

AD-A132 172

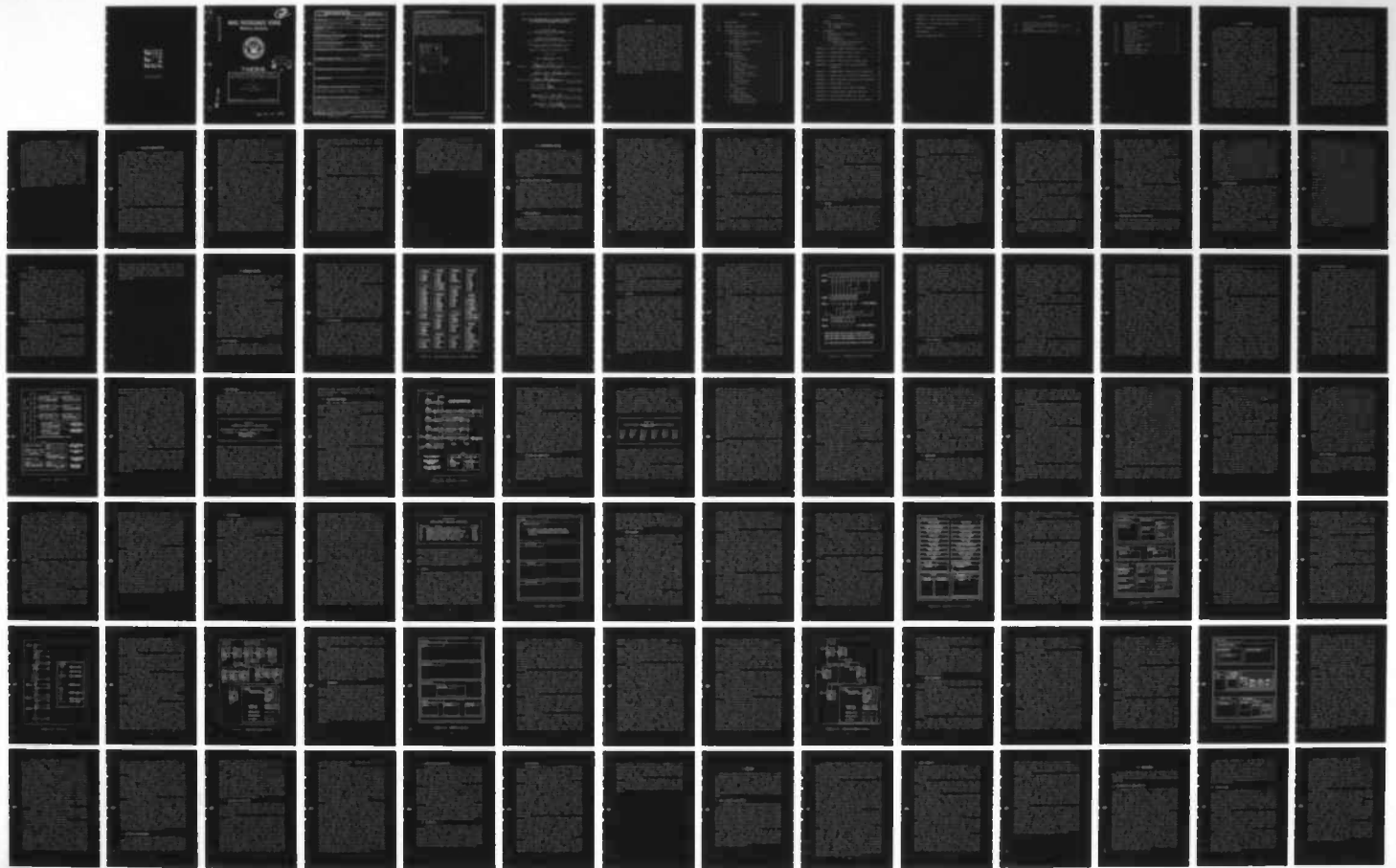
DESIGN AND IMPLEMENTATION OF A BASIC CROSS-COMPILER AND
VIRTUAL MEMORY MA. (U) NAVAL POSTGRADUATE SCHOOL
MONTEREY CA M R KINDL ET AL. JUN 83

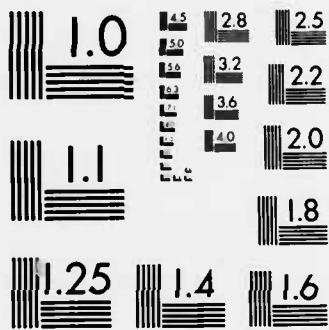
1/4

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

2

ADA 132172

NAVAL POSTGRADUATE SCHOOL

Monterey, California



DTIC
ELECTE
SEP 7 1983
S D

THESIS

DESIGN AND IMPLEMENTATION OF A BASIC
CROSS-COMPILER AND VIRTUAL MEMORY MANAGEMENT
SYSTEM FOR THE TI-59 PROGRAMMABLE CALCULATOR

by

Mark R. Kindl
and
James H. W. Inskeep, Jr.
June, 1983

Thesis Advisor: Bruce MacLennan

DTIC FILE COPY

Approved for public release; distribution unlimited

83 09 06 061

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A132	3. RECIPIENT'S CATALOG NUMBER 172
4. TITLE (and Subtitle) Design and Implementation of a BASIC Cross-Compiler and Virtual Memory Management System for the TI-59 Program- mable Calculator		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June, 1983
7. AUTHOR(s) Mark R. Kindl James H. W. Inskeep, Jr.		6. PERFORMING ORG. REPORT NUMBER
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California		8. CONTRACT OR GRANT NUMBER(s)
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		12. REPORT DATE June, 1983
		13. NUMBER OF PAGES 305
		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) BAX59, BASIC, Cross-Compiler, linker, TI-59, Segment, Calculator, Compiler, Pascal, Translator		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The instruction set of the TI-59 Programmable Calculator bears a close similarity to that of an assembler. Though most of the cal- culator instructions perform primitive data movement and/or sequence control, some can do the work of small high level language procedures. Regardless of this fact, to design and debug TI-59 programs of moderate size can be more difficult than doing the computations themselves. Programming in a higher order (Continued)		

ABSTRACT (Continued) Block # 20

language such as BASIC offers many advantages over calculator code. This report presents the design and implementation of a cross-compiler which translates correct BASIC programs into equivalent TI-59 programs. This software package includes a linker which maps calculator instructions to a set of magnetic cards. The cards are then used to implement a manually operated virtual memory system for the calculator. This expands program step capacity, and permits more complex programs to be written in BASIC language for translation into TI-59 instructions.

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	



Approved for public release; distribution unlimited.

Design and Implementation of a BASIC Cross-Compiler
and Virtual Memory Management System
for the TI-59 Programmable Calculator

by

Mark R. Kindl
Captain, United States Army
B.S., United States Military Academy, 1974

and

James H.W. Inskip, Jr.
Captain, United States Army
B.S., United States Military Academy, 1974

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1983

Authors:

Mark R. Kindl

James H.W. Inskip, Jr.

Approved by:

James H.W. Inskip, Jr.

Thesis Advisor

Q. E. Latta

Second Reader

David K. Hsiao

Chairman, Department of Computer Science

Keith T. Marshall

Dean of Information and Policy Sciences

ABSTRACT

The instruction set of the TI-59 Programmable Calculator bears a close similarity to that of an assembler. Though most of the calculator instructions perform primitive data movement and/or sequence control, some can do the work of small high level language procedures. Regardless of this fact, to design and debug TI-59 programs of moderate size can be more difficult than doing the computations themselves. Programming in a higher order language such as BASIC offers many advantages over calculator code. This report presents the design and implementation of a cross-compiler which translates correct BASIC programs into equivalent TI-59 programs. This software package includes a linker which maps calculator instructions to a set of magnetic cards. The cards are then used to implement a manually operated virtual memory system for the calculator. This expands program step capacity, and permits more complex programs to be written in BASIC language for translation into TI-59 instructions.

TABLE OF CONTENTS

I.	INTRODUCTION	10
II.	SOFTWARE REQUIREMENTS	13
III.	PRELIMINARY DESIGN	17
	A. PRELIMINARY DESIGN DECISIONS	17
	1. Cross-Compiler	17
	2. Linker	20
	B. PRELIMINARY DESIGN ORGANIZATION	23
	1. Cross-Compiler	24
	2. Linker	26
	3. Direct Interface	26
IV.	DETAILED DESIGN	28
	A. CROSS-COMPILER	28
	1. Initialization	29
	2. Scanner	32
	3. Error Handling	35
	4. Symbol Table Management	39
	5. Expressions	42
	6. Unstructured Jumps	43
	7. Looping and Branching	45
	8. Functions	49
	9. Code Resolution	53
	10. Input/Output	56
	E. LINKER	58
	1. Preprocessor	60
	2. Segmentor	71
	3. Post Processor	77
	C. INTERFACE ENGINEERING	83
	1. Addressing TI-59 SBR	84
	2. Structured Subroutines	86

3.	Recursion	36
4.	Input/Output	87
V.	TESTING	89
A.	TEST PROGRAM DESCRIPTION	89
B.	TEST COMMENTS	91
VI.	CONCLUSION	93
A.	EVALUATION OF TEST RESULTS	93
B.	CONCLUSIONS	94
C.	RECOMMENDATIONS	96
1.	Hardware Related Suggestions	96
2.	Array Implementation	97
APPENDIX A:	WBASIC SUBSET RECOGNIZED BY BAX59	99
APPENDIX B:	CONDENSED BAX59 USER'S GUIDE	105
APPENDIX C:	CROSS-COMPILER SOURCE CODE	110
APPENDIX D:	RWTBLF FILE--ORDERED RESERVED WORDS	183
APPENDIX E:	LABELF FILE--TI-59 LABELS/RESERVED REGISTERS	185
APPENDIX F:	BIFNLF/EIFNLF FILES--BUILT-IN FUNCTIONS	186
APPENDIX G:	CTEXTF FILE--TI-59 KEYCODE TRANSLATIONS	187
APPENDIX H:	MSGF FILE--CROSS-COMPILER OUTPUT MESSAGES	189
APPENDIX I:	LINKER SOURCE CODE	191
APPENDIX J:	MESSAGEFILE FILE--LINKER MESSAGES	242
APPENDIX K:	ARTILLERY TEST PROGRAM SOURCE CODE	248
APPENDIX L:	TEST PROGRAM LISTING FILE (LISTF)	251
APPENDIX M:	TEST PROGRAM NAME MAPPING FILE (NAMEF)	265

APPENDIX N: TEST PROGRAM DATA/READ MAPPING FILE (READF)	267
APPENDIX C: TEST PROGRAM LINK INTERFACE FILE (SCRATCH)	269
APPENDIX P: TEST PROGRAM LINKER OUTPUT	282
LIST OF REFERENCES	301
BIBLIOGRAPHY	302
INITIAL DISTRIBUTION LIST	303

LIST OF TABLES

I. Production Rules for Expressions 42

II. II-59 Keycode Sequences Equivalent to Boolean
Operators 46

III. EAX59 OPTION Statement Parameters 58

LIST OF FIGURES

4.1	Cross-Compiler Contour (PROGRAM BAX59)	30
4.2	Reserved Word Table Arrays	34
4.3	Symbol Table	40
4.4	Code Data Structure	44
4.5	Linker Contour	59
4.6	System Utilities Contour	63
4.7	ELI_SEG_TBL Contour	65
4.8	II-59 Code	68
4.9	Sequential Segment Table	70
4.10	COALESCE Contour	72
4.11	Coalesced Segment Table	76
4.12	INSTRUCTIONS Contour	80

I. INTRODUCTION

Hand-held programmable calculators provide an extremely portable means of computation. Designed primarily for small-scale numerical computation, these devices are limited by their small memory capacity, slow processing speed, and inability to perform symbol manipulation. These constraints, however, cannot hide the programmable calculator's usefulness and power as a computational tool. The instruction sets of these machines resemble those of assemblers. Most instruction words are primitive and perform simple data movements or sequence control. Yet, some specialized instructions do the work of small high order language procedures. Typical examples include polar to rectangular coordinate conversion, trigonometric functions, and logarithmic functions. In most assembly languages these operations must be constructed from primitive instructions. Even with these added features, the designing and debugging of calculator programs for non-trivial problems often requires an expenditure of effort which conceals the usefulness of the final product as well as the calculator.

There are a number of reasons for this difficulty. The lack of a sophisticated display mechanism such as a CRT prevents the user from viewing more than a single data item or instruction at any given time. Printing devices can lend assistance, yet most provide little more than a means of dumping memory contents. Furthermore, program debugging can be very difficult if the only error diagnostic is a single flashing display or an incorrect result.

Maximum memory storage capacity constrains even main-frame computers. One solution has been the implementation of virtual memory, whereby a relatively limitless on-call

secondary store is used to back up the primary storage. Programmable calculators usually have secondary storage in the form of magnetic cards. Normally, these cards are used as archival rather than on-call storage. The instruction sets of calculators are generally a cross between assembler instructions and a math function library. Compared to an assembler, the calculator instruction set is small and includes only the most basic sequence controls. Though it is possible to build more sophisticated control constructs from the primitives, such endeavors are often constrained by storage capacity. As a result, if complex programs are to fit into memory, it becomes necessary to learn or invent machine dependent "tricks."

The programmer's inability to use meaningful names for variables can create more difficulties during calculator programming. Numerical register indices (a form of absolute addressing) must be used to reference variables. One of the major advantages of assemblers is that they provide for variable naming. While composing his code, the calculator programmer must either remember the register indices of his variables or continuously refer to his own written symbol table while composing code. Both methods become more error prone as the number of variables in use grows. Programs of any substantial computing power usually require large numbers of variables.

The problems associated with calculator programming are many of the same problems which plagued experienced programmers of large scale computers in the past. How can a beginner be expected to design good, sophisticated programs for a pocket calculator if it can be so difficult for someone with experience? One concludes that the majority of users write small, relatively straight forward programs and never fully utilize the power of such calculators.

The Texas Instruments TI-59 Programmable Calculator is one of the more popular models. Its value as a powerful engineering tool is indicated by its use at the U.S. Naval Postgraduate School and in the U.S. Army Field Artillery. Yet, sophisticated programming of the TI-59 suffers from the very weaknesses mentioned earlier. Why, then, should there be such interest in this device? Perhaps, the best answer to this question is provided by Hamming [Ref. 1]. He feels that such a primitive programming machine offers the user valuable experience because it is easy to operate and allows the beginner direct access to very basic computing hardware. But, he warns that attempting mastery of the TI-59 language is a waste of time. One who must do sophisticated or extensive programming for this calculator should, instead, use a cross-compiler to automate and reduce his effort. This report presents the design and implementation of one such cross-compiler for the TI-59.

II. SOFTWARE REQUIREMENTS

A systematic approach to software development begins with the defining of general requirements. In this case, the basic design goal is the production of an effective software tool which will simplify program development and increase memory capacity for the TI-59 Programmable Calculator. Achievement of this goal should result in several enhancements to the utility and capability of the calculator. There will be an increase in its ability to execute larger and more sophisticated software. Most computations which can be programmed in BASIC and some existing BASIC software (which may require minor modifications to be explained later) will become as portable as the TI-59.

Important requirements for a user-oriented language translation system should include that it be easy to use and easy to learn. The BASIC programming language is an obvious choice for the source language; it is popular, simple, and easily learned. More importantly, many BASIC constructs and key words are similar to those of TI-59. This similarity and the fact that both languages are line-oriented and sequential in nature greatly facilitates translation between them.

Many versions of the BASIC language currently exist. Because of its availability at the Naval Postgraduate School and its having many structured control flow constructs, Waterloo BASIC Version 2.0 (WBASIC) [Ref. 2], was chosen as the specific source language to be implemented by this compiler. The power of the TI-59 compared to the WBASIC language places restrictions upon the set of WBASIC commands which can be translated. The specific WBASIC subset implemented is deferred to the discussion of design issues in the

next chapter. While WBASIC is easy to learn, it should be apparent that subsetting the language will introduce exceptions and restrictions which will tend to complicate learning for the novice and confuse the veteran. It is desirable to maintain as few exceptions as possible, and to require that restrictions be clean and obvious. A construct should be implemented as completely as possible (within obvious limitations, such as the file handling or alphanumeric features) or not at all.

Provision for error detection and debugging is another important requirement of a language system. The intended use of this initial system will be as a supplement to an existing WBASIC interpreter or compiler. As such, the cross-compiler will assume error-free source program input. The only requirement for error detection will be for the compiler to recognize words/constructs which are not implemented, but which are ordinarily legal WBASIC commands. Debugging of TI-59 programs is so much more difficult than debugging of higher level language programs that it is reasonable to assume that a user would prefer to debug his WBASIC program using the WBASIC interpreter/compiler available. Once the program is logically correct, it may be cross-compiled to TI-59 code, at which time it will be checked for subset and calculator capacity errors.

The TI-59's designers have provided it with capabilities which can be roughly equated to the power of higher level language routines. Interchangeable Solid State Software (trademark of Texas Instruments, Inc.) modules allow on-line access to utility program libraries. Program steps required to call them and the exclusive reservation of particular registers are usually the only storage costs paid for use of these library programs. It is required that the power of these modules be harnessed by the translator under design. Additionally, other sophisticated features of the calculator

should be exploited whenever possible in order to maximize and enhance advantages gained by high level language programming.

The linker will statically link the steps of TI-59 programs so that it will not be necessary for a complete program to reside in calculator step storage during execution. Since the swapping in and out of memory modules in the form of magnetic cards can become quite complicated for a running process, it will be necessary to keep this manual system as transparent to the user as can be reasonably expected. The fact that the calculator has a single item display window and associated register will certainly restrict the degree of transparency which might otherwise be possible.

A system must perform static linking if it exceeds program storage available to it during execution. This program will be segmented into overlays according to the size of memory available to it and the portions of code it needs to execute at any given time. That a program be segmented so as to minimize overlay swapping, must be an additional explicit requirement of a linker whose overlay swapping will be supervised manually. It is assumed that we cannot significantly affect the execution speed of the calculator. Thus, the intent of the minimization requirement should be obvious--suppress program segmentation which will tend to involve human thrashing.

The system source code must be portable. In the simplest case, it is desired that the unmodified source code be capable of utilization on any machine which possesses the resources to store, compile, and execute it. Because of operating system variations in such conventions as file naming and handling, transfer and processing of the source code in unmodified form on another machine (with operating system different from that on which it is developed) will be

very unlikely. However, the need for changes should be kept to a minimum and should be localized.

Finally, as with most all major software projects, maintenance and readability are considered paramount. Even after its completion, the system will certainly contain undiscovered bugs, areas to improve, and room for additions. Furthermore, development of a large prototype software system requires a great deal of careful planning for addition and modification. Adherence to the programming principles which support both readability and maintenance is absolutely necessary. Additionally, detailed documentation of the source code will supplement and assist in achieving these goals.

III. PRELIMINARY DESIGN

After requirements definition the next step in the software engineering process is the formulation of a preliminary design. Sound software design principles are applied to previously stated requirements to construct the framework for a software solution. It is during this phase of design that many of the most critical decisions are made. These decisions may be based upon a variety of considerations, each of which directly impacts the software organization. These decisions and the resulting organization are explored in this chapter.

A. PRELIMINARY DESIGN DECISIONS

Before a design can be formalized the engineer must investigate all design options and tools available. The following section summarizes the major decisions which strongly influenced and constrained many aspects of the project. With most large software projects, time is an extremely critical resource. As such, its impact upon preliminary design considerations is usually quite strong. Keeping on schedule is generally cost effective. It will be readily apparent that time played a key role in this design also.

1. Cross-Compiler

The fundamental considerations which most influenced design of the BAX59 (Basic X-compiler for the TI-59) cross-compiler were the method of parsing it would use, and the language(s) in which it would be written. The availability of several versions of Pascal at the Naval Postgraduate

School and the working experience of the authors of this report with Pascal were, perhaps, the overriding reasons for its early selection as the design language. In addition, the extensibility, strong typing, and block structure of Pascal support modularity, readability, and maintainability. It was at this point that the parsing technique became an issue. The decisions were reduced to a selection between two alternate approaches. Berkeley Pascal was available on a Digital Equipment Corporation VAX-11/780 with Bell Laboratories' Unix operating system. The Unix system included software tools LEX and YACC which are capable of automatically generating a lexical scanner and LALR(1) parser from an input grammar. YACC allows the user to specify code generating actions which will be executed as the productions of the grammar are processed. The alternative system was entirely International Business Machines Corporation. IBM Pascal VS was available on an IBM-3033AP with VM/370 operating system. This system does not have the tools for automatic generation of a compiler front end. Instead, a scanner, recursive descent parser, and code generator would have to be developed from scratch.

While the prospect of automating the development of BAX59 seemed more time efficient and less trouble, it turns out that subtle problems involved are many. A compiler constructed using LEX and YACC is generated in the C language, a kind of structured assembly language. While it is possible to link object code compiled from Berkeley Pascal to object code compiled from the C language, the mixing of source code tends to destroy the portability and maintainability required of the system. Modifications or improvements to the finished system could only be made if the programmer were familiar with Pascal, C, and their interface. Likewise, a machine would be required to have both C and Pascal compilers in order to process the source

code for use. Thus, a recursive descent compiler in pure Pascal VS was the alternative selected. It quickly became apparent that a recursive descent compiler would be far easier to develop in pure Pascal. Using a block structured language which supports recursion, explicit use of parse tables and stacks is unnecessary. The activation record stack resulting from the recursive procedure calls implicitly holds the same information stored in a parsing stack. The advantage of using Pascal VS is the powerful debugging tools which this language system provides. While BAX59 was written in Pascal VS, to the greatest extent possible only those constructs and features which are standard Pascal [Ref. 3] were used.

Another major consideration involved the identification of the particular WBASIC language subset which could be translated to TI-59. Both feasibility and time constrained this selection. Commands and functions which primarily perform character string and file manipulation were quickly eliminated. The TI-59 is weak in alphanumerics and its storage capacity is too small to consider any concept of file handling.

The WBASIC language is rich with matrix and array functions and constructs. The overhead and difficulty of implementing these operations would outweigh any programming benefits they could provide. As a result, functions and constructs involving all composite data types were ruled out. With only slight overhead, it is possible to implement limited size, single-dimensional arrays. However, time restrictions required that this concept remain a suggested improvement.

In order to simplify the translation of a WEASIC source program, it was decided to allow the BAX59 scanner to recognize all WBASIC keywords as reserved words. This provided a clear distinction between real errors and

occurrences of legal keywords which had passed through the WBASIC interpreter but which had not been implemented in the subset. Otherwise, legitimate WBASIC keywords, not implemented in BAX59, would be treated and translated as identifiers. This obvious inconsistency might be very difficult for the user to detect as an error. It should be noted that WBASIC function names are not handled in this way. The reason is that the user can extend the BAX59 built-in function library. Further discussion of this idea is deferred to Chapter IV.

Appendix A is a summary of the WBASIC keywords and functions which have been implemented in BAX59 version 1.0. There are three general categories of keywords recognized by this cross-compiler. Command reserved words are implemented WBASIC keywords which indicate the start of a particular WBASIC construct or statement. Supplemental reserved words are implemented WBASIC keywords which cannot be used to begin a construct or statement, but which can be used (optionally at times) within same to guide the interpreter. Unimplemented reserved words are all WBASIC keywords which have not been implemented in BAX59. The use of this last category of reserved words will result in a fatal subset error during translation.

2. Linker

In designing the linker three major problems arose. This first problem involved the fact that the linker is mainly a postprocessor of compiled data. As such, the linker is highly dependent on the compiler portion of the project. If this dependency were allowed, then most work on the linker would have to be deferred until the compiler was formalized and in the implementation phase of design. The second problem involved settling on a strategy to segment compiled code according to the software requirement to

minimize magnetic card reads. Two courses of action were discovered, each of which had advantages over the other. The third problem involved how prompting procedures were to be used to ensure proper execution of the segmented program. Procedures were required to be user-friendly and easily understood.

In the first problem it was decided to make the coupling between the linker and the compiler as loose as possible thereby reducing the dependency. This was achieved by defining a specific "third party" interface between the compiler and linker. This interface was defined to be a text file containing the four coded pieces of information required by the linker to accomplish its task.

This arrangement had several advantages and disadvantages. One advantage was that it allowed for the parallel development of the linker and the compiler. Since the interface was well defined, no other information needed to pass between the linker and the cross-compiler. By using this system, interfacing considerations such as naming conventions were nil since each process was totally independent. Another advantage concerned future implementations. It was envisioned that future versions of the system would be implemented on microcomputers. By having the project divided in half both logically and physically it would be easily adaptable to the more constricting memory requirements of the microcomputer environment. These two advantages alone outweighed the only major disadvantage of the decision. The disadvantage is that the linker needed to be able to regenerate the compiled linked TI-59 code structure which was originally produced by the compiler. As it turned out, this penalty was small when compared to the overall size of the finished linker.

The second problem was a very difficult problem to solve. Due to the limited size of the calculator memory and the cumbersome nature of the magnetic card backing storage system, the software requirements dictate the minimization of magnetic card reads. This requirement mandated the following decision: a code segment-break cannot occur within a backward loop. It would be preposterous to read a magnetic card every time a program encountered a segment break within a thousand iteration backward loop. This led to the following hierarchy of segmentation rules. First priority, segmentation may not occur within a backward loop. Second priority, maintain invoked subroutines with the invoking code. Third priority, keep adjacent sequential code together. To implement these decisions it became necessary to examine the control flow structure of an input program.

The decision as to how to accomplish this control flow examination is the foundation of linker design. Basically, two options were determined. One dealt with the input program as a whole and the other dealt with it as a series of sequential parts.

In dealing with the input program as a whole, the design algorithm would check to see if the program met the memory limitations. If it did not then the algorithm would examine the control structure and determine where to make an optimal break, that is, a break in a sequential portion of the program. It would then check each new segment to ensure that they complied with the memory restriction. The algorithm would continue until all segments met the memory requirements.

In the other method the program was decomposed into a series of sequential segments (a sequential segment is defined as a segment which does not contain a backward loop reference to any instruction other than possibly to the

first instruction of the segment). The algorithm first determined the sequential segments. Next the algorithm combined adjacent segments until the memory limit was encountered. At this point a segmentation occurred. The memory limits were reset and the combining process continued until another limit was encountered or the whole sequentially segmented input program was processed.

The second method was selected for two reasons. The first reason is that the first method eventually required the evaluation of code on a small segment level much like the second to determine a suitable segmentation point. Rather than do this and more, it was decided to just evaluate the small segments and build up rather than down. The second reason was that the second method lent itself to a recursive solution during the recombination process. The recursive solution greatly reduced the length and complexity of the segmentation code.

The third problem involved deciding upon a method for accomplishing the prompting of the user. One method dealt with assigning coded prompt numbers of short length to be built into the code. The other method involved building larger self-explanatory prompts into the code. The second choice was selected. This was done to reduce the number of instruction references that a user might have to make during the execution of the generated calculator program. This was in keeping with the requirement to make the system user-friendly.

B. PRELIMINARY DESIGN ORGANIZATION

Thus far the system design space has been narrowed to the design language, source language subset, and the general techniques for compilation and linking. Organization of the software into functional categories may now begin. This

next phase of development is characterized by a more specialized, yet still preliminary consideration of system components. It should be apparent by now that a natural division into two major functional components, cross-compiler and linker, has been assumed since conception of the system. For a two-man design team, this partitioning appeared to have the greatest potential for success. It allowed the simultaneous development of two independent system components of low coupling [Ref. 4: p. 85] and high cohesion [Ref. 4: p. 106]. The result of this separation was a minimization of programmer interaction, maximization of work time efficiency, and simplification of interfacing. The remainder of this section outlines the preliminary design and organization of the cross-compiler, linker, and the direct interface between them.

1. Cross-Compiler

The common form of all versions of BASIC language can be characterized as imperative, line-oriented, and sequential. The design of BAX59 is based upon this fact. Each WBASIC line is parsed, beginning to end, by recursive descent. Equivalent TI-59 code is generated for each line concurrently with the parsing operation. This means that BAX59 will successfully translate a sequence of syntactically correct yet meaningless WBASIC statements into equivalent TI-59 code. However, the TI-59 code will be as meaningless as the source code. For this reason, it is recommended that the user successfully execute his WBASIC source program using a WBASIC interpreter (or compiler/loader) prior to translation with BAX59.

A line-oriented view of the source code provides several advantages to the design. First, there is a direct sequential correlation between the original source program and the translated TI-59 code. As will be seen later, this

allows easy management of the generated code and its associated data. Second, the parse driver routine can be a fairly simple loop, since lines are parsed to end of line one at a time until the end of the source file. Third, viewing the source as a sequence of independent pieces greatly facilitates an iterative enhancement approach [Ref. 5] to the progressive development of EAX59. This approach virtually guarantees a working prototype throughout the coding phase, and supports both reliability and maintainability. Modifications can be quickly tested within the context of the entire compiler system to date. Upon completion, the programmer tends to have greater confidence in his product, because a great deal of testing has already been conducted.

EAX59 was temporally organized into three major functional sections: initialization, translation, and resolution. The primary operations performed by initialization involve setting up data structures and initializing variable values. There are three major data structures manipulated by the translation section: the reserved word table, the symbol table, and the code data structure. Conceptual subdivisions of this section, namely the scanner, the parser, and the code generator, manage each database respectively. While the scanner is a separate routine by itself, the parser and code generator are not as separately defined. These functions are actually performed concurrently by a set of mutually recursive procedures under the direction of the main driver. This driver calls the correct subprogram into execution as its corresponding WBASIC construct is recognized. Once translation has been completed, the resolution section processes the generated code into a form suitable for final output. This includes label insertion, peephole optimization, and absolute address resolution.

2. Linker

The linker was organized into three phases. The design of these phases were the direct results of the preceding design decisions made in the preliminary phase.

The first phase is the direct result of the loose coupling between the linker and compiler and the decision to combine small sequential segments to form memory-size constrained segments. It is the preprocessor phase. This phase processes the interface input file and reconstructs the compiled linked code structure. In addition the preprocessor determines the sequential segments of code and constructs an internal data table called the segment table which is used by the second phase, the segmentation phase. The segmentation phase utilizes the recursive algorithm to recombine sequential code. The postprocessor phase is the result of output design decisions. This phase inserts the prompting code and develops the segmented code lists to be output to the user. It then produces the code in a text format together with specific instructions as to its use.

3. Direct Interface

The design organization is built around the loosely coupled compiler and linker. This coupling is made possible through the rigid definition of the interface text file. The organization of this text file is critical to the design and will be described in more detail.

The text file is the only direct transmittal of data between compiler and linker. Four pieces of data are transmitted. They are the following: a number signifying the next register available for use; generated code list in a numeric formatted form; text containing DATA/READ information; and text containing the mapping of TI-59 registers to BASIC variables. Each piece of information is preceded by a

\$XXX in the first column where XXX is a number. This simple format enables the linker to easily locate the correct information and process it accordingly. Since this is the only explicit interface, the compiler and linker are not as dependent on each other as they would have been in a closely coupled system.

IV. DETAILED DESIGN

The source code contained in Appendices C and D provides high resolution understanding of the system. However, in order to provide rationale behind design issues for a language system comprised of almost 10,000 lines of code and comment, discussions of some detail are in order.

It is not our intention to explain everything. What we wish to do in this chapter is to introduce design details and strategies of the more important concepts and components in the system. This will serve to illustrate software engineering and how it is used in this project.

Upon the completion of preliminary design, the detailed design is begun. It is during this phase that the actual details of full implementation are defined and laid out prior to coding. Categorized under cross-compiler and linker, the general format for the sub-sections of this chapter will include an informal solution strategy for a specific design problem, followed by a discussion of the major data objects and procedures which manipulate those objects. Where appropriate, inter-procedure interfacing criteria are outlined. In several cases, significant problems, their solutions, and possible system improvements are discussed. The last section presents implicit interface design which impacts greatly on the system.

A. CROSS-COMPILER

The fundamental design of BAK59 is a finite state machine driven by a main loop. Once values and data structures have been initialized, program control enters the main loop which scans, parses, and generates TI-59 code for one

WBASIC source line. At the end of line, the loop checks for end of source file. As long as the end of file is not detected, the main loop repeats its processing of each succeeding line of source code. When end of file is found, control exits the main loop and post-processing begins on the generated TI-59 code. This includes insertion of suspended code data, optimization, address resolution, and final output of code and associated data.

The entire cross-compilation process is broken down into 15 functional areas. These are outlined by the contour diagram in Figure 4.1 Note that solid lines indicate actual procedures (P/) or functions (F/), while dotted lines only indicate logical association. Although these areas often depend upon one another, the particular services performed by each differ enough to allow independent analysis. Were it not for limitations imposed by the Pascal language, many more procedures and functions would have been tightly packaged in order to hide implementation details between functional areas. What follows is a survey of the major functional areas of EAX59 and the design details for each.

1. Initialization

In the interest of execution time efficiency all run-time actions which required completion prior to the start of parsing are incorporated into procedure INITIALIZE. As a result, some variables and databases which would otherwise have been local to their respective procedures, required globalization. One particular example is the reserved word table. This data structure is built of data supplied from outside the program in file RWTBLF. Since the reserved words contained in this file need only be loaded once (at the start of execution), there was no good reason to place them into procedure SCAN, which is called often by parse routines. While it would have been possible to use a

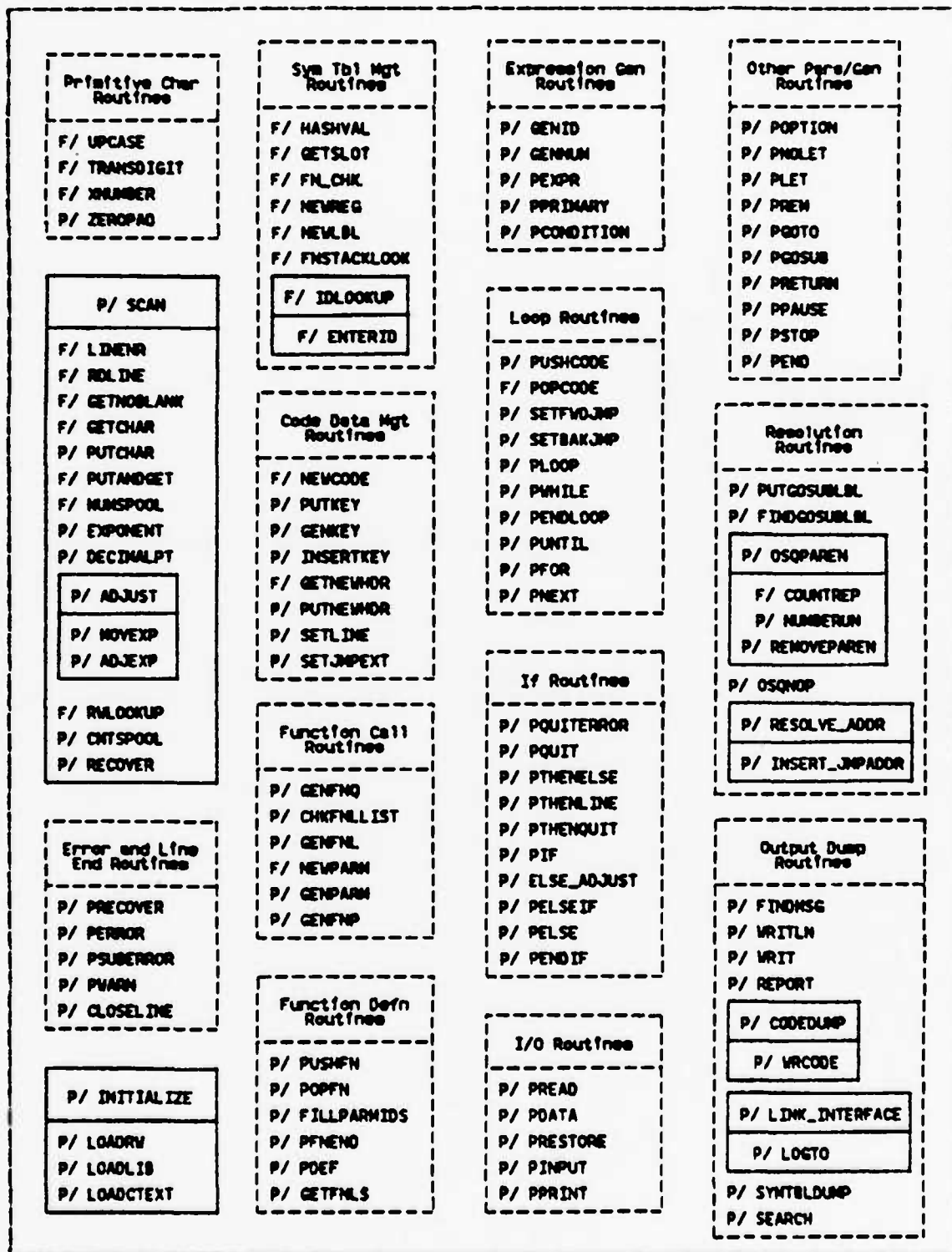


Figure 4.1 Cross-Compiler Contour (PROGRAM BAX59).

boolean switch to detect the initial call to SCAN and subsequently generate the reserved word table at this first call, this would have increased the coupling between modules. Loading the reserved word table from outside the program offers several advantages. First, changes to the reserved word table can be made easily (this process will be discussed when the scanner is considered). Second, the user can check one file (RWTBLF) in order to see the reserved words recognized by FAX59. Third, the loading routine does not need to know the words themselves, only the name and format of the file in which they reside.

Built-in functions are also loaded by procedure INITIALIZE. This operation actually requires two ordered steps. First, the symbol table must be created. Second, function recognition and generation data is read from outside files (BIFNCF and BIFNLF) and the symbol table management routines are used to put this data in the table. Loading the built-in functions by using the same routines which the parser will use to manage the symbol table, ensures table consistency and promotes readability. This approach has been taken as often as possible in designing the cross-compiler.

The complexity of the above initialization processes as well as the the TI-59 keycode text (CTEXTF) loading process required that these operations be abstracted into individual subprograms. However, procedure INITIALIZE performs many other pre-compilation activities which could be performed sequentially. Probably the most important of these activities is the simple initialization of variable values. The importance of this task is elevated by a serious hole in the Pascal VS implementation. The Pascal VS compiler will not detect the failure to initialize a variable value prior to its use. What is worse, random values which exist in pointer references or other variable storage

areas will be used as is, whether they were put there by the user or not. As a result, failure to initialize values was a major source of error during development of BAX59. These types of errors tended to be extremely difficult to debug, since they often surfaced late and usually in modules long thought to be robust.

In summary, procedure INITIALIZE loads all information which will be needed by the translation and resolution stages, constructs conveniences data such as character sets, assigns starting values to all variables and pointers.

2. Scanner

At the lowest level of abstraction within the translation component of BAX59 is the scanner, procedure SCAN. This single, self-contained subprogram is basically designed on three important concepts. First, the scanner is itself a finite state machine. Second, with the exception of procedure INITIALIZE and some system constants, its implementation is transparent to the rest of the cross-compiler. Third, the database which it uses for token recognition is simple, time efficient, and general.

The state machine logic of procedure SCAN provides knowledge of token streams in free format and nothing more. Its primary job is to read the source file character by character in order to isolate and recognize single tokens. However, procedure SCAN is designed to do much more. First, it reads and converts line numbers. It also fills as necessary the line buffer and accumulator, the data structures which store the line and token currently being scanned respectively. The scanner also detects the end of a source file which has no explicit WBASIC "END" statement. This allows a more graceful conclusion to what might otherwise be an abrupt exit.

Two other functions performed by the scanner will illustrate its transparency to the rest of the cross-compiler. These are recognition of the end of a line and of continuation lines. Procedure SCAN reads and loads the line buffer (LINEBUF) with a new line of source text each time the end of line character ("@") or continuation line character ("&") is found. The only difference is that the end of line token number must be passed up to the parsing routines so that the main loop will know when it has parsed one entire line. On the other hand, the continuation character can remain invisible to the parser, which views only whole source lines.

As mentioned before, the database used by procedure SCAN is the reserved word table. Although referred to as a table, the internal representation of this database is actually three coordinated arrays constructed from the RWTBLF file by procedure INITIALIZE. These arrays are used to compare the characters of a token in the accumulator to the characters of reserved words. The simplicity and efficiency of this comparison is illustrated in Figure 4.2, which depicts a condensed schematic of the arrays. Note that the characters in the RWCHAR array are arranged in order of increasing word length. A reserved word look up is based upon the length of the word in the accumulator. Comparison begins at the first character of the first word in the RWCHAR array which matches the length of the token in the accumulator. Comparison ends when either all characters in the accumulator match a string in the RWCHAR array, or when the characters of all words of a given length have been compared to the accumulator without success.

The RWORD array references the start index of each word in the RWCHAR array, while the RWLENG array references the start index for the first word in the RWORD array for that length index. The indexes of the RWORD array are used

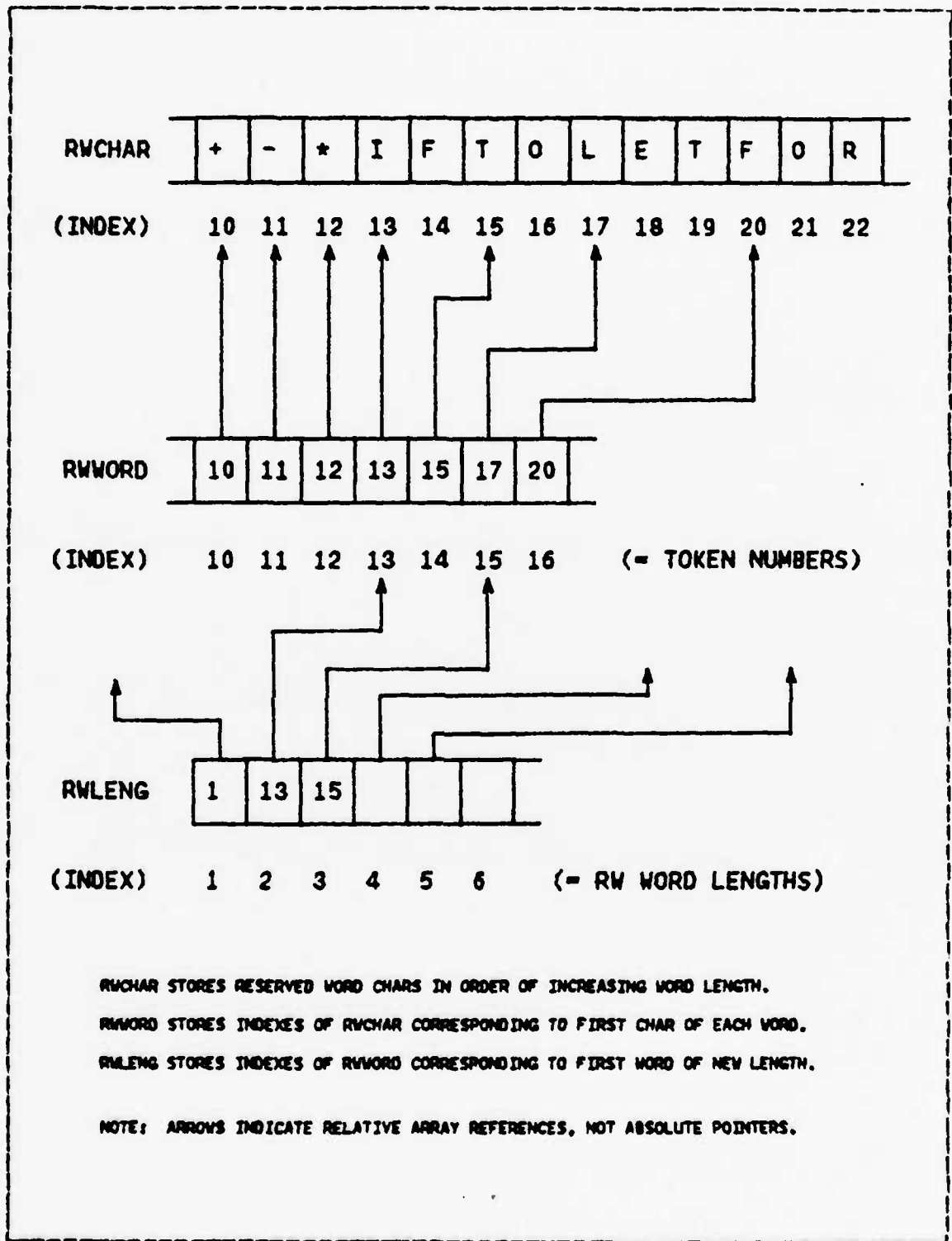


Figure 4.2 Reserved Word Table Arrays.

as the token numbers returned by the scanner after a successful look up. Should a look up operation fail to recognize the accumulator token, then the token is assumed to be a variable identifier. Other token types, such as numerics, are recognized prior to table look up. This mechanism and the fact that the scanner is independent from the parser require that WBASIC keywords be reserved. If keywords were overloaded as variable identifiers, the parser would have to communicate its token type expectation to the scanner. Token overloading would greatly hamper readability.

One scanner related problem which required a relatively complex solution concerned the conversion of WBASIC to TI-59 numeric values. The calculator display window restricts the number of significant figures which can be entered from the keyboard. For numbers without exponents, a maximum of ten digits (with decimal point) can be entered. Numbers with exponents are allowed a maximum of eight digits in the mantissa and two in the exponent. Because of this restriction, a rather complicated procedure was designed to convert WBASIC numeric values to TI-59 compatible values without losing equivalence. Procedure ADJUST performs decimal point shifting and exponent modification on WBASIC numerics which contain too many significant figures for the TI-59. The operation can, of course, reduce significance by truncation of excess digits. Except for this loss of digits, equivalence is maintained.

3. Error Handling

At this point it is appropriate to discuss error recognition and recovery. As implied earlier, the error detection capability in BAX59 is relatively weak and incomplete as compared to full language compilers. The reason is that the system requirements specified error-free input

source files. The primary use for this system is as a supplement to an existing BASIC language interpreter or compiler. Debugging of TI-59 programs is a hard enough task without adding the complexity imposed by absence of BASIC language run-time diagnostics. Therefore, users of BAX59 are strongly advised to ensure that a WBASIC program is correct syntactically, semantically, and logically by running it in the WBASIC environment, before translation to TI-59 code. Error handling in BAX59 is restricted to detection of subset related exceptions, calculator capacity limits, and errors of opportunity.

The cross-compiler is designed for recognition of two major types of errors: fatal errors and warning messages. Fatal errors are further categorized as scanner or parser detected.

Warning messages are generally unrelated to WBASIC syntactic or semantic problems. They refer to potential difficulties with the TI-59 run-time environment, most commonly (but not always) calculator capacity. Such conditions as too many registers in use, too many labels in use, or excessively nested subroutine calls, will trigger warnings. Each message is explicit and cautions the user of a situation which is considered abnormal to the calculator. Since these errors are unrelated to the WBASIC source code, warning messages do not halt the parsing or code generation processes. However, TI-59 code generated from a WBASIC source file that produced warnings is not guaranteed to execute properly, if at all.

The warning message is similar to non-fatal errors in full language compilers. The reason for continuation of code generation is slightly different. A user of BAX59 will most likely need to modify and tailor his WBASIC program to fit calculator constraints and capacities. Warnings are a non-fatal means of providing near equivalent code data for

use in comparison of efficiencies, capacities, or consistency. Even though the code may not successfully execute on the calculator, it still represents a direct translation from WEASIC and is a fairly accurate indication of program size, register/label use, etc.

Unlike a warning, one fatal error will flag the main loop against further parsing and code generation. However, scanning for tokens continues until the end of the source file is reached. Thus, only a single fatal syntax error can ever be detected in one BAX59 execution, although the scanner will continue to detect any number of lexical errors. Fatal errors are also categorized as subset or non-subset related. Non-subset errors are those previously referred to as errors of opportunity. During the coding phase of development, simple syntax checks were often inserted into the logic of the parsing routines. These were usually one-line IF-THEN-ELSE constructs which cost very little but were highly protective. For example, the main loop calls procedure PGOTO whenever the GOTO command is recognized. Since error-free input is required, this procedure could have been written to assume that the next token must be a numeric line reference. Instead, it was a simple matter to check the next token's type and call a syntax error (PERROR) if it were not numeric. Note, however, that the logic of PGOTO will not call an error for a numeric token which contains a fractional part, clearly a syntax error. In fact, the cross-compiler is not likely to detect an error at all. Execution may result in a Pascal VS runtime error. The reason is that the numeric string will be converted to an integer value based on ordinal values of the characters. The decimal point will appear to BAX59 as any other character. However, its ordinal value will be added during conversion resulting in an inconsistent integer value for the line reference. The routine used to set jump

pointers will probably not be able to find the line number since it is already in error.

Although often incomplete, these error traps provided much assistance in tracking system bugs. The technique used was to translate simple source programs known to be correct. Errors, tripped at these check points by system bugs, usually indicated the likely trouble spots. The faulty routine had helped in parsing either the statement which caused the error or the statement which immediately preceded it. Since the token which tripped the error was also known, the exact routine and the specific bug were easily found.

Subset related errors were defined in the software requirements. The user must be told where and how he has misused the system. BAX59 incorporates all WBASIC keywords (Version 2.0) in its reserved word table. The main loop logic contains the information to distinguish between implemented keywords and unimplemented keywords. This technique allows the reserved word table and implemented subset to be easily expanded (or contracted). Such a technique strongly supports the requirement for maintainable source code.

There is more room for improvement in the area of error detection and handling than in any other aspect of the cross-compiler. The capability could certainly be extended to protect against all possible syntactic and semantic errors so that prior compilation or interpretation would be unnecessary. However, the benefits to be gained are questionable, since run-time and logical debugging of TI-59 programs is no easy task. A special file to hold error message text might help to reduce some of the awkwardness in portions of the code which issue these messages. Within this file the messages could be indexed by number, thereby allowing more verbose and possibly clearer explanations of errors. Generally speaking, the critical resource of time forced the design of error handling to be barely adequate.

4. Symbol Table Management

One of the more important duties of the parser is to manage the symbol table. The BAX59 symbol table is a variable bucket hash table similar to one described by Aho and Ullman [Ref. 6]. The data structure used is an array of pointers. The indices to this array of base pointers are hash values computed by taking the modulo 99 sum of the ordinal values of identifier characters. This operation is performed by procedure HASHVAL. Figure 4.3 depicts the structure of the table itself and its four types of identifier entries. Three of the four types of identifiers are functions. These will be discussed later in this chapter. The important structural feature to notice now is that each node has a SLOTP field regardless of variant tag. The SLOTP field is each entry's link in the variable length chain which forms a bucket. In order to insure that no uninitialized pointer references or variables occur, new nodes are created as needed by the separate function GETSLOT. The job of this function is to create the node and to insure that all of its fields have been initialized to default values. This same approach to data structure construction is used throughout BAX59 in order to protect against random initialization by the Pascal VS compiler.

The look up operation of procedure IDLOOKUP is simply to hash the characters of the identifier token in the accumulator to the correct base pointer bucket in the symbol table. The IDENT field of each slot node is compared to the accumulator token until either a match or a nil pointer is found. If a match is found, a pointer to that slot is returned. If no match is found, then a slot for the accumulator identifier token is automatically added to the symbol table in the bucket just searched. A pointer to this new slot is then returned. This is in accordance with the

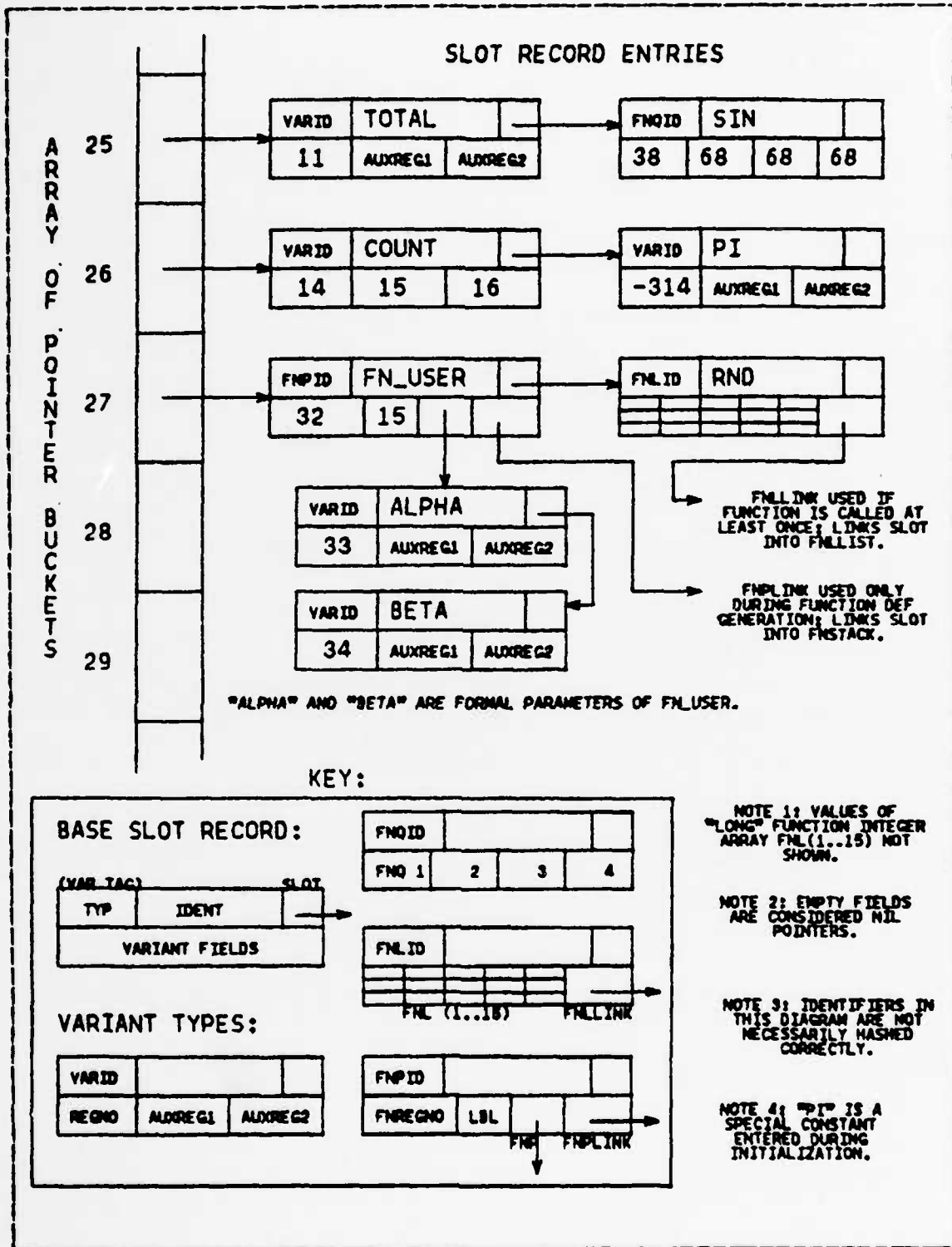


Figure 4.3 Symbol Table.

semantic rules of the BASIC language which allows implicit declaration of variable names by using them in statements.

The insertion of new identifiers into the symbol table is performed by procedure ENTERID. This procedure creates the new slot, fills all fields which are known, and links the slot into the symbol table. It is also during this process that identifiers are assigned TI-59 registers for code generation purposes. Function NEWREG handles the register pool, which is actually nothing more than an implicit stack of integers. An important feature regarding the assignment of registers to variable names is that the user has some control over these assignments from outside the program. Included in the LABELF file is a place to list register numbers which the user wants to reserve for his own use. Function NEWREG will not assign these numbers to WBASIC variable names. The significance and power of this control feature becomes more apparent during the discussion of functions. The user is cautioned against reserving the last assignable register number (system parameter in constant declaration block: REGBASE). Reserving this register will short circuit the logic which reports a TI-59 memory overflow warning message, the situation in which too many registers are in use.

As a final note, there are two forms of output which are closely associated with the symbol table. One is the WBASIC variable name to TI-59 register mapping which correlates variables to register assignments. The other is an optionally available symbol table image, which lists each table entry in bucket order with type and register assignment. Both outputs are discussed in the last section of this chapter.

5. Expressions

The most fundamental and most common construct seen by the parse routines is the arithmetic expression. The many similarities between BASIC language expressions and TI-59 expressions make them relatively easy to parse and generate. However, a few subtle differences cause abnormal situations requiring careful design. If there is one lesson to learn from this discussion, it is this: in compiler design, when in doubt revert to the grammar specification.

TABLE I
Production Rules for Expressions

```
<EXPRESSION> ::= <PRIMARY> {<BINARYOP> <PRIMARY>}
<PRIMARY> ::= {+|-} <PRIMARY> |
               <UNSIGNED NUMBER> |
               <IDENTIFIER> |
               ( <EXPRESSION> )
```

Table I lists the grammatical specification for a WBASIC expression. The two production rules in Table I are abstracted by the two BAX59 parse procedures PEXPR and PPRIMARY. They are designed to parse and generate code through mutual recursion. Careful examination of the case statements within these procedures will reveal the differences between WBASIC and TI-59 expressions. While both use infix notation for binary operators, unlike WBASIC, TI-59 unary operators and function applications are postfix. This minor twist in notation adds a little complexity to the logic of the expression parsing routines. However, once designed, the code for translation of expressions became the

fundamental base upon which assignment statements, conditional expressions, functions, and many other constructs could be built.

6. Unstructured Jumps

Some of the easiest constructs to understand and implement were the unstructured control statements GOTO and GOSUB. To realize their simplicity it is necessary at this point to introduce the code data structure which is constructed by the generation routines.

Illustrated by Figure 4.4, the code data structure is, perhaps, the most unique design concept of this cross-compiler. There are two types of nodes: WBASIC line number nodes and TI-59 keycode nodes. Since unstructured constructs in WBASIC are dependent upon source line numbers, there had to be a method of associating the TI-59 code with those same line numbers. Figure 4.4 shows how line nodes and code nodes are linked to duplicate this association. It is important to note that the TI-59 code chain is completely independent (and may be traversed as such) of the WBASIC line number chain. The line nodes merely provide a frame of reference for the TI-59 code.

