-C language. Its chapters present the elements of the language in an order that builds from simple to comprehensive concepts. Emphasis is placed on accuracy and brevity, so that these chapters meet two needs: they provide a quick but thorough treatment of the language, and also serve as reference material.

Section 3 describes the compiler itself. Chapters deal with I/O concepts, standard functions, invoking the compiler, code generation, program efficiency considerations, and how to use the compiler to generate new versions of itself.

Finally, there are appendices containing the entire source listings of the compiler and its library of arithmetic and logical routines. There are also appendices designed for quick reference by the Small-C programmer.

My sincere appreciation goes to those who encouraged and assisted me in this work. To Marlin Ouverson, who conceived the need for such a book and convinced the publisher. To Ron Cain, who created the original Small-C compiler and provided many useful guidelines for developing the current version. To Ernest Payne, who provided the impetus and most of the code for developing the CP/M library for version 2.1. To Neil Block for his assistance in developing various features introduced with version 2.0—mainly the new control statements. To Andrew Macpherson and Paul West for reporting

several bug fixes and enhancements. To Jim Wahlman and George Boswell, who proofed the text for content and grammar, respectively. To Hal Fulton for his assistance with the galley proofs. And finally, to my wife Glenda, who dealt so patiently with all that I neglected while this was in the making.

Introduction

The C programming language was developed in the early seventies by Dennis M. Ritchie of Bell Telephone Laboratories. It was designed to provide direct access to those objects known to computer processors: bits, bytes, words, and addresses. For that reason, and because it is a block-structured language resembling ALGOL and Pascal, it is an excellent choice for systems programming. In fact, it is the language of the UNIX operating system.

C is good for other uses, too. It is well suited to text processing, engineering, and simulation applications. Other languages have specific features which in many cases better suit them to particular tasks. The complex numbers of FORTRAN, the matrix operations of PL/I, and the sort verb, report writer feature, and edited moves of COBOL come to mind. C has none of these. Nevertheless, C is becoming a very popular language for a wide range of applications, and for good reason—programmers like it.

Those who use C typically cite the following reasons for its popularity: (1) C programs are more portable than most other programs; (2) C provides a very rich set of expression operators, making it unnecessary to resort to assembly language even for bit manipulations; (3) C programs are compact, but not necessarily to the point of being cryptic; (4) C includes features which permit the generation of efficient object code; and (5) C is a comfortable language in that it does not impose unnecessarily awkward syntax on the programmer.

UNIX is a registered trademark of Bell Telephone Laboratories

For a description of the complete C language as implemented under the UNIX operating system, the reader is referred to *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie (Englewood Cliffs, N.J : Prentice-Hall, 1978).

In May of 1980, *Dr. Dobb's Journal* ran an article entitled "A Small C Compiler for the 8080s." In the article, Ron Cain presented a small compiler for a subset of the C language. The most interesting feature of the compiler, besides its small size, was the language in which it was written—the same language it compiles. It was a self-compiler. It could be used to compile new versions of itself. With a simple, one-pass algorithm, it generated assembly language code for the 8080 processor. Being small, it had its limitations. It recognized only characters, integers, and one-dimensional arrays of each. The only loop-control statement was the **while** statement. There were no Boolean operators, so the bitwise logical operators : (OR) and & (AND) were used in their place. But even with these limitations, it was a very capable language and a delight to use compared to assembly language.

Ron Cain published a complete listing of the compiler and graciously placed it in the public domain. Both the compiler and the language came to be known as Small-C. His compiler created a great deal of interest, and soon found itself running on processors other than the 8080.

Recognizing the need for improvements, Ron encouraged me to produce a second version, and in December of 1982 it appeared in *Dr. Dobb's Journal*. The new compiler augmented Small-C with (1) code optimization, (2) data initialization, (3) conditional compilation, (4) the **extern** storage class, (5) the **for**, **do/while**, **switch**, and **goto** statements, (6) assignment operators, (7) Boolean operators, (8) the one's complement operator, and various other features. This book describes an updated version (2.1) of that compiler and its language.

Section 1 is a survey of program translation concepts. If you are already familiar with the use of compilers, assemblers, loaders, and linkers, you may wish to proceed directly to Section 2. If you are not familiar with the 8080 CPU, however, you should read Chapters 1 and 3 dealing with the 8080 and its instruction set before proceeding to Section 2.

Section 2 describes the Small-C language. The order of presentation moves from simple to complex. Each aspect of the language is given a chapter to itself and builds on preceding material. The result is a section which both introduces the language and serves as reference material.

Section 3 is dedicated to the practical aspects of using the language and the compiler.

