SOFTWARE INTERPRETERS FOR MICROCOMPUTERS

Thomas C. McIntire

Software Interpreters for Microcomputers

THOMAS C. McINTIRE
NCR Corporation
Dayton, Ohio

John Wiley & Sons
New York • Santa Barbara • Chichester • Brisbane • Toronto

Library of Congress Cataloging in Publication Data:

McIntire, Thomas C 1942-
Software interpreters for microcomputers.

Bibliography: p.
Includes index.
1. Microcomputers—Programming. 2. Interpreters (Computer programs) I. Title.

QA76.6.M323 001.6'425 78-6608
ISBN 0-471-02678-6

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Software Interpreters for Microcomputers

Preface

Microprocessor technology has come of age. The computer-on-a-chip is being used in an ever-widening variety of applications. The data processing industry in particular is benefiting from the favorable price-performance ratios possible with microcomputers—computers that have a micro as the central processing unit. Business data processing generally favors the use of high-level programming languages, and interpreters provide a software method for interfacing various languages to microcomputers.

For the systems architect we present the arguments, both pro and con, for selecting the appropriate software architecture. For the software designer, the significance of different language attributes is described along with the considerations necessary for an interpreter-driven system.

In concept at least, programming a microcomputer is not very different from programming other computers. Thus, much of this book applies to interpreter designs for systems other than micros, although many of the specifics to be considered for microprocessors are identified. Also, the software engineer of the microcomputer interpretive system must eke out the maximum of performance, often in the minimum of space. For this reason the final chapters of this book are devoted to optimization techniques.

This book should also be useful to the applications programmer. How do interpreters work? Why are certain functions slower than others? Why do some routines consume large amounts of memory? In general, the answers to these questions can be found here, and more optimum programs can result.

Before modifications or improvements to an existing system can be attempted, the maintenance programmer must understand the overall philosophy of interpreters. In this case, study should include the rationale that influences the design of such systems. Therefore, many of the trade-offs that may be made in the design of an interpreter are discussed.

The basic principle of software interpreters is simple, but individual component routines may be complex and the overall size of the system may be large. For the benefit of the software engineering student, then, material is included on how to approach the design phase and on how to begin implementation and testing. Some suggestions are included on systems documentation.

The choice of the tutorial model (BASIC) was made with particular concern for that rapidly growing market: the home computer hobbyist. A number of BASIC interpreters are commercially available, or the hobbyist may wish to attempt his own design or modify an existing system. The model described here should provide the basis for many forms of experimentation.

Although the discussion is not complete in every area, almost every facet of interpreters, their use, and their design, is touched upon.

Thomas C. McIntire

Acknowledgments

The writing of this book was an arduous undertaking, and the thoughts, efforts, and expertise of many people contributed to its completion. I am unable to ascertain the origins of many of the concepts and methods I have described, but I wish to acknowledge the contributions of the professionals of the industry and of computer scientists in general.

I also thank those people whose personal assistance to me made this work possible.

For both stimuli and critique, I must thank my friend, associate, and boss, Keith Lohmuller. I express my gratitude also to Bill Bird, Assistant Vice-President, Marketing Software Programs, for inviting me into the fold of the NCR Corporation; association with the many professionals of this prestigious industry giant has provided a valuable research vehicle and has enriched my life.

It is often said that behind every man stands a woman—and so it is with me. Notwithstanding her continuous help, encouragement, and confidence, I can offer here only a simple "thank you" to my wife, Gloria for her many hours of typing.

As to the practice of programming, I must attribute eons of experience to Steve Clark. Our years together in the bit-bending business leaves me forever in his debt for the knowledge and help he has so freely shared. His technical critique of the manuscript merely adds to my continuing indebtedness.

No writer is truly self-sufficient, and my thanks also go to Ken Sessions as my editor and mentor.

To the readers doing the final tally: where you are enlightened and informed, it is due to the efforts of many; any ambiguity is entirely my own.

T. C. McIntire

Author's Note

The organization of this book attempts to enfold the many topical ramifications into a usable form. Throughout I have sought a reasonable balance between thoroughness and brevity. The intent has been to provide sufficient detail for the interested student without trying the patience of the professional.

The background material in Part 1 may appear very basic to veteran readers, but it is necessary to establish a mutual ground for communication. Interpreter usage rationale is examined, and microprocessors are explained.

The categorization of interpreter types in Part 2 is for the benefit of later discussion. Since no finite scheme of description is fully suitable for naming different types of interpreters, this part provides a base address for indirect referencing.

In Part 3 I enumerate all the various aspects the designer of an interpreter must consider. The critical user of an interpretive system may gain insight as to the scope of the designer's task. For those embarking on interpreter designs, I have included material on a methodology for problem definition. The template form of block diagramming is my own interpretation of a technique that is actually employed in some software development shops.

The model described in Part 4 was selected to show implementation techniques for solving certain interpreter design problems.

Part 5 discusses some of the many "tricks of the trade" in the programming profession. Since entire textbooks can be and have been written on these subjects, it was necessary to limit this part. All programming efforts must contend with speed and overhead tradeoffs. Interpreters are at least one step removed from the primary function of using a computer to do a problem task, and it is especially important to practice economies in their design. Although placed physically last in the book, techniques and methods for programming specific functions must temper many decisions early in the process of design.

In the same context, I offer this advice to the novice designer: Every effort expended during design and problem definition will be well repaid in the final analysis.

T.C. McIntire

Software Interpreters for Microcomputers

Contents

PART 1

Background

Interpret . . . to translate nonmachine language into machine language.

Interpreter . . . a software routine that, as processing progresses, translates a stored program expressed in pseudo-code into machine code and executes the intended operations.

A thorough discussion of computer interpreters must include the *why* function. Since computers are problem-solving machines and their programs constitute a part of the total system, we must examine the solution capability of interpreter programs. Interpreters are generally considered as part of the software. To appreciate the role of interpreters it is necessary to identify those processing problem attributes that are common to the use of all computers, regardless of specific job tasks.

In interpretative software systems, as in all technologies, there are advantages and disadvantages associated with various implementation techniques. We shall therefore examine several alternative methods of system design so that we may judge when interpreters should be used.

In advancing our theme that microcomputers tend to motivate the use of interpreters, we must appreciate the nature of these stimuli. The widespread usage of microprocessor-based systems attests to their acceptance and need. Representative micros are included in this part to show the causative factors that influence the popularity of interpreters.

Chapter 1

Why Interpreters?

The fact that interpretive software systems are so commonplace in the world of computers implies that there are decided advantages to their use. There is often no perfect single solution to a large group of problems. Since interpreters are not used by every system, we can infer that there are some disadvantages. In the selection of choices we must analyze both the positive and the negative aspects.

In addition to the good and bad points, alternative possibilities must be identified and examined in order to arrive at a sound conclusion. This chapter deals with the favorable aspects of interpreters, the unfavorable aspects, and some of the alternatives frequently used.

1.1 Advantages

The owner of a computer system once asked me what the impact would be if he decided to replace his system with that of a different manufacturer. Much to his chagrin, my answer implied bad news. This problem is not atypical, and it is useful to introduce one of the most favorable arguments for interpreters: case of migration.

In Figure 1.1, the location of the interpreter between the application programs and the computer implies an insulation function. The physical placement of an interpreter between the programs and the processor can, in fact, result in insulation. In the case of the system owner mentioned above, all of his application programs had been written in an *assembler language.* Assemblers are usually designed such that for a given source program statement, a single machine-language object code is generated. The output from the assembly process is machine code, and this restricts the use of that code to a processor that can properly execute it. The portent for users contemplating switching "engines" is the possibility of having to scrap and rewrite their entire program library, which may have evolved over a long period of time and at considerable expense in labor and dollars.

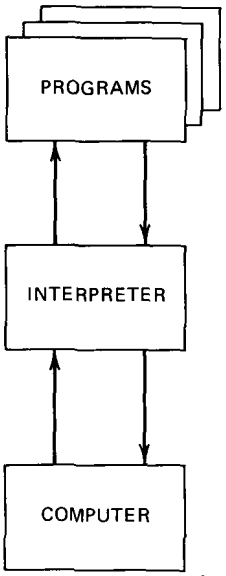

Figure 1.1 An interpreter can function to insulate programs from hardware changes.

Technological advances by the computer industry have generated this dilemma with such recurring frequency that many methods have evolved to make the problems of migration less severe. These solutions involve compromise, however, and are therefore discussed further in the section dealing with alternatives.

System architectures based on the use of interpreters makes it possible to change the processor with virtually no impact on the users of the system. But to exploit fully this switching capability requires that all aspects of the original implementation be biased in favor of interpretive schemes. In this instance, the languages used for application programming must be oriented to produce an object code that is not sensitive to a particular computer's machine structure. This is achieved through the use of assemblers and compilers that produce as output a *pseudo-object code.*

The advantage of interpreters stems from the use of an artificial object code and strings of instructions written in machine language that translate their meaning during execution. It is necessary, however, that the interpreter program itself be written in the native machine code of the using computer. However, only a single native language program is required, permitting the use of all programs whose artificial object code agrees with the structure expected as input by the interpreter.

It is far less costly to develop a single interpreter than to rewrite entire libraries of programs. This has long been recognized by the manufacturers of data processing systems, who frequently employ interpretive methods to their own benefit.

The vendors of many systems on the market today offer various types of software products in conjunction with their hardware. The leaders in the computer industry offer complete data processing systems, consisting not only of the hardware but of system software and application program product sets as well. Even those companies specializing in smaller systems or in primarily the hardware products offer such software as language compilers and common utility programs.