Procedure SETLINE, called at the beginning of the main driving loop, is responsible for insuring that new line nodes are created and inserted into proper order. As each line is parsed, special holding pointers (FIRSTLP, LASTLP, BEGINCP, ENDCP, LPLEAD, LPTRAIL, LPCUR, CPCUR) keep track of all key locations in the structure. As Figure 4.4 indicates, it is possible to have line nodes created and linked prior to their encounter in the source code. This occurs whenever a forward jump (GOTO) is parsed. Since the line reference of a forward jump has not been parsed, its line node would not exist. However, the jump pointer (JMPP) must be anchored to a node. So the line node and an anchoring

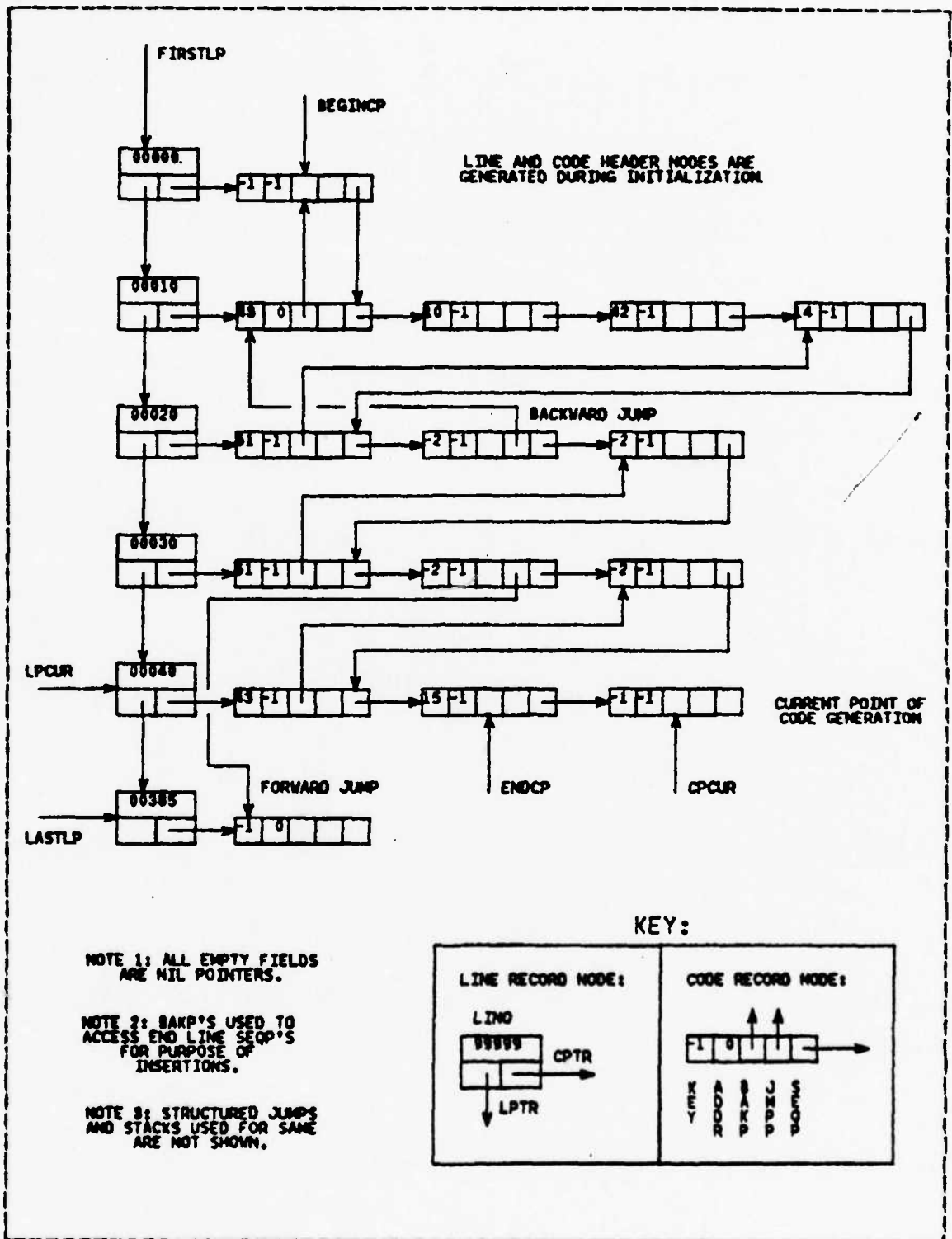


Figure 4.4 Code Data Structure.

code node are inserted in correct order ahead of the current line. Procedure SETJMPEXT sets forward as well as backward jump references. Of course, setting backward jumps is easier because the line number node has already been created and is in place. It should be noted that procedure SETLINE always checks its forward line number chain before creating a new line node. If a line node exists whose line number field (LINO) is equal to the next WBASIC line number, then it will be used instead.

The technique for handling GOSUB statements is similar to but slightly more involved than the GOTO. Since the GOSUB is actually an unstructured subroutine call, it was necessary to maintain consistency in code generation so that the linker could recognize the difference between subroutines and unconditional jumps. All TI-59 subroutines are prefaced with and called by a label name. Therefore, while initially the GOSUB can be treated as a GOTO, at some time later a label must be inserted at the head of the subroutine body, which is the node referenced by the jump pointer. This is done during the resolution phase of compilation by procedure FINDGOSUBLEBL. This fairly tricky insertion is one reason for the existence of the back pointer field (BAKP) in code nodes. This operation will be explored in the sub-section on resolution.

7. Looping and Branching

Users of BAX59 are strongly urged to practice structured programming when writing WBASIC code. Translation to and execution on the TI-59 are far more regular and predictable when the input source code is structured and readable. Much of the design of the entire system is based upon the assumption that the source program will be structured. You will understand why more thoroughly in the section describing linker design.

We begin discussion of structured looping and branching by introducing procedure PCONDITION. This procedure is fundamental to the parsing and code generation for simple boolean expressions (compound boolean expressions have not been implemented). While WBASIC has a fairly common set of boolean operators, the TI-59 does not. There was a need to construct efficient sequences of TI-59 code which are equivalent to the WBASIC boolean operators. These equivalences, shown in Table II, are implemented in

TABLE II
TI-59 Keycode Sequences Equivalent to Boolean Operators

A = B	A <> B	A >= B	A <= B	A > B	A < B
RCL A	RCL A	RCL A	RCL A	RCL A	RCL A
X->T	X->T	X->T	X->T	X->T	X->T
RCL E	RCL E	RCL B	RCL B	RCL B	RCL E
INV	X=T	X->T	INV	X>=T	X->T
X=T		INV	X>=T		X>=T
		X>=T			

procedure PCONDITION. While it would have been possible to implement compound boolean expressions (AND and OR), the lack of time and the fact that their logic could be duplicated using IF statements prevented this enhancement. It was, however, a very simple extension of logic to recognize and translate a negation (NOT).

In the implementation of a block structured language which allows nesting, the use of stacks is an important technique. And so it was with looping in BAX59. By nature loops involve backward jumps. As with unstructured jumps, there existed a need to anchor pointers on code nodes whose

source code had yet to be translated. In the case of loops, the inverse of this concept is also true. There was a need to create pointers from code nodes whose source code had yet to be translated. Since structured nesting of loops is checked by the WEASIC interpreter, it was possible to pre-create these nodes and push them onto a stack until their place of insertion is encountered. This is exactly how loops are translated using LOOPSTACK and ENDLOOPSTACK. When the LCOP statement is encountered, a NOP keycode node is created and pushed onto the ENDLOOPSTACK. In the case of the WHILE statement, a boolean expression is processed, a forward reference node is created, a jump pointer is set to the node (for the false branch to end of loop), and this node is pushed onto the LOOPSTACK. When the ENDLOOP or UNTIL is found, the stacks are popped and the NOP nodes are inserted. The nature of correct nesting guarantees that NOP code nodes popped from the stack will have jump pointers referencing or will be referenced by the appropriate code nodes.

Iterative loops are written by using the FOR-NEXT construct. The stack implementation is similar to that described above. The main difference is in the additional calculator resources required for such a loop. Unlike ordinary variable names, the FOR loop variable requires from two to three register assignments. The fields AUXREG1 and AUXREG2 in the VARIE tagged slot record are used for this purpose. AUXREG1 holds the TI-59 register number which will store the upper (lower) limit of the FOR index variable. The FOR index variable increment will always default to +1 unless the STEP option is used. If STEP is used, then AUXREG2 will hold the register number which stores the increment value. The user should understand that use of a FOR loop carries a fairly heavy overhead in terms of both register and program step use. A single simple FOR

statement in WBASIC translates to use of two registers and over 20 program steps. Most of this overhead is caused by the run-time checking of the FOR index variable value against its limit for each iteration through the loop.

Branching is another construct implemented with stacks. There are actually two forms of branching: the unstructured or line-oriented IF statement and the block structured IF statement. The unstructured IF is actually only partially implemented. WBASIC allows either a jump to a line number or execution of any single statement within each of the IF branches. Because the structured IF can be written to perform the same way, it was decided to restrict the unstructured IF to allow line jumps or the QUIT statement. The implementation of line-oriented jumps has already been discussed. An IF-THEN-QUIT or IF-THEN-ELSE-QUIT is handled by setting a jump pointer to the ENDLOOPSTACK. The effect is to force program control to exit the current loop. If control is not within a loop (i.e. ENDLOOPSTACK is nil), then an error condition is raised. FOR loops are not considered loops in this context.

The more powerful of the two IF statements is the block structured IF-ELSEIF-ELSE-ENDIF. This form disallows the use of keyword THEN, since it is implied. Once again, stacks are used to implement the structured IF. The logic and its correspondence to the manipulation of stacks is roughly similar to that of looping. Instead of directing jump pointers to the end of loops, they are directed to the next ELSEIF or ELSE. An unusual situation occurs, however, in the case of the IF statement. Stack manipulation for the IF-ENDIF is slightly different from that for the IF-ELSE-ENDIF or the IF-ELSEIF-ELSE-ENDIF. To understand the problem, assume the viewpoint of a parser which has just evaluated the condition of a structured IF. At this point you do not know whether or not an ELSE/ELSEIF or an

immediate ENDIF will follow the true branch. To which stack will the jump pointer of the false branch be set? In order to cover both possibilities, a pointer to a node pushed onto each stack is required. However, there is only one jump pointer field (JMPP) for the code node which represents the jump address to the false branch. Our solution uses the back pointer field (BAKP) to reference a node in the IPSTACK, while the jump pointer field (JMPP) references a node in the ENDIFSTACK. Procedure ELSEADJUST performs the resetting of pointers required when an ELSE/ELSEIF is encountered. When the ENDIF is encountered, the BAKP is tested for a nil pointer. A nil BAKP at the top of the ENDIFSTACK indicates that an ELSE/ELSEIF has been seen. This is because procedure ELSEADJUST is the only routine which can clear the BAKP reference before the ENDIF is encountered.

The cause for all the foregoing complexity is the fact that IF-ENDIF has a single false branch which must be a jump past the ENDIF. The tail of the true branch merely falls through the ENDIF. On the other hand, IF-ELSEIF-ELSE-ENDIF can have several false branches, only one of which may jump to the ENDIF. The tails of the all true branches must be jumps to the ENDIF. The logic of BAX59 is designed to recognize and generate equivalent TI-59 code for any of these possibilities.

8. Functions

The most powerful feature of the BAX59 cross-compiler is the translation of functions. Both built-in and user-defined functions are handled. In order to take full advantage of the calculator's capabilities, it was necessary to design three distinctly different types of functions. The first type, referred to as "quick" functions, are the common arithmetic/trigonometric functions such as LOG, SIN, COS. The second type of function harnesses the power of the

Solid State Software module. These are referred to as "long" functions. Both of these first two types are built-in functions. The third type is user-defined "parameter" functions, which are translated from WBASIC source code specifications.

The difference between "quick" and "long" functions is basically the number of TI-59 program steps generated for each. "Quick" functions generally translate to a one or two step TI-59 keycode sequence. However, they may have as many as four steps. Because they are short, "quick" functions are inserted as in-line macros. On the other hand, "long" functions may translate to as many as 15 steps. Therefore, their length requires that they be called as subroutines rather than translated in-line.

Read from the BIFNQF and BIFNLF files respectively, the code for both "quick" and "long" functions is entered into the symbol table during initialization. BIFNQF and BIFNLF may be revised by the user from outside the cross-compiler. By knowing the TI-59 key stroke sequence, a user may add his own functions to either file. As a special user note, the format for additions to these files is critical. The number of key strokes in a function sequence may not exceed the maximum limit for the type of function. If less than that limit, then the end of the sequence must be padded with NOP (68) key strokes to the maximum limit. These limits may be altered by adjusting the system parameters FNQLEN and FNLEN in the constant declaration block of the BAX59 source code.

Most all functions that could be implemented as "quick" have been and are listed in the BIFNQF file. However, only the RND(X) (random number generator) function has been implemented as "long." To illustrate the concept of "long" function, we will walk through the design of RND(X).

Suppose you desire to write a TI-59 program which uses a random number generator. You might write your own pseudo-random number generator subroutine, but the TI-59 has such a routine built into its read-only Solid State Software module. Use of this built-in facility would clearly be more space efficient. WEASIC also has such a function, RND(X). If it had not, it would be possible to write one in WEASIC using the DEF FN_RND(X) statement. Before translation it would be necessary to remove the function definition block and replace the FN_RND(X) calls with RND(X). However, this is not required in our example. You must ensure that the TI-59 registers used by the Solid State Software module to run the RND(X) function are reserved in the LABELF file. This information can be found in the Master Library Manual [Ref. 7], which is the Master Solid State Software module reference guide. RND(X) uses registers 01, 02, 03, 04, 05, 06, and 09. "Long" functions always take a single parameter (even if it is a dummy). Register number 10 has been designated to store this parameter and should also appear on the reserved list. This parameter register assignment may be changed in system parameters of the BAX59 source code if desired. Each time it is encountered within the WEASIC source program, RND(X) will translate as a call to a subroutine whose single parameter is stored in register 10. The first time seen, the RND symbol table node will be linked to a special list (FNLLIST). At the conclusion of code generation, FNLLIST will be traversed and the key sequence which executes RND(X) as well as any other "long" functions on the list, will be added to the code data structure as subroutine bodies.

The real power of this facility lies in its user-controlled flexibility. The user may convert almost any program function in the Solid State Software module into a single parameter "long" function. All he must do is reserve

the correct registers in the LABELF file, list the key sequence in the BIFNLF file, and, if necessary, fix the values of all but one of the function input parameters (or create a dummy). If the function does not exist in WBASIC, then he must write the DEF block for it in order to check program correctness prior to translation.

Having strayed from implementation design toward system utility, we now return to implementation discussion of so-called "parameter" functions. The name given these user-defined functions applies more to how they are implemented rather than to their nature. The parsing routines always expect parameters but do not require them. Parameterless functions are, indeed, recognized.

Although the cross-compiler will correctly translate a function definition (DEF statement or block) whether it occurs before or after its respective call, the linker requires that all subroutine/function bodies be placed after the main program.

When a new function identifier is recognized (by the "FN_" prefix), a new FNPID tagged slot is created for the symbol table. Procedure GENPARG is then called upon to parse actual parameter expressions, generate code which performs their run-time evaluation, and construct the formal parameter list. Parameters, if found, are linked in order to the FNP field of the symbol table slot for the function. While registers are assigned to these parameters, the corresponding formal parameter names cannot yet be entered since they are not known until the function DEF statement is found. Note that formals are assumed to match actual parameters by both order and quantity. There are no checks in BAX59 to insure this correctness. Only a run through the WBASIC interpreter will verify parameter correspondence.

When the function definition is found in the DEF statement, a process similar to parsing the call takes place. The formal parameter names are now inserted into the parameter list attached to the function slot in the symbol table. Before the function body is processed, the slot is pushed onto the FNSTACK. This stack simulates an activation record stack. Each identifier look up that is performed by procedure IDLOOKUP requires that the FNSTACK be examined for active functions. If a formal parameter name is found in an active function parameter list which matches the identifier being sought, then its register assignment is used for code generation. As a result, standard rules of variable visibility and scoping apply. When the end of a function body (FNEND statement) is encountered, the function slot is popped from the FNSTACK and its formal parameters are no longer visible to the run-time environment.

As a final note, the user should know that "parameter" function names receive their own register assignment. This register is the place in which the final value of the function is returned. This register is zeroed during run-time just prior to the execution of the function call. However, after execution the value in this register persists until the next call on the function. This corresponds to an identical situation in the WBASIC run-time environment.

9. Code Resolution

If the physical end of the WBASIC source program is reached, or if a WBASIC END statement is found, parsing is stopped, the bodies for any "long" functions used are generated, and the code data structure is closed out with nil pointers. At this point, the code resolution phase of compilation begins.

The first step in resolution is to locate and insert labels at the destinations of all unstructured subroutine (SBR) calls. These, of course, were generated by GOSUB statements. Since GOSUB is a line-oriented jump, then there is a pointer in the code data structure referencing the destination of that jump. Procedure FINDGOSUBLBL traverses the code data nodes searching for SBR keycodes which are followed by a node with a non-nil jump pointer (JMPP). A very complicated check is made to ensure that the SBR label has not already been inserted by an identical SBR call. If not, then the back pointer (BAKP) is used by procedure PUTGOSUBLBL to assist in the insertion of the label at the jump destination. Once the insertion has been completed, the address field (ADDR) of the JMPP target is set from zero to negative one and the jump pointer (JMPP) is set to nil. This signifies to other routines that this jump has been resolved. The process continues until the end of the code data is reached.

The next step in resolution is to perform a special brand of TI-59 "peephole" optimization. The most common forms of excess parentheses pairs are removed. Such forms as "(RCL nn)" and "(2.333E-12)" will have been generated as a result of parsing even simple assignment statements and expressions. Since the parentheses in these expressions are unnecessary and use up valuable program steps, they are removed, provided they are not referenced by a jump pointer. If referenced by a jump pointer, the node's address field (ADDR) value will be 0 instead of -1 or -2. Removal of these will cause dangling jump references.

Looping and branching generate many place holding NOP keycodes. These are also an unnecessary use of program steps. However, remember that almost all of these were generated to anchor or project jump pointers. Thus, before removal their jump pointers must be reset. Procedure OSQNOP

passes over the code data twice, once to reset all jumps to and from NOP's, and the second to locate and remove the NOP's. It is important to realize that there is a distinction between a useless NOP and one which is acting as a label identifier or a jump address place holder. Because the TI-59 requires that particular keycodes be followed by labels, register numbers, or addresses it is easy to check keyccode usage. This information is actually loaded during initialization into the UNIT field of the CODETEXT record. It is an integer 0..3 which indicates whether the TI-59 code node is a one, two, three, or four keystroke instruction. This information is used to pass over keycodes which are required parts of a larger instruction.

The final stage of resolution is to convert relative jump (pcinter referenced) addresses into absolute (numerical) addresses. This must be the last step because previous code insertion/deletion routines constantly change absolute addresses. At this point no code insertion or deletion occurs. Procedure RESOLVE_ADDR passes over the code data twice. The first pass fills the address fields (ADDR) of all code nodes in sequential order starting at 000. Now that each exact absolute address is known, all jump pcinters which are still marking address space and referencing a destination node can be resolved. A TI-59 coded address consists of two parts. During the second pass procedure INSERT_JMPADDR is called at non-nil jump pointers to read the destination address, split it into its two integer parts, and insert the parts into the address space nodes. Once all jumps have been resolved, the code data structure is ready for output and linking.

10. Input/Output

In this subsection we discuss two input/output related issues: Implementation of I/O constructs and OPTION messages to the compiler. The limited capabilities of the calculator required that file handling and string handling aspects of WBASIC be eliminated from our subset. For similar reasons the I/O constructs which could be translated from WBASIC required restrictions.

While the WBASIC I/O statements INPUT and PRINT normally provide for file management, the BAX59 implementation cannot. The cross-compiler recognizes PRINT followed by any number of simple expressions separated by commas. The TI-59 code generated will evaluate these expressions and print their values (to either the display register or the Texas Instruments PC-100 Printer Cradle). On the other hand, the INPUT statement takes any number of variable identifiers separated by commas. For each identifier in the INPUT list, the TI-59 program halts execution, displays the register assignment for that identifier, and stores the input value entered by the user in the register assigned.

Many programs require the reading of large amounts of data, often at the start of execution. In this situation the INPUT statement tends to generate an excessive amount of program step overhead. Unless the program is designed to be interactive, this overhead unnecessarily increases TI-59 program size. In order to provide a more space efficient means of data entry, a limited translation of the WBASIC DATA and READ statements was designed. In some sense, these statements provide a substitute for file handling. The DATA statements are placed at the beginning of the WBASIC source program. Each statement may be followed by numeric data items separated by commas. The total number of data items in one program is limited to the number of unreserved

registers available in the calculator (based upon the system parameter REGBASE). If this limit is exceeded, a warning message will be issued. READ statements take variable identifiers and may be written with the DATA statements, however, the number of variables input to READ statements should never exceed the number of data items provided by DATA statements. This condition will also cause a warning message and further DATA/READ statements will be ignored. The parse routines make register assignments to the variables in the READ statements, and concurrently build a list which maps the data items to their respective registers and variable names. This list is one form of compiler output. Using the list the user can pre-load TI-59 registers with numeric values and be assured that they will be in correspondence with the translation of variable names. More importantly, no TI-59 program steps are used for this initial input. In fact, the data could be read from a magnetic card into a memory bank prior to execution.

As we have previously implied, there are many forms of output which can be generated by the cross-compiler. Additionally, the user will probably have to do some debugging. We have chosen to provide a primitive set of tools and options which can be toggled on or off from outside the BAX59 source program. The toggles are set or reset by using the OPTICN statement in the WBASIC program. Caution! Do not confuse this statement, which is unique to the BAX59 cross-compiler, with the WBASIC OPTION statement. They are not the same. BAX59 does not recognize WBASIC OPTION parameters and WBASIC does not recognize BAX59 OPTION parameters. Table III lists the possible options available to BAX59 users. To toggle the options, simply include an OPTION statement as the first line of the program to be translated. Desired parameter settings should follow the OPTION reserved word separated by spaces. Positive parameters set the

TABLE III
BAX59 OPTION Statement Parameters

Parameter	Option	Default
+0	Generate linker interface file	false
+1	Generate code for PC-100 printer	true
+2	Optimize out unnecessary parentheses	true
+3	Optimize out unnecessary NOP's	true
+4	Translated TI-59 code to list file	true
+5	Image of symbol table to list file	false
+6	Contents of code structure to list file	false
+7	Each lexical token to terminal	false
+8	Each lexical token to list file	false

toggle true; negative parameters reset the toggle false. In the case of the zero parameter, the sign has no effect.

As a final note, an OPTION statement may not be placed in the WEASIC source program until it is ready for translation. Also, placing an OPTION statement in any line but the first may produce unpredictable results.

E. LINKER

The linker's purpose is to produce a segmented version of the compiled code and present the code in a format that is user friendly. The informal strategy used to accomplish this was discussed in the preliminary design phase of Chapter III. The detailed design that supported the solution strategy called for the linker to operate sequentially through three major phases. In Figure 4.5, the contour diagram for the linker is presented. The preprocessor phase of the linker includes actions from some of the SYSTEM UTILITY procedures and the BLD_SECTBL procedure. The remaining two procedures, COALESCE and INSTRUCTIONS, accomplish the segmenting and postprocessing activities.

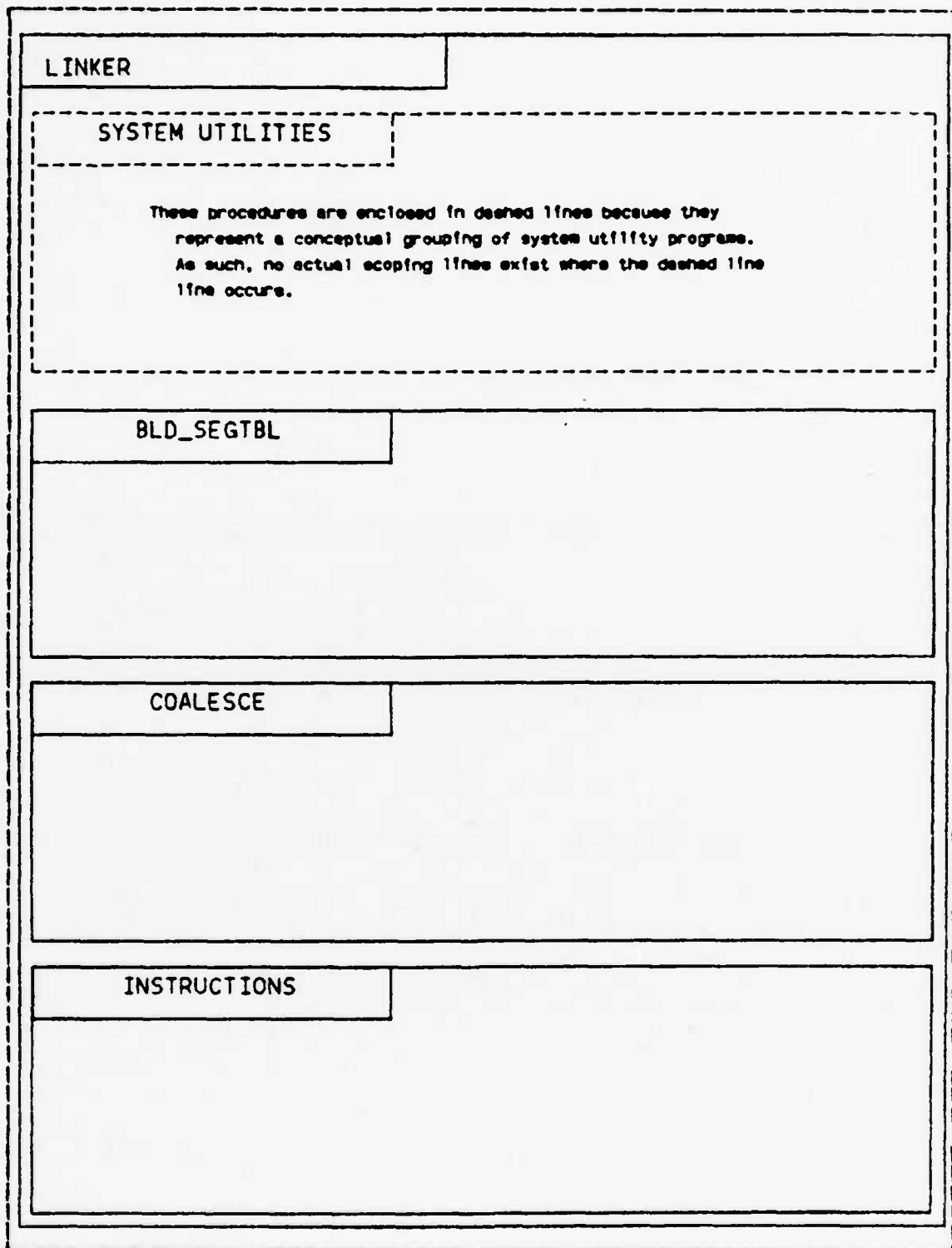


Figure 4.5 Linker Contour.

Each of these major actions were described in the preliminary design phase in Chapter III. The detailed design of these specific operations will be presented in the next sections. Only those major design considerations required for understanding the operation of the major operative phases will be presented.

1. Preprocessor

As was mentioned in the preliminary design, the primary purpose of the preprocessor is to reproduce the compiled linked code list and generate a table that represents the sequential segments of the compiled code.

The informal strategy called for a two step operation. In the first step, textual integer pairs are read from an input file into a data record. Each record is linked to the preceding record forming a linked list which reproduces the linked list of code generated by the compiler. The next step evaluates the linked list to determine where the sequential segments are located. Information concerning each sequential segment is stored in another record and linked to the preceding sequential segment record, thus forming a linked list of sequential segment description records. Evaluation for sequential segments would occur by TI-59 labeled subroutines. Each list of sequential segment records would be pointed to by a header record which contains the subroutine name. Each of these subroutine name header records would be linked to other subroutine name header records in the same order in which they were detected in the generated code.

Two data structures were needed to support this strategy. The first structure comprises a linked list of records. Each record contains all the information that is contained in one program step in the TI-59 calculator, including the address of the instruction and the instruction

integer codes. Each record is linked to the following record by a dynamic pointer, which captures the sequential nature of the compiled code. Another dynamic link is provided for those records containing keycodes that may cause the flow of control to change from a sequential flow. The generated linked list of records is a complete internal representation of the compiled code.

A second data structure is needed to represent a sequential segment of code. Vital program control flow information must be captured by the structure so that segmentation rules may be applied during linker processing. To accomplish this a sequential segment table was developed utilizing a record format to describe each segment. This table record holds data such as segment start address, stop address, whether the segment is covered by an iterative backloop, a list of forward jumps and a list of subroutine invocations that originate within the segment. Each one of these records is linked to the following sequential segment's record. In addition, the sequential segment records are grouped according to subroutine. That is to say, only those sequential segments residing within one TI-59 subroutine definition are connected together in sequential order.

The linked sequential segments are tied together by other records of the same basic type but different variants. Each subroutine grouping of sequential segment records is pointed to by a linked list of header nodes. These nodes contain the name of the subroutine and the subroutine definition address. Each header is linked to another subroutine header in the same order in which subroutine definitions occur within the generated TI-59 code.

To capture information relating to forward jumps, a variant of the sequential segment record is used to build a forward jump list. This list contains the originating

address of the forward jump and the address of the instruction to which control is transferred. Because the actual jump address is used, the link to the jump location is termed relative. Each jump node is dynamically linked to following jump nodes to form a jump list. This list is, in turn, dynamically pointed to by the sequential segment in which the jumps originate.

To capture information regarding subroutine invocations, the same type of structures is used as for the jump node lists. The only difference is that the subroutine lists point to the invoked subroutine in a dynamic manner. That is to say, that a dynamic pointer is set to the first sequential segment of the invoked routine in the sequential segment table. This is basically the only difference between the subroutine invoke list and the forward jump list.

Figure 4.6 is the contour diagram of a conceptual grouping of procedures referred to as the SYSTEM UTILITIES group. These procedures are not explicitly grouped together by code; rather, the grouping is to facilitate discussion and understanding. There are several operations within this group which manipulate the data objects.

In creating the linked compiled code list of records two separate procedures are used. The first procedure is called INPUT. This procedure builds the initial linked structure. It utilizes an input file containing the integer pairs representing TI-59 code steps. Essentially it creates one record for each pair and links the previous record to the new record. The only thing not done is the setting of pointers to represent an indirection in the flow of control.

This is the job of the procedure SET_JMPS. In this procedure, the major activity is the detection in the actual keycode portion of the compiled code of an instruction that represents a possible change in control flow. When one is

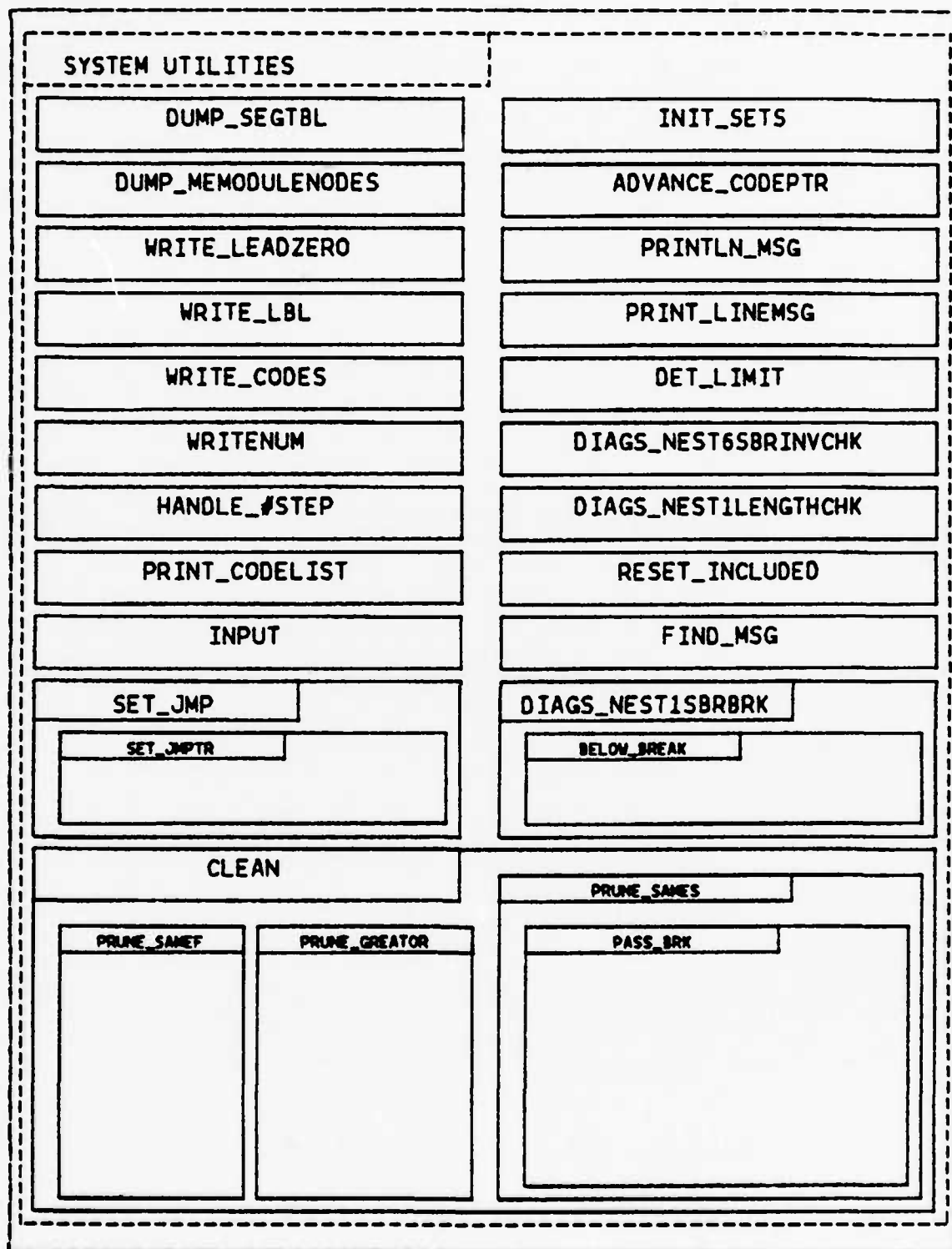


Figure 4.6 System Utilities Contour.

detected, a jump pointer (the indirection pointer) is set pointing to the record containing the next TI-59 program step to be executed.

The operations which create the second data object are a little more complicated and are contained in Figure 4.7. The action of building the sequential segment table data structure is broken down into three steps. The first step begins the formation of the table while the second step completes segment detection. The last step captures other information and ensures that internal interfacing requirements are met.

The first part of this procedure is accomplished through the BLD_PRIMSEGTABLE procedure. This operation passes over the compiled codelist structure and determines where a subroutine starts, stops, or issues a back jump command, and locates the terminal points of the back jump commands. Each of these points is called a critical point. When detected, each critical point is inserted in the segment table data structure under the header node containing the TI-59 subroutine code name which is being processed. In addition, each of the jump commands with their initiation and termination points are inserted into the structure. This completes the first major step.

The second operation is accomplished through the BLD_ADVSEGTABLE procedure. In this procedure the initialized data structure is fleshed out. Up to now only critical points have been inserted. As these are points and are not double ended, segments have not been delineated. This procedure examines the segment data structure and adds points to delineate where a segment starts and ends. It does this by subtracting one from the point following it and taking this to be its end point. This results in a series of records which are all covered by an iterative backloop, with the exception of the first record. This is noted in

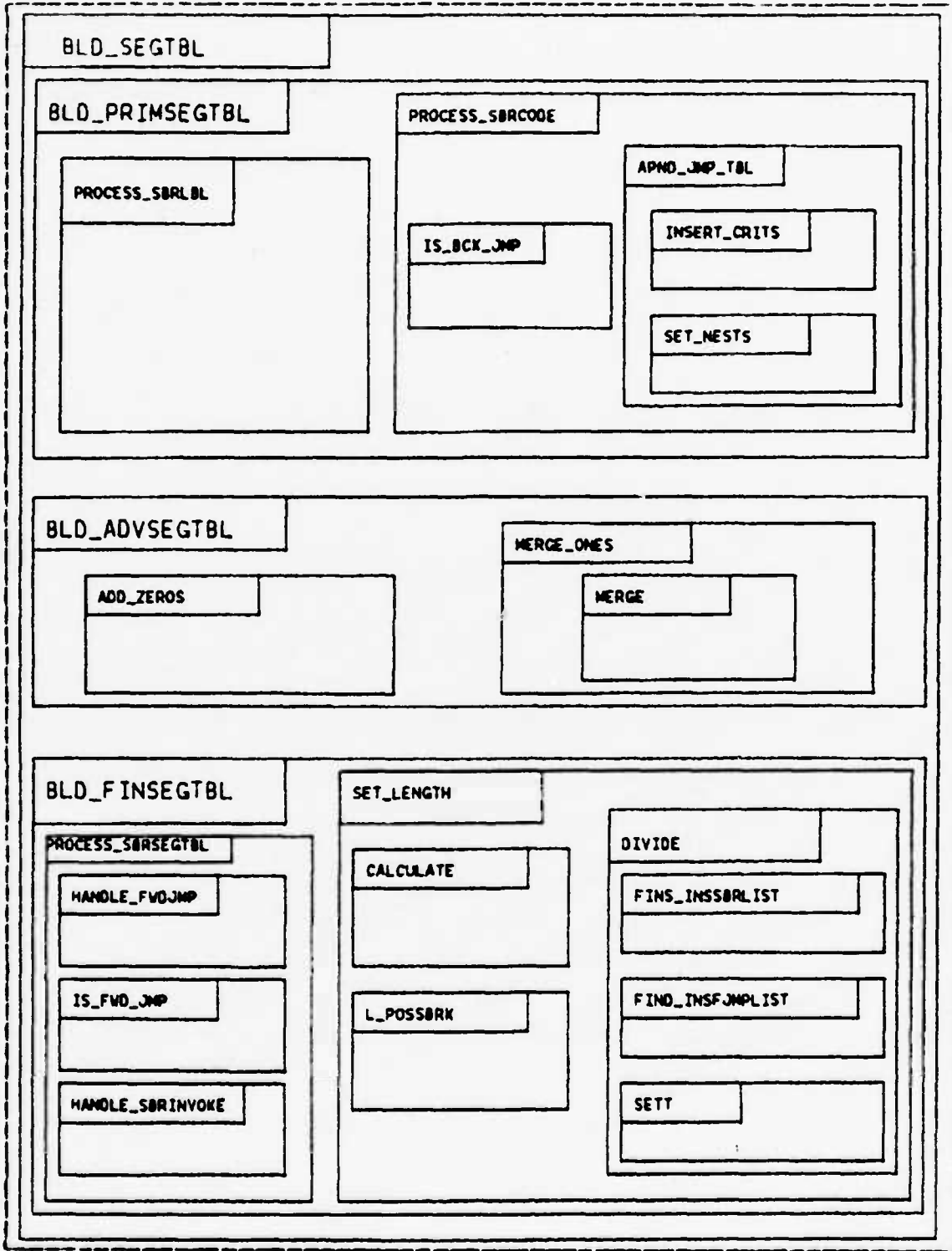


Figure 4.7 BLD_SEG_TBL Contour.

the record. Next, the procedure examines the modified structure and determines by examining the addresses where there are holes in the table. These holes correspond to sequential segments that are not covered by an iterative backloop. These records are then inserted into the structure. Lastly, adjacent segments that are covered are merged to form one record representing a sequential segment that is covered by the largest iterative backloop.

The segment table structure is completed in the last procedure, called `EID_FINSEGTBL`. In this operation two primary things happen. First, `PROCESS_SBRSEGTBL` evaluates the compiled code list and determines where forward jumps and subroutine invocations occur. It inserts these locations into the proper sequential segment that covers the area where the call or invocation occurs. Second, `SETT_LENGTH` checks that each sequential segment does not violate the memory size limit of the calculator. It does this by checking each sequential segment record and calculating a size. If the size is too great, then the segment is divided in half and a new segment is inserted into the table. This is not done for segments that are covered by an iterative loop as this would represent a break of an iterative loop. Other actions that must occur include readjusting forward jump lists and subroutine invocation lists if a division does occur. One interesting point worth noting is that when the length check is made additional steps must be allocated to the actual length to compensate for the possibility of prompting code being added for an invocation to a subroutine that does not currently reside in memory. This is the purpose of `L_FCSSBRK` in Figure 4.7.

The data structure operational procedures access the data structures through pointers which point to the structures. The pointer to the compiled code list is referred to as `BUILT_CODE`. The pointer to the sequential segment table

structure is called SEGTBL. These are the only data which are passed among procedures. One point to remember is that SEGTBL points to the header node list containing the names of the subroutines. The actual sequential segment lists reside underneath the header nodes.

Since understanding the data structure and its construction is essential to understanding the remainder of the linker, an example will be examined to demonstrate the preceding sections.

In Figure 4.8 a sample topology of a TI-59 program is given. It includes four subroutines of various sizes and with various control flow indirections. In looking at the diagram it is important to note the absolute address locations given, for these will be critical to understanding the development of the sequential tables.

As was mentioned, the first operation is the restructuring of the generated TI-59 code. Figure 4.8 represents approximately this structuring. The actual code line is rebuilt internally in the machine and is pointed to by pointer EUILT_CODE.

Figure 4.9 is the completed sequential segment table, without the linked header node list. To understand the concept of sequential segment a comparative look at Figures 4.8 and 4.9 must be made. In Figure 4.9 the first sequential segment is defined as being between addresses 000 and 049. This is reflected in Figure 4.8. When looking at the sequential record one sees that the forward jump information is captured in the forward jump list node which, in this case, is only one node long. When looking at the second sequential segment one notes that there is a nested back jump. The sequential segment is defined to be that segment which is covered completely by back jumps. In this case it extends from 050 to 199. If for some reason the back jumps shown in Figure 4.8 did not fully contain each

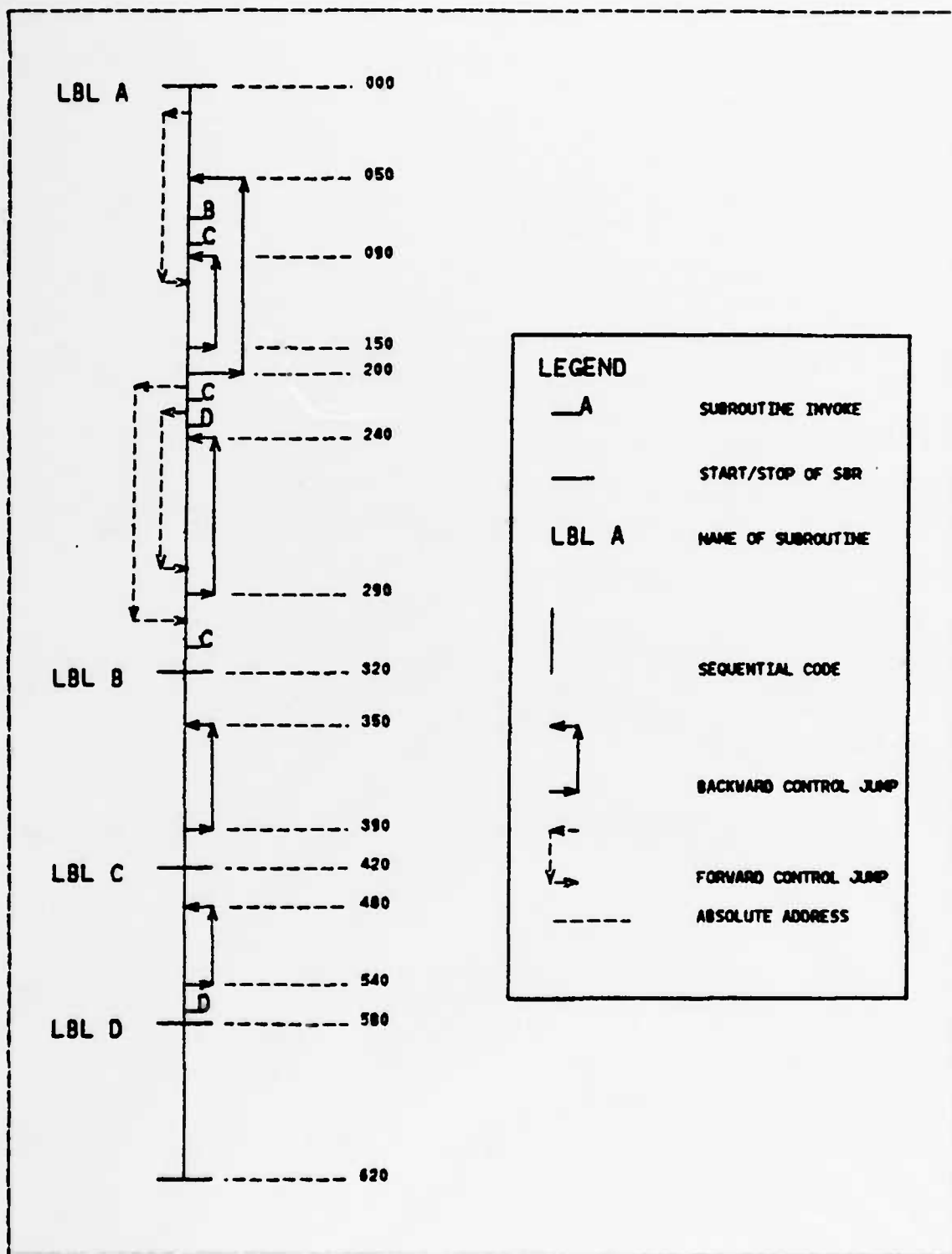


Figure 4.8 TI-59 Code.

other, that is to say, one jump started at 199 and stopped at 090 and the other started at 150 and stopped at 050, then the cover would still extend from 050 to 199. The reason is that this region of code is probably caught in an iterative loop and cannot under any circumstances sustain a break within this cover.

Another point to be made is the manner in which each subroutine's sequential segments are recorded together. In addition each invocation is recorded as is each forward jump. During the completion of the table, invocations to the same routines from different invocation locations are deleted, thus leaving only one link to the called routine for that sequential segment.

A final point concerns the recursive nature of the structure. By assuming that the first subroutine is the main routine and that all other lower level routines are below it (in the sense that they are pointed to from invocation nodes) one can see that any routine used to combine segments can be used on any subroutine's sequential segments. This opens the door for recursion to be used in a bottom up recombination scheme to be discussed later.

Many problems were encountered in the development of the preprocessor phase of the linker. Only the most difficult or annoying will be discussed.

One of the first problems concerned the multiple meaning of program steps in the generated TI-59 code. A separate TI-59 program step may be either a command, register number, flag number, or part of an address. The meaning is dependent on the last valid command. Commands can affect the interpretation of a program step as far as three step positions away (analogous to the concept of one-byte, two-byte, and three-byte instructions in assembly code). This had to be taken into account when doing any operation requiring an interpretation of the code. This

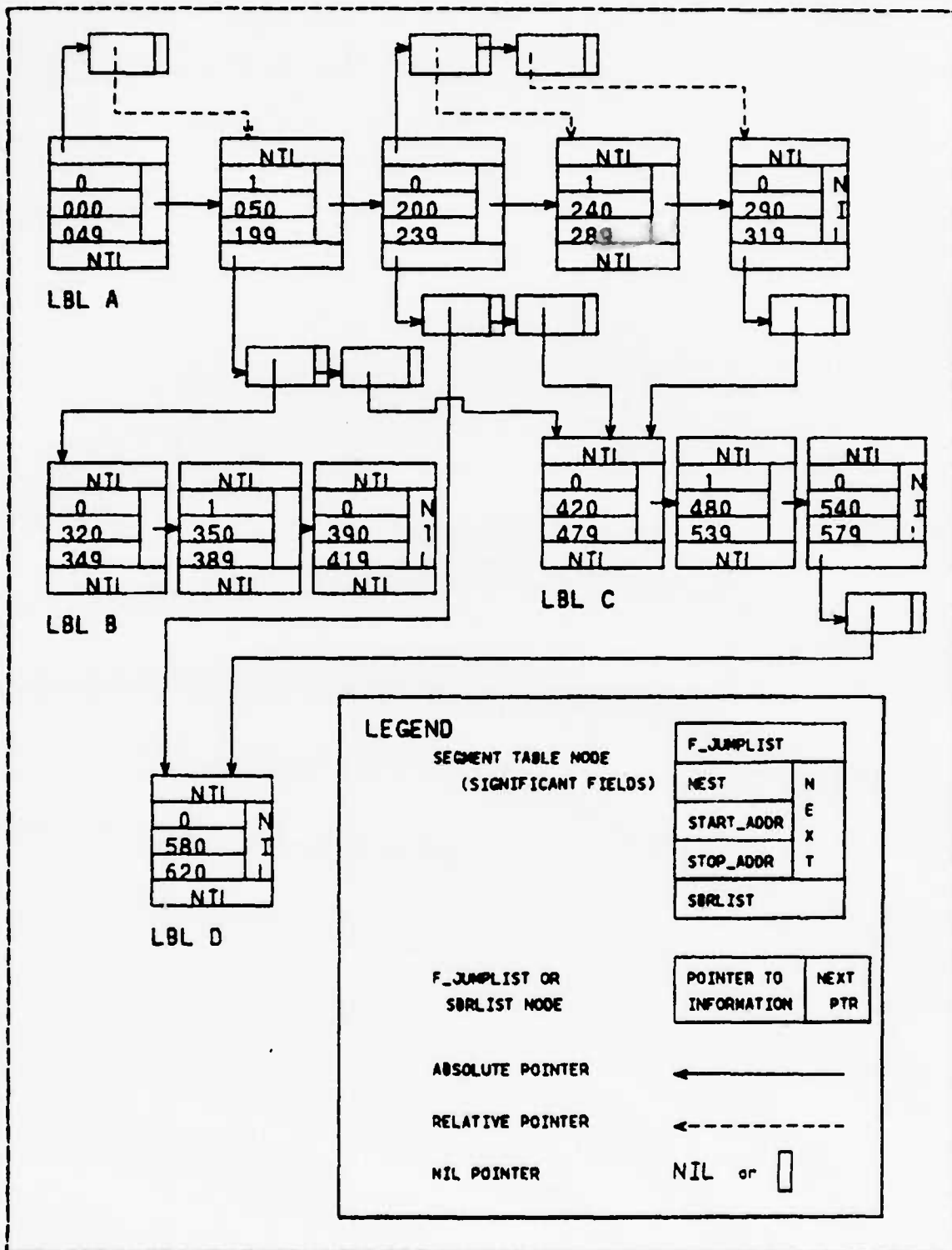


Figure 4.9 Sequential Segment Table.

resulted in special code sets being initialized and special routines being written to print out labels and move the compiled code list pointers. All of these are shown in Figure 4.6.

Improvements in this operative phase could be realized. In the early stages it was decided to separate the compiled code lists from the segment table lists. This was to avoid accidental tampering with the compiled code, since the integrity of the compiled code was the paramount consideration. It would be possible to make the compiled code lists a variant of the segment table. Then, instead of having relative pointer indexes to compiled code addresses, an absolute pointer could be used. This may reduce the size of the program significantly in that types would now be compatible and a reduction in the number of output routines due to the different types would be realized.

2. Segmentor

After the input file has been preprocessed then the linker passes into the segmenting phase of the operation. The routines that support this phase are built into the Pascal procedure called COALESCE depicted in Figure 4.10.

The informal strategy called for the sequential segments of a subroutine to be combined to form larger sequential segments. This recombination would be allowed as long as memory limits were not violated. This required that invoked subroutines be combined first before the caller so as to make room for the invoked routine's code. If the invoked routine could not reside then a break was placed in the dynamic link to the invoked routine and prompting code added to the caller's length for memory size checking purposes.

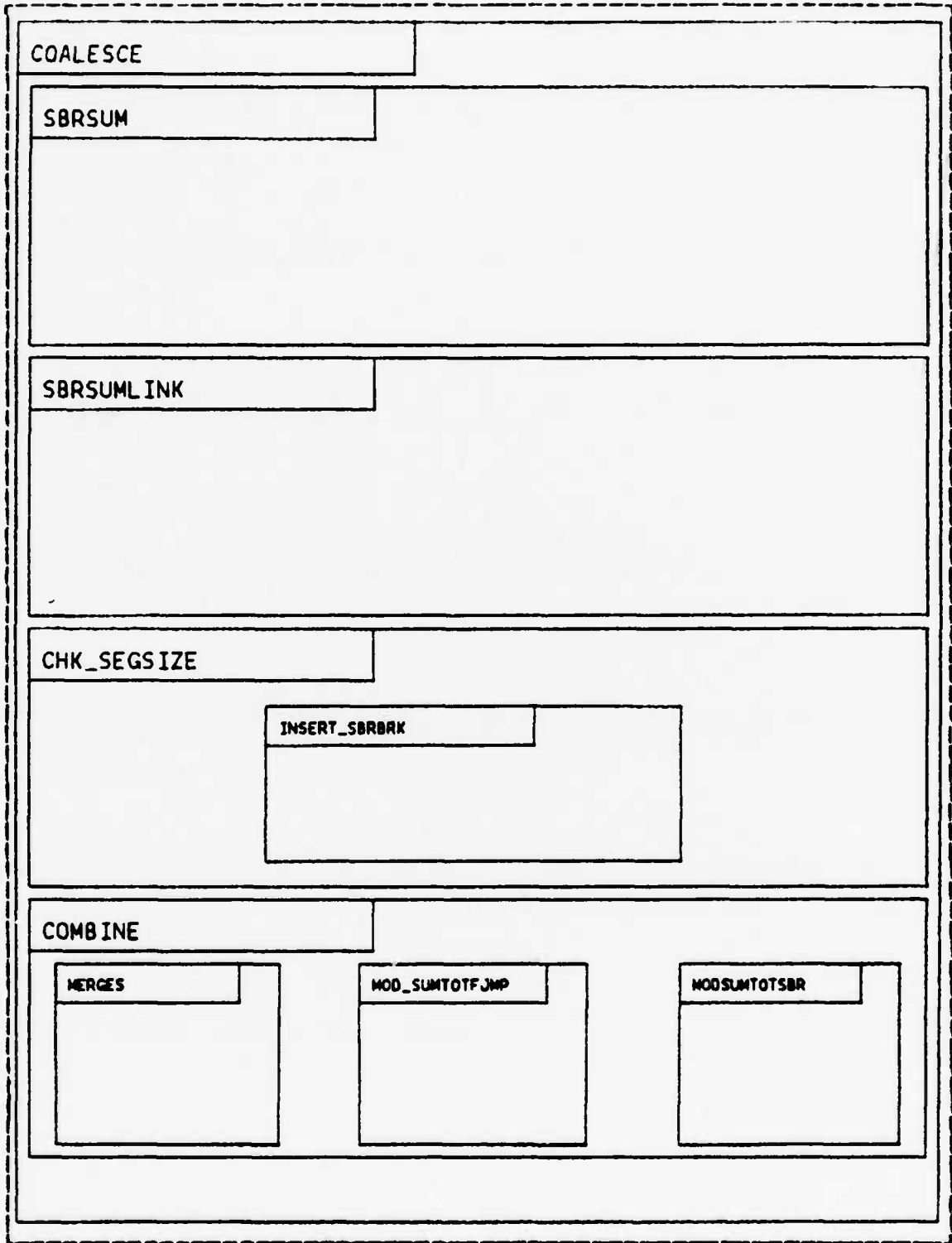


Figure 4.10 COALESCE Contour.

This strategy has one major requirement: the invoked routine must be recombined before the caller can be recombined. For this reason a recursive solution was adopted. In this solution, the main subroutine is recombined. The first part of the recombination process is to ensure that invoked subroutines will reside with the calling sequential segment. If a subroutine is encountered that is not combined or coalesced, then the program will recurse on the new routine. Recursion will close out upon completion of coalescing of a particular routine. When all the sequential segments have been checked then adjacent segments are combined.

Another part of the strategy calls for the combination process to stop when a size limit is encountered. When this happens then some sort of break notation must be used to mark where the limit was exceeded so as to prevent production of code segments that exceed the memory capabilities of the calculator. After the break has been set then the process of recombination begins on the other side of the break with the non-combined segments, starting again with a memory limit of zero.

This process of breaking and checking limits results in the sequential segment table containing break points. These break points delineate the exact locations where program segmentation will occur. These points will mark those portions of code which can fit in the calculator memory according to the rules of segmentation outlined in the preliminary design.

The data structure that supports this strategy is the sequential segment table. No other structure is used. The only addition to the structure is the node referred to as a subroutine invocation break node. This node is inserted between a subroutine invocation list node and the invoked subroutine sequential segment table. All other

changes to the structure involve removing nodes and combining adjacent information into one node.

There are two major activities that support the above strategy. The first activity is the checking of a segment and the second activity is the combining of adjacent segments. Overseeing these activities is a single driver. This topology was suggested by the recursive nature of the solution. The procedures which support these activities are shown in Figure 4.10.

The driver is represented by the Pascal procedure COALESCE. This routine is called whenever a new subroutine is encountered that has not been combined or coalesced. The interior Pascal procedure CHK_SEGSIZE verifies that the specific segment it is looking at, together with all called subroutines on the subroutine invocation list for that segment, will reside in calculator memory. This routine uses SBRSUM and SBRSUMLINK to determine the lengths of invoked routines. It recurses mutually by calling COALESCE in the event that an invoked routine has not yet been coalesced. It determines this by looking at a boolean field in the segment table. This field is set true if the subroutine has been coalesced. The other procedure, COMBINE, accomplishes the actual combination of adjacent sequential segments. It uses the length predictor routines MOD_SUMTCTF_JMP and MOD_SUMTOTSBR to predict a combined length which takes into account any changes that might occur in the subroutine invocation or forward jump lists. If the combined length is within limits then a recombination occurs; if not, pointers are advanced. This means that any sequential segment records which follow the initial sequential segment records are part of a new memory calculation. In other words, any sequential segment links that are not nil represent a break between the linked sequential segments. Upon exiting COALESCE, the subroutine that has

just been coalesced is marked as such in the sequential record's bcclean field reserved for that information.

To visualize the result of the segmenting phase another look at the example is provided. Figure 4.11 represents a segmentation based on a memory limit of 550 steps. Note that each of the invoked subroutines has been coalesced into a single sequential segment. Also note that a break was made in the main subroutine. This is shown by the fact that the main routine is not a single segment record. By examining the table it can be seen that the routine labelled "C" will be copied twice when the two memory sized segments are produced.

To interface between modules in this recursive environment several things were assumed or used. The first was that the data structure would serve as the repository of global data. In addition, a variable would be used to keep track of the current size of the combining memory program steps. This variable was passed as a parameter in order to preserve its value throughout the recursion. All pointers were passed as local parameters. This preserved locations in the data structure as the algorithm progressed through the different levels of recursion.

These operations did require some other work in order to obtain valid data that would correctly calculate code lengths to include multiple copies of subroutines. The problem occurred when there were multiple invocations to the same subroutine from different segments (or even different subroutines) that up to now were all included in the same memory limit calculation. To solve this another field was placed in the segment record to indicate whether or not the particular routine had been included or not in length calculations. Whenever a sum was calculated and a routine included then the field was set true. Whenever a new memory limit was reset back to zero following the implantation of a

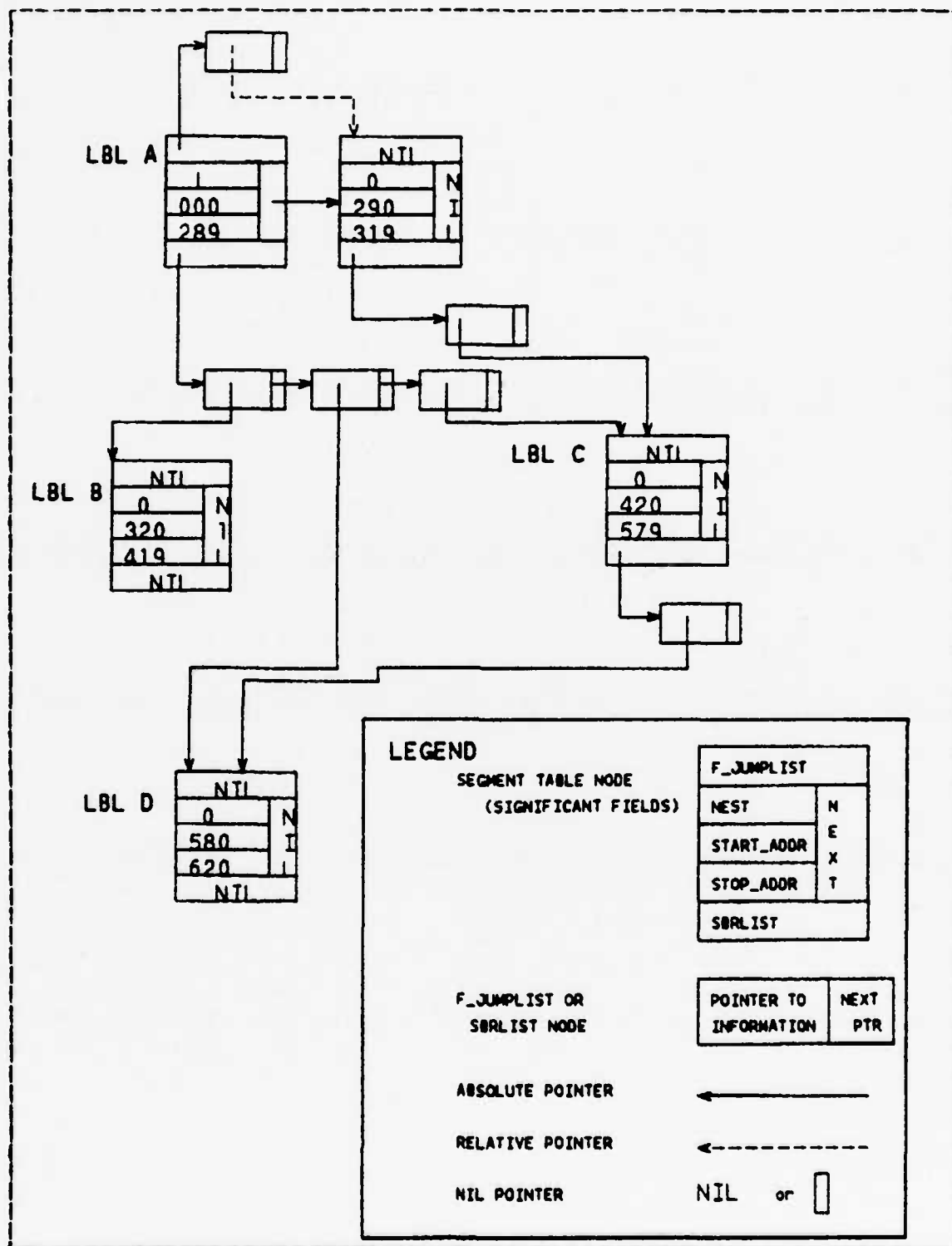


Figure 4.11 Coalesced Segment Table.

break point, a SYSTEM UTILITY procedure was used to reset all the included fields back to false. This means that only one copy of a subroutine would be considered for each calculator sized memory computation.

Future implementations should develop a better method for recording whether a segment is coalesced or included. The inclusion of this field in the segment table record was a "quick fix." This fix results in wasted storage as it is only used in the first record for each subroutine. An improvement would be to use another variant record to record all current data, with the exception of coalesced and included information, for all other sequential segment nodes other than the first sequential segment node. This would save memory.

3. Post Processor

After the segmentation phase, the linker passes into the postprocessor phase. It is this phase that provides the required output for the user.

The informal strategy divides this phase into three distinct operations. The first operation designates the start of each calculator sized segment of code. These segments, which meet the memory requirements of the calculator, are referred to as memory modules. The second operation copies the required code into each segment and inserts any segmentation prompting instructions that are needed for successful code execution. The last operation consists of outputting the segmented code in a user friendly instruction sheet format. This completes all linker actions.

In order to support the informal strategy, several data structures are used, two of which were described earlier. They are the segment table and the compiled code list. At this point the segment table has been coalesced

and contains the locations of the segmentation breaks that will minimize card reads. The compiled code will be copied by segment as delineated in the segment table. A third structure is built in this phase and a fourth structure is provided with the program.

The third structure is referred to as the memory module data structure. It is a Pascal variant of the segment table, which allows compatible pointer references between the two objects. The structure consists of a linked list of head nodes, which are named by respective memory module number. They represent one calculator's worth of available memory programming steps as determined by the calculator partition. Each node of this linked list points to two locations. One location is to the first sequential segment table record node following a segmentation break. The second link is to the copied code that will make up the programming steps of the memory module. In Figure 4.11, there would be two memory module header nodes. The first header node always points to the first sequential segment of the main subroutine. In the example this first memory module would be pointing to the node beginning with address 000. The second memory module node would point to a record that follows a break. This would be to the sequential segment node beginning with address 290. Just as there are no other breaks, there are no other memory module nodes. The other pointers would point to a linked list of code.

The copied compiled code list is a part of the memory module data structure. It is another Pascal variant of the same record type. This list is similar in structure to the compiled code list reproduced during the preprocessor phase. The only difference is in type. Another difference is that there are no jump pointers or dynamic pointers indicating a change in flow of control. The structure is just a linked list of sequential code. This structure is pointed

to by the memory module header node. Another point to be made is that the linked code list, when completed, does contain other code that is needed for prompting. As such it is not a one for one copy of the compiled code list. Lastly, a look at Figure 4.11 will show exactly the segments of code that can be expected to form the two memory module structures. By looking at the sequential segment nodes and following their dynamic pointers of the subroutine lists all required code start and stop addresses are given. It is this "look down" facility of the sequential segment table that make it so useful.

The fourth data object is provided with the linker program. It is a textual file which contains text messages which are used by the linker. Each message is delineated by a \$XXX where XXX is an integer. The linker, when provided the number portion of a message, can easily locate the message. Once located it can either extract values or copy the message verbatim to an output file. This is what occurs during the formatting of the instruction messages.

The operations that build and manipulate the data structures function in three phases. Figure 4.12 is the contour diagram of the subroutine that supports these operations.

The first operation is the construction of the header memory module nodes. This is accomplished by the Pascal procedure BLD_MEMMODULENODES depicted in Figure 4.12. This procedure traverses the segment table and looks for break points. When it finds one, it checks to see if the break has already been detected. If it has not been detected then it builds the header node and assigns it a memory module number. The reason for the check is that the traversing mechanism is based on recursion. In this strategy, traversing is begun with the main subroutine. In Figure 4.11 this would correspond to subroutine LBL A, node

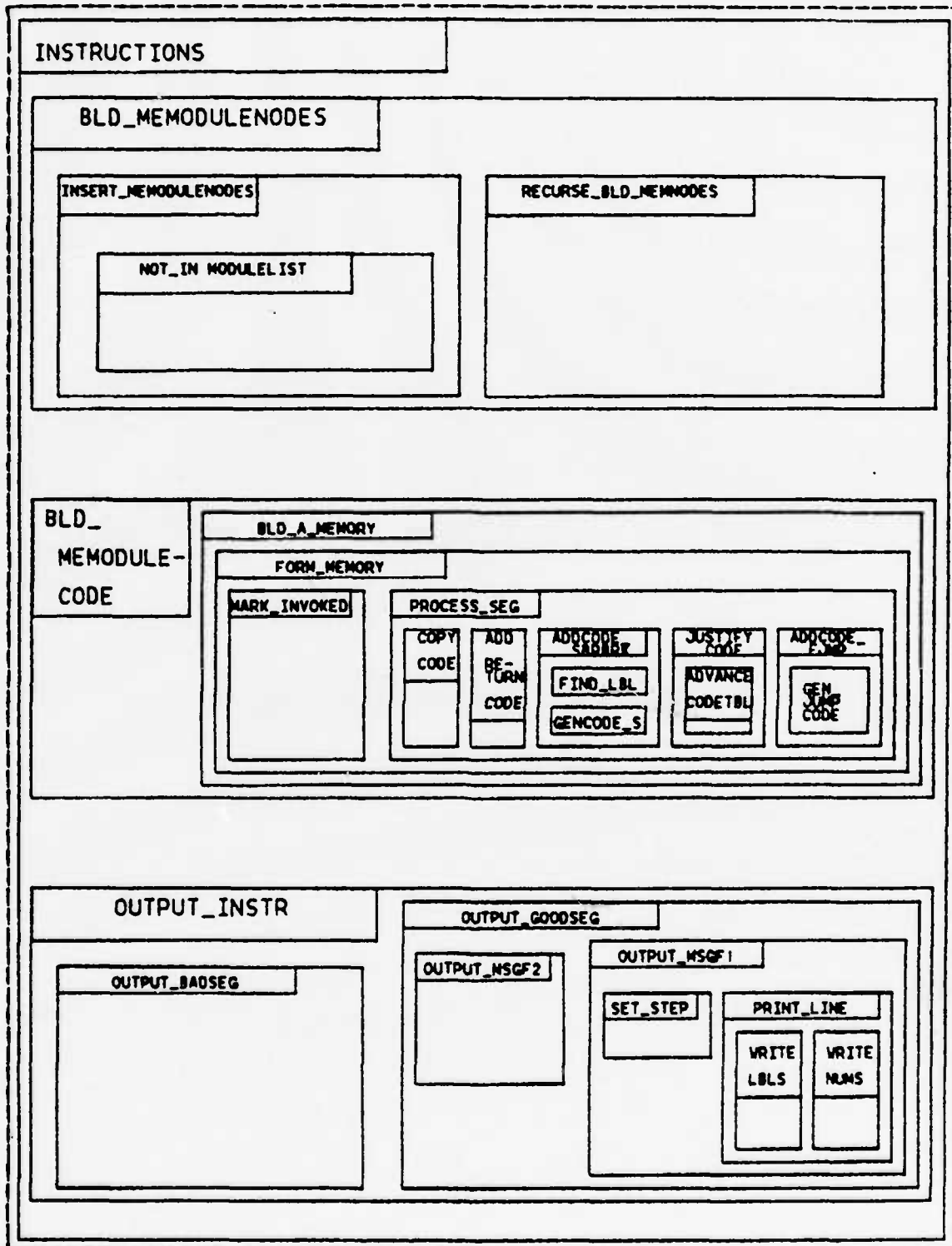


Figure 4.12 INSTRUCTIONS Contour.

start address 000. It then searches right along the same subroutine to detect all breaks of that subroutine. Then it resets back to the start of LBL A and begins to check the subroutine list of each node that comprises LBL A. Recursion is implemented at this point when the subroutine link is traversed and another subroutine is discovered. If a break is discovered in the subroutine list then another memory header node is built and the program bypasses the break and recurses on the next subroutine. This traversal mechanism leads to multiple discoveries of the same breaks. Consequently, the check is made to ensure multiple copies are not placed in the memory module header list.

The next operation consists of copying code from the compiled list, resetting address key codes for jumps and adding prompting code to each specific memory module code list. Figure 4.12 contains the Pascal procedure BLD_MEMMODULECODE which accomplishes the above tasks. This is done by moving down the memory module header list in a sequential fashion. At each header, the link to the segment table is traversed to determine exactly what segments of the compiled code are to be copied. This is the duty of BLD_A_MEMORY in Figure 4.12. Once the start and stop points are determined and copied then recursion is utilized to traverse the subroutine links of the sequential segment table to obtain the required copies of resident subroutines to support the functioning of higher level segments. During this operation the segment table is used as a "check pad," that is, copies are marked included after being copied and are reset upon completion of copying. Another function accomplished during the processing of a memory module segment is the addition of prompting code. Lastly, addresses are reset and justified to include the resetting of jump address key codes to reflect new jumps to internal prompting messages and the absolute address of the originally compiled code list.

Once the memory module data structure is completed then the structure is presented to the user in an instruction type of format. This is the purpose of the Pascal procedure OUTPUT_INSTR. This procedure utilizes two data structures. It uses the provided message file structure and the memory module structure. The first action is to output the instruction introduction. This the procedure does through the use of the message file and the SYSTEM UTILITY programs FIND_MSG, PRINTLN_MSG and PRINT_LINMSG. These are depicted in Figure 4.6. These procedures allow the linker to copy verbatim messages in preformatted form. Once this is done the procedure copies the codelist from each of the memory module lists of code. Once a specific module is copied the driver routine, OUTPUT_MSGF1, prints out specific information to delineate each memory module. After this action is accomplished, the procedure traverses the segment table and prints out additional user information that will aid the user in the execution of his program.

Interfaces between modules are accomplished as usual with pointers. These pointers point to their respective data structures. Global information is recorded in the data structures or in special global variables which are passed as parameters during recursive operations.

One major problem that was encountered and solved in an interesting manner concerns the formatting of the output. The vast amount of instructional information that was required to be output made inclusion in the source code ridiculous. To solve this, the message file system was developed. This system consists of a text file containing preformatted messages and a several procedures located in the SYSTEM UTILITIES contour in Figure 4.6. Each message is delineated by a "\$" and a number. Two types of messages can be processed. One kind of message results in a complete copy from the first line following the message code (\$XXX)

down to the line preceding the next "\$" encountered. The other messages are one-line messages which copied until the "\$" at the end of the message. This gives the programmer the capability to write out blocks of text and to write out text and computer generated information on the same line.

Another capability provided by the package is the ability to search out messages from other files. The procedure FIND_MSG takes as parameters a file as well as a message number. This facility allowed the linker to be loosely coupled with the cross-compiler by interfacing with a message number coded file produced by the cross-compiler. All that the linker needed to know was under which number a required piece of information was coded and the interface file name to affect an interface.

An improvement might be realized in the output of the generated code list. Currently there are two separate sets of procedures used to output code lists. This was primarily due to typing differences. However, the second set of print procedures located in INSTRUCTIONS (see Figure 4.12) is probably more efficient. Furthermore, if the reconstructed code were changed to be a variant of the segment table structure then a reduction in Pascal code lines would be realized through the elimination of a code list group of printing procedures. A further increase in efficiency may be realized in any operations requiring use of the reconstructed compiled code list.

C. INTERFACE ENGINEERING

In any detailed design, careful consideration must be given to interfacing criteria. Interfacing criteria should be as explicit as possible, however this is not always possible. Sometimes, design decisions or engineering interpretations have implications that affect other modules or

submodules. These are generally of an indirect nature in that the interface is implied in the system structure and is not explicitly passed from module to module.

These types of interfaces surface in the detailed design phase. Decisions regarding TI-59 address labelling and structure of the TI-59 subroutine greatly affected the design. In addition, assumptions about the use of structured programming and the prohibition of recursive WEASIG programs facilitated system design. Simple redefinition of the use of WBASIC commands READ/DATA provided an easy form of I/C, but again was an implied interface. These types of implied interfaces will be examined in the following sections since they are critical to understanding the system operations and to future maintenance.

1. Addressing TI-59 SBR

One implied interface resulted from a decision on the mechanism of subroutine invocation which would be used by the system. This decision arose from the fact that the TI-59 calculator may invoke subroutine code in several different ways.

To understand why a decision was required a look at the subroutine naming conventions and procedure for invocation are in order. A subroutine name is composed of two program steps. The first step is the keycode 76. This is the IBL code. It tells the calculator that the next key is a subroutine name. The next program step is the actual subroutine name. The keys which may serve as actual subroutine names may be one of two types. The first type comes from keys which are undefined. That is to say the keys are not used by the calculator to perform calculator functions; they are strictly reserved for naming subroutines. The other type of name comes from defined keys. By this we mean that the calculator uses the keys in some fashion in

addition to naming subroutines. These keys are overloaded: they can have two meanings.

To define which meaning is to be interpreted the calculator requires that a subroutine definition be preceded with a special key called the label key. This alerts the calculator to the fact that the next program step is a subroutine name. To invoke a subroutine, the calculator requires double meaning labels to be preceded with a special key called the invocation key. This alerts the calculator to the double meaning much as the label key. For undefined keys, this alert key is not required though its use will not alter any transfers. There exists another method of invoking subroutines. This calls for the invocation alert key to be followed by an absolute address. It is the duality of meanings which presented problems.

To overcome these problems two decisions were made. The first decision required that all undefined keys be treated as if they were defined keys. This resulted in only one case to be developed to handle subroutine naming. The penalty for this decision is that an added step was generated whenever an undefined key was used as a label and a call to that label was made. The other decision disallowed the use of absolute addresses in the subroutine invocation. All invocations would use labels. This decision carried a penalty in terms of execution speed. The calculator must search program step memory to locate named subroutines whereas address references can be reached directly. On the other hand there are several benefits. One program step was saved since a label requires only one step whereas an absolute address requires three. The other benefit permitted the definition of the invocation alert key as a two step instruction and not a two or three step instruction. This was the primary reason for this implementation. All of the system was designed with this decision in mind.

2. Structured Subroutines

A problem arose with the defining of subroutines. The calculator permits subroutines to be defined within other subroutines. Though this in itself is not extraordinary, the fact that nested definitions may be closed out with the same subroutine return key is not usually permitted. This type of structure made subroutine detection for segmentation purposes very difficult.

To solve this problem it was decided to allow only structured subroutine definitions. That is to say, only one return was permitted for each label. In addition, it was decided to disallow nested definitions. In fact, the decision was made to require that the programmer position his main program first in WBASIC source code and to have all of his subroutines follow in the manner described above.

This interface design decision forced a specific program structure. This structure was easy to detect, easy to compile and easy to segment. These benefits were realized at a cost of not being able to generate efficient or "tricky" code and of reducing the programmer's leeway in developing his WBASIC source code program structure.

3. Recursion

Although TI-59 calculator instructions tend to be evasive on the subject, recursive programs can be executed in a few special situations. Some BASIC languages support recursion, others do not. Although the WBASIC language supports recursion, limitations imposed by the calculator forced us to disallow the translation of recursive source programs. Other reasons for this decision involved complexities which such programs would present to the linker.

4. Input/Output

The development of the input/output structure was designed around the three major limitations of the calculator: programming steps available, storage registers available, and the calculator numeric display. In order to develop an efficient system to allow for input and output, restrictions were placed on some WBASIC commands.

Since the calculator display allows only numeric input and output, then any information passed between a human operator and the calculator must be in numeric form. Prompts used to communicate with the human must be imbedded in the compiled code. In the case of input and output, these prompts have an overhead in that they use up valuable programming steps. For example, with each INPUT command in WBASIC, a total of seven steps are generated to produce a prompt and store a value in the calculator. The problem with this occurs when large numbers of variables need to be input. If 60 variables are required for input then the INPUT command will generate 420 steps. This is clearly unacceptable.

To solve this problem the semantics of the WBASIC commands READ and DATA were modified. Use of these commands within a source program does not increase the number of program steps in the translation. A table of WBASIC variable names assigned to TI-59 registers is produced and placed in the interface file. This allows the human to input all of his data prior to execution without having to pay the penalty of generating extra prompting code.

Another decision concerns the location of the DATA and READ statements: All DATA/READ pairs must occur together at the beginning of the program. The reason for this should be clear. It would be impossible to place the commands in the middle of the program because they would

have no effect. With the exception of a single NOP (which is eliminated during peephole optimization), these commands generate no code. As a result, no run-time modification of variables can occur. Furthermore, DATA/READ statements placed within loops would invalidate the DATA to READ mapping, a static table.

The whole purpose of the redefinition of these commands was to give the user an optional form of I/O. This was expected to increase the efficiency of the generated code. These decisions influenced much of the design of the system.

V. TESTING

The test program and its generated output are provided in Appendices K through P of this report. The test program was chosen for several reasons. The first reason was to test the actual ability of the system to produce a usable TI-59 coded program. The second reason was to attempt to obtain an idea as to the efficiency of the system-generated code. The following sections present a description of the test program and comment on the efficiency of the generated code.

A. TEST PROGRAM DESCRIPTION

In developing a test program several considerations had to be taken into account. The first consideration required that the generated code be verifiable. To verify the generated code, it was decided to program a solution to a problem for which verifiable solutions existed. Verification would be achieved if the system-generated solutions matched those of the existing solutions for the same inputs. The second consideration was to attempt to arrive at some sort of efficiency comparison between the system generated TI-59 program and an optimized hand coded TI-59 program.

The test problem which was selected fit the above criteria. First of all, solutions to known input values existed. This ensured proper verification of the generated code. Secondly, a highly optimized hand coded solution to the problem existed for comparison. The problem selected is called the "Gunnery Problem."

The gunnery problem is to determine firing data for a howitzer cannon. It consists of inputting the following data: piece location, target location, piece corrections and howitzer ballistic coefficients. The output which is generated consists of time to target, elevation to achieve target hit and the lateral deflection (an angular measurement) to align with the target.

The solution involves calculating a range to target as well as an azimuth to the target. The azimuth is then converted to a lateral deflection, which is understood by the piece to be the correct azimuth on which to align. Next, three quadratic equations are solved to determine elevation, time of flight and shell drift. These are applied to the lateral deflection and to the piece elevation. A decision based on the calculated range to target is needed to ensure correct ballistic coefficients are used in the computation. These coefficients are based on the charge which is to be fired to achieve the range.

The hand coded version solution has been in use since 1976. The accuracy of this version was checked by artillery ballistic tables and by the Field Artillery Digital Computer (FADAC), the recognized source of all correct firing data. By using this solution, vast quantities of test input and output were available for program verification runs.

The hand developed version is highly optimized. For example, the hand version stores eight numbers in four storage registers. This is accomplished by storing the numbers on either side of the decimal point in a real number. This real number is then decomposed in a subroutine to obtain the correct number. In addition, this version makes use of calculator commands that were not implemented by the FAX59. For example, there is great use of the indirect store, decrement and skip on zero and polar to rectangular commands. All of these features made the hand coded version a highly optimized program.

B. TEST COMMENTS

The WEASIC solution was coded using basically the same program structure as the hand coded solution. This source code is presented in Appendix K. The interface file generated by the cross-compiler is presented in Appendix O. The final output generated by the linker is presented in Appendix F.

In verifying the accuracy of the generated code, many runs were made using input data for which known solutions existed. There were no deviations from the known solutions in any of the test runs. The conclusion is that the generated code was accurate.

In comparing efficiency a common unit of measurement was needed for comparison. This common unit was chosen to be the TI-59 program step. The reason for this is simple. Both registers and program steps reside in the same memory. Registers occupy eight programming steps. To measure how much memory a program uses it was only necessary to multiply the number of storage registers times eight and add it to the number of program steps to arrive at a figure which measured memory usage within the calculator.

This was done with the BAX59 generated code and the hand coded programs. The hand coded solution used 441 steps and 60 registers for a total step count of 921. The BAX59 generated code used 652 steps and 86 registers for a total step count of 1340. This represented an increase over the hand coded solution of 419 steps or a 45% increase.

A time to solution comparison was made next since the primary purpose of the BAX59 system is to allow an individual to quickly obtain a program capable of running on the TI-59 calculator. To begin with, the hand coded version was developed by three individuals over a period of several weeks. The BASIC program used as input for the BAX59 system

took only one person one day to write and debug. The utility of a higher level language greatly simplified the programming process. It is this savings in programming development which makes the desirability of the BAX59 system readily apparent.

In looking at the system-generated program some more comments can be made about where the relative overhead occurs. Of the total 652 programming steps it was noted that 84 steps were due to prompting code. Six storage registers were used as manual return registers while one register was used as a temporary display storage register. Another register was used in the manual subroutine return prompting scheme. This totals for the Linker an overhead of 148 programming steps. This Linker overhead represents 11% of the total generated steps indicating that the compiler generated at least 34% more code than the hand optimized coded program.

One last comment concerning the actual running of the BAX59 generated program needs to be made. In running the BAX59 code it was determined that time of execution became totally dependent on the amount of cards required to be read in and out. If the head reader in the calculator functioned properly, then this required small amounts of time to accomplish. If however, the head reader malfunctioned, then the reading of cards became an ordeal. In addition, the user had to pay close attention to the program prompting scheme or he would become lost between card reads.

VI. CONCLUSION

In the following discussions, the test program is evaluated together with the actual design. Based on this evaluation several conclusions are drawn and recommendations presented.

A. EVALUATION OF TEST RESULTS

In examining the test data, it is important to realize that this is only a single test. As such, it does not represent the whole set of BASIC programs which may be executed by the BAX59 software system. However, the test does give an insight into the actual efficiencies which might be expected. While actual numerical data is given these data should not be viewed as a statistical analysis of the system. Rather, the data is meant to provide some frame of reference for the discussion of system efficiency.

In examining the test program, it is noted that excess code generated amounts to roughly 45%. Of this, approximately 11% can be attributed to the linker, while 34% can be attributed to the cross-compiler. Although the total overhead seems rather large, the reason for building the system must be recalled. The primary reason is to facilitate the rapid design and implementation of programs on the TI-59 calculator. In view of this, it becomes clear that the overhead is secondary to the problem. Our real yardstick for success is whether or not TI-59 programs can be developed more rapidly than hand coded programs. The test program provides an insight into this side of the problem. Development time was about one order of magnitude faster compared to the hand coded solution.

This rapid development time more than justifies the high overhead. This is true in most academic applications, as most program executions are limited in nature. If on the other hand, a program is to be executed many times, then an optimized hand coded program might be better than the machine generated version. The final decision lies with the user. His program execution requirements, and the amount of time he has available to design and build his solution, will drive his selection.

E. CONCLUSIONS

In view of the target machine limitations, it is probably safe to conclude that the system is a valid first-cut prototype. The prototype proved the desirability and feasibility of the concept, that quick calculator programming can be realized with the minimum of effort. The following paragraphs discuss the prototype's limitations and suggest reasons why the current system is not yet useful as a good production system.

The calculator is severely limited in memory capacity. The TI-59 calculator has only 959 program steps for program usage. The overhead in code generation and segmentation prompts use up 45% of these steps. Together with the fact that only three memory partitions between program steps and storage registers are available, the 45% becomes a significant driving figure in calculator use.

The memory problem restricts the varieties of programs which may be written for translation. Programs may not be written which require a main routine having a long back jump that covers a vast portion of memory. This is because the linker segmentation rules will be violated underneath the covered iterative segment. The segmenter will fail to segment. Thus, only programs which are sequential in nature are suitable for the system.

Another related problem is that the smaller the memory, the more segmentation breaks will occur in the code. The more segmentation breaks, the more card reads will be required to achieve a successful program execution. Often, a problem will arise with the card reader of the calculator. Like any piece of equipment with a motor and magnetic tape head, it is fairly sensitive and prone to failure. If the card reader fails just once in the execution sequence, then there is a high probability that the program will fail to terminate successfully. The minimization of magnetic card reads is desirable for this reason as well as for reduction of user thrashing.

Another restriction occurs in the language subset. Arrays were not implemented in the first prototype design. This limitation impacts directly on the types of programs which can be developed for and translated by BAX59.

Arrays are used primarily for iterative work. Without arrays, iterative work, while still possible, is very limited. Much computer power lies in the ability to execute iteration rapidly. As noted in preceding paragraphs, this limitation occurs as a result of the small memory capacity of the machine. Because of this, it is felt that the implementation of arrays should occur when the prototype is matched with a more capable calculator.

We have suggested major limitations of the system as it currently stands. For these reasons it is felt that the system should be viewed only as a first working prototype. However, we feel that this prototype successfully demonstrates the concept that the power and efficiency of calculator programming can be greatly extended through higher level language programming.

AD-A132 172

DDSIGN AND IMPLEMENTATION OF A BASIC CROSS-COMPILER AND
VIRTUAL MEMORY MA. (U) NAVAL POSTGRADUATE SCHOOL
MONTEREY CA M R KINDL ET AL. JUN 83

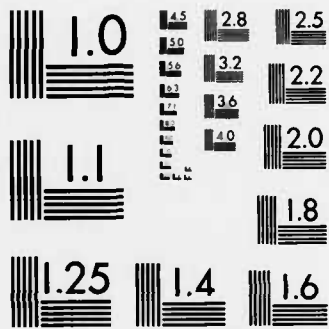
2/4

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

C. RECCMMENDATIONS

If the concept of BAX59 is useful, then the next logical step is to develop the second prototype. The second prototype should not be hindered by the current TI-59's limitations. Otherwise, the next machine should be a more practical one, allowing easy hand held calculator programming. Many firms now market machines which have built-in BASIC language interpreters.

1. Hardware Related Suggestions

In order to avoid the major restriction, namely memory, the BAX59 system must target a larger capacity calculator. This calculator should have about 10,000 program steps and approximately 400 to 500 storage registers. A memory partitioning capability should be available to maximize memory usage.

As a follow on to increased memory capacity, the next prototype should have a hardware device available which will enable the host computer to download the generated calculator program into the target machine. This would eliminate hand punching program steps, which would be prohibitive on a calculator program of 10,000 or more steps.

The linker algorithm examines the dynamic structure of the program to facilitate segmenting the program. This algorithm segments sequential code that is not covered by a back loop. It may be possible to use this algorithm in the development of a single page swapping mechanism/system for a small microcomputer. The purpose of the algorithm in such a system would be to segment a program too large for the microcomputer, in such a manner so as to minimize the number of single page swaps with the system's disk storage unit. Such a system might be desirable for a small microcomputer in which memory is a problem.

2. Array Implementation

A very useful extension of the language subset would be the inclusion of arrays. Coupled with memory expansion, the capability to process arrays would make it possible to do more iterative programming.

A simple but costly implementation of single dimensional arrays is, perhaps, the most feasible approach. For each array declared, three registers will be required to store indexing and access information. Call the WPASIC record fields which will store the assigned register numbers BASE, HOLD, and CALL. BASE stores the number of the array base register (array index 0). During assignment statement translation, HOLD stores the index for an array identifier on the left hand side of the assignment statement equal sign. CALL stores the index of any array identifier currently being evaluated in simple expression. Of course, registers will be necessary for storage of the array itself. The simplest technique is to require that all array index ranges start at zero. Additionally, there will be no run-time checking of range limits. With the foregoing restrictions, estimated TI-59 program step requirements for translation of even simple assignments involving single dimensional arrays are very high. Evaluation of one array reference such as "A(X)" translates to 13 program steps. The assignment statement "A(X)=A(Y)" translates to 26 steps. A more complex reference such as "A(A(X))" requires about 24 steps. Together with the number of registers needed to store values and access data, usage of calculator memory may rapidly approach capacity levels with array manipulation.

Particular array values are accessed with the IND (indirect) instruction. Our basic strategy is to add the evaluated array subscript to the register number in BASE, and store it in HOLD (for left side of assignment

statements) or CALL (for all others). When a value is to be assigned to an array element on the left of an assignment statement, then the right side is evaluated and "STO IND" HOLD stores the value at the correct location. If it is only necessary to evaluate an array item within an expression, then "RCL IND" CALL recalls the value of the appropriate item. More efficient translation schemes might be possible; however, our technique has been tested on the TI-59 calculator, and works well.

The most difficult aspect of implementation is the the task of installing the translation scheme and a parsing scheme. Fortunately, the BASIC language requires explicit array declarations using the DIM statement. Procedure PDIM must be written to parse these declarations, create symbol table entries, and make register assignments. The SLCTRCO record would need an additional variant tag type for array types, call it ARRID. A slot with this tag would carry fields BASE, HOLD, and CALL in its variant part. Finally, we would have to adjust the simple expression parsing and code generating procedures PEXPR and PPRIMARY so that they could recognize array references and act accordingly. Of course, there are other source code adjustments that would be necessary to fine tune the system. However, this discussion has suggested our outline of major steps involved in array implementation.

Description of Command Reserved Words:

DATA: Create an internal data list (see READ, RESTORE)

DATA <integer|real> {, <integer|real>}

DEF: Define a single or multi-line function (see FNEND).

single line: DEF <fn_name> [{<parameter-list>}] = <expr>
multi-line : DEF <fn_name> [{<parameter-list>}]
 ...body of definition
 FNEND

ELSE: Indicate instructions to execute if no IF/ELSEIF conditions were satisfied (see IF, ELSEIF, ENDIF).

ELSE

ELSEIF: Cause execution of a number of statements depending on the given condition (see IF, ELSE, ENDIF).

ELSEIF <boolean-expr>

END: Mark the end-of-source in the program (last line).

END

ENDIF: Indicate the end of an IF-ELSEIF-ELSE structure (see IF, ELSEIF, ELSE).

ENDIF

ENDLCCP: Mark the end of a loop (see UNTIL, WHILE, LOOP).

ENDLCCP

FNEND: Mark the end of a function definition (see DEF).

FNEND

FOR: Mark the start of a loop (see NEXT).

FOR <for-var> = <expr> TO <expr> [STEP <expr>]
 ...statements to execute in loop
NEXT <for-var>

GOSUE: Transfer control to the line specified, until a RETURN is reached (see RETURN).

GOSUE <line#> (Note: GO SUB is not recognized)

GOTO: Transfer control to the line specified (see ON).

GOTO <line#> (Note: GO TO is not recognized)

IF: (1) Cause transfer of control to either of two statements or QUIT a loop depending on a condition.

```
IF <boolean-expr> THEN <line#>|QUIT [ELSE <line#>|QUIT]
NOTE: This is an exception to WBASIC which
      allows any single statement after THEN
      and/or ELSE.
```

(2) Cause execution of either of a group of statements depending on a condition (see ELSE, ELSEIF, ENDIF).

```
IF <boolean-expr>
...statements to execute if expression TRUE
[ELSEIF <boolean-expr>
...statements to execute if 2nd expression TRUE]
[ELSE
...statements to execute if none are TRUE]
ENDIF
```

INPUT: Transmit data from the terminal to a number of variables (see PRINT). No variables stops execution.

```
INPUT [<expr> {, <expr>}]
```

LET: Assign the value of an expression to a variable.

```
[LET] <var> = <expr>
```

LOOP: Mark the beginning of a loop (see WHILE, ENDOOP, UNTIL).

```
LOOP
...statements to execute in loop
ENDLOOP
```

NEXT: Mark the end of a FOR loop (see FOR).

```
FOR <for-var> = ...
...statements to execute in loop
NEXT <for-var>
```

PAUSE: Suspend execution of the program.

```
PAUSE
```

PRINT: Transfer a series of values to printer or display. If no expression is found, a line space will result. (see OPTION)

```
PRINT [<expr> {, <expr>}]
```

QUIT: Leave the current block (WHILE, UNTIL, LOOP).

```
QUIT
```

READ: Transfer data from the list of items specified in DATA statements (see DATA, RESTORE).

```
READ <variable> {, <variable>}
```

REM: Indicate that the line is a comment. The exclamation character (!) may also be used to indicate a comment.

REM [<comment>]

RESTORE: Cause the next READ statement to get data values starting at the first item in the DATA list (see READ, DATA).

RESTORE

RETURN: Transfer control to the statement following the last GOSUB executed (see GOSUB).

RETURN

STOP: Terminate program execution.

STOP

UNTIL: Mark the end of a loop to be executed until the given condition is true (see WHILE, LOOP, ENDLOOP).

LOOP

...statements to execute in loop
UNTIL <boolean-expr>

WHILE: Mark the beginning of a loop to be executed until the given condition is no longer true (see LOOP, ENDOOP, UNTIL).

WHILE <boolean-expr>

...statements to execute in loop
ENDLOOP

OPTICN: Set/reset boolean toggles within BAX59 to control generation of output files.

CAUTION: This is not the WBASIC OPTION!
This OPTICN should be used only after
a correct WBASIC program has been
constructed and is ready for translation.

OPTICN <opt-parm> {<opt-parm>}

where <opt-parm> (option parameter) is
an integer range -8..+8;
sign indicates the direction of toggle:
positive = true/on, negative = false/off;
sign is assumed positive if omitted.

	Default
0 = generate linker interface file.....	false
1 = generate code for PC-100 printer.....	true
2 = optimize out unnecessary parentheses.....	true
3 = optimize out unnecessary NOP's.....	true
4 = translated TI-59 code to list file.....	true
5 = contents of symbol table to list file.....	false
6 = contents of code structure to list file.....	false
7 = each lexical token to terminal.....	false
8 = each lexical token to list file.....	false

Description of Supplemental Reserved Words:

NOT: Negate a boolean expression (see IF, WHILE, UNTIL).

NOT <boolean-expr>

STEP: Designate the increment (decrement) value of a FOR variable (see FOR). (default = +1)

FOR <for-var> = <expr> TO <expr> STEP <expr>

THEN: Mark the beginning of the true branch of a line-oriented IF statement (see IF).

IF <boolean-expr> THEN <line#>|QUIT [ELSE <line#>|QUIT]

TO: Mark the expression which represents the limiting value of a for-variable (see FOR)

FOR <for-var> = <expr> TO <expr> [STEP <expr>]

Symbols and Operators:

<>	not equal	+	addition
<=	less than or equal	-	subtraction
>=	greater than or equal	*	multiplication
**	raise to the power	/	division
<	less than	(open expression
>	greater than)	close expression
=	equal	,	list item separator
!	end of line cmt	.	decimal point

Special Characters:

(not reserved words--recognized directly within the scanner without reference to the RW table)

& Signifies that the current line is continued on the next line or is a continuation of the last line.

00120 REM This comment is too long for one line, so it &
00130 & must be continued on the next line.

- Underscore; used within variable identifier names to assist in readability; also used to designate a user defined function identifier.

LET FINAL SUM VALUE = FIRST_VALUE + SECOND_VALUE
DEF FN_FACTORIAL...

Built-in Functions:

AES ATN CSC IP PI SIN
ACCS COS EXP LOG RND SQR
ASIN COT FP LOG10 SEC TAN

ABS: Returns the absolute value (magnitude) of parameter
 ABS(x)

ACOS: Returns the arccosine (in radians) of parameter
 ACCS(x)

ASIN: Returns the arcsine (in radians) of parameter
 ASIN(x)

ATN: Returns the arctangent (in radians) of parameter
 ATN(x)

COT: Returns the cotangent of parameter angle in radians
 CCT(x)

CSC: Returns the cosecant of parameter angle in radians
 CSC(x)

EXP: Returns the value of e raised to the power x
 EXP(x)

FP: Returns the fractional part and sign of parameter
 FP(x)

IP: Returns the integer part and sign of real parameter
 IP(x)

LOG: Returns the natural logarithm (base e) of parameter
 LOG(x)

LOG10: Returns the logarithm (base 10) of parameter
 LOG10(x)

PI: Returns the value of the constant pi
 (pi = 3.141593 WBASIC ==> pi = 3.14159265359 TI-59)
 PI

RND: Returns a pseudo-random number in the range (0,1)
 RND(x)

SEC: Returns the secant of parameter angle in radians
 SEC(x)

SIN: Returns the sine of parameter angle in radians
 SIN(x)

SQR: Returns the square root of parameter
 SQR(x)

TAN: Returns the tangent of parameter angle in radians
 TAN(x)

APPENDIX B
CONDENSED BAX59 USER'S GUIDE

This guide is intended to be a useful compendium of important points the user should consider when preparing, cross-compiling, and linking a WBASIC source program with the BAX59 system. Included are suggested programming techniques which will optimize and improve resulting TI-59 code. Some of the information contained in the design document is repeated here for the sake of consistency. There are a few previously unmentioned items, many of which are essential to successful use of the system.

1. Whether you are translating a prewritten WBASIC program or one which you are writing yourself, review it for constructs and functions which are not implemented in BAX59. Use Appendix A as a quick reference for this purpose. Finding and eliminating unimplemented functions is more important than constructs. BAX59 will detect and report construct subset errors, however, unimplemented functions are assumed to be variable identifiers and will be entered in the symbol table as such. An error may or may not be reported as a result of faulty syntax; this depends upon the context of the unimplemented function.
2. Every line of the source program must have a line number, including blank lines. Every line number must be in chronological order between 00000 and 99999 exclusive.
3. The end of the source file does not require the END statement; however, whenever an END statement is encountered, the end of the source program is assumed

and translation halts. The END statement will not generate a TI-59 program stop. If you desire that a TI-59 program halt gracefully, you must use the STOP statement(s) in the WBASIC source code.

4. There is no overhead involved in using blank lines or comments. Although a NOP is generated for each, it is subsequently optimized out during resolution. This also allows unconditional jumps to such constructs without cause for concern. However, such practice is not recommended and will hamper debugging.
5. You are strongly urged to practice structured programming. While an unstructured program will be translated, its physical correspondence to the original WBASIC source code is likely to be less recognizable. Also, such a translation will very probably confuse the linker. Ensure that your subroutines have only one entry point and one RETURN, otherwise you will definitely confuse the linker! In order to improve the physical correspondence between source and translation, you are encouraged to use blank lines and comments. This will usually assist in debugging, if required.
6. The structured order of WBASIC program parts should be as follows:
 - A. OPTION statement (for BAX59 only)
 - B. DATA/READ statements
 - C. Main Body (including a at least one STOP)
 - D. Function DEF's and Subroutines
 - E. END statement

Note: The linker expects to find the label representing the main body (LBL A) as the