Seven appendixes finish out the volume. Appendix A is a complete source listing of the compiler Appendix B is a listing of the logical and arithmetic library Appendix C lists areas of possible incompatibility with the full-C language, referring to the body of the text for further details Appendix D lists and explains the error messages produced by the compiler Appendix E is a code chart of the ASCII character set Appendix F is a quick reference summary for 8080 assembly language programming And Appendix G is a quick reference summary of the Small-C language and function library

Contents

Figures

Tables

Listings

Section 1

Program Translation Concepts

The term *program translation* denotes the general process of translating a program from source language into actions. Two basic concepts are involved: *generation* and *interpretation*.

Generation is the process of translating programs from one language to another that is closer to *machine language*, the language of the computer's central processing unit (*or CPU*). Compilers, assemblers, and loaders are generative program translators.

Interpretation is the final stage of program translation. It involves translating programs from language into actions. This stage is also called *program execution*. One *executes* or *runs* a program to make it perform its intended function. Interpretation may be done by another program (an *interpreter*) or by the CPU. The CPU scans machine-language programs in memory, performing the instructions it finds. This is always the last stage of program translation since, even if a program is being interpreted by software, the interpreter itself is being interpreted by the CPU.

Perhaps the best way to go about understanding the program translation process is to work from the CPU upward. That was the historical sequence, and it seems most natural to proceed that way. To save time, we will look at an actual CPU, the Intel 8080—the same one used by the original Small-C compiler.

Chapter 1

The 8080 Processor

Figure 1-1 is a diagram of the 8080 central processing unit and memory. Memory may be considered a simple array of eight-bit bytes. Each byte has a unique address which may be expressed as a 16-bit unsigned binary integer. The first byte is at address 0, the second at address 1, the third at address 2, and so on. The highest possible address is 65535 decimal (FFFF hex). The values stored in memory may represent either data or instructions that direct the operation of the CPU

Two consecutive bytes may be taken together as a single 16-bit number and, as such, may represent either data or the address portion of an instruction These long numbers are called *words.* They are always stored with the low-order byte first (the byte having the lowest address), and the high-order byte following. The CPU is able to *read* the value of a memory byte or word by transferring it to a CPU register (described below). When that happens, the previous contents of the register are lost, and the value in memory remains unchanged. The CPU may also *write* a value into a memory byte or word by transferring it from a register. In that case the register remains unchanged, and the original value in memory is replaced. In both cases the CPU must send the address of the desired byte or word to the memory unit. The address of a word is the address of its first (low-order) byte.

The CPU may be viewed as a collection of *registers* (storage places) which temporarily holds the values of bytes or words. Registers are faster than memory, and the instruction set of the CPU is designed to manipulate register values with greater flexibility. The CPU registers have the names A, B, C, D, E, F, H, L, PC, and SP. The single-letter registers are eight bits wide, and the PC and SP registers each have 16

bits. Some instructions treat the A, F, B, C, D, E, H, and L registers as four 16-bit register pairs: AF, BC, DE, and HL. In such cases, the register denoted by the first letter of the name contains the high-order byte, and the one denoted by the second letter holds the low-order byte. Thus, when a 16-bit number is in the HL register pair, H contains the most significant bits and L the least.

The F register is special, since it is not used to hold data. Rather, it is a collection of *condition flags:* bits indicating the conditions produced by the most recent arithmetic or logical instruction. Each flag bit has a name. The zero flag Z is *set* (contains the value one) if the last arithmetic or logical instruction resulted in a value of zero; otherwise, it is *cleared*, or *reset* (contains the value zero). The sign flag S gives the sign of the result; it matches the high-order bit of the result—one for negative values and zero for positive values. The carry flag CY is set if there is a carry out of (or a borrow into) the leftmost bit position; otherwise, it is reset The parity flag P serves two purposes. Bitwise logical instructions set it or clear it according to the parity of the result. An even number of ones in the result (*even parity*) sets P, and an odd number of ones (*odd parity*) clears it. The eight-bit arithmetic instructions set P in case of an overflow condition, otherwise they clear it. These flag bits may be tested by conditional *jump*, *call*, and *return* instructions, of which more will be said later. The AF register pair is also called the *program status word* (PSW) since F contains status information

As mentioned earlier, memory contains both data and instructions. Instructions tell the CPU what operations to perform and in what order to perform them. Data referenced by instructions are called *operands*. Instructions in memory may be one, two, or three bytes long. The first byte is always a code telling the CPU what kind of operation to perform. This *operation code*, or *opcode*, identifies a particular instruction from the set of instructions known to the CPU.

Instructions which make no reference to memory consist of only an opcode byte. They move data between registers, operate on the contents of registers, and test register values.

Some instructions refer to an *immediate* operand, an operand which is included in the instruction itself. If the operand occupies a byte in memory, then the instruction is two bytes long, an opcode followed by a one-byte operand If it occupies a word, the instruction is three bytes long, an opcode followed by the low-order byte of the operand, and then the high-order byte

Some instructions refer to operands in memory by specifying their addresses. There are two cases to consider. Some are three-byte instructions consisting of an opcode followed by a two-byte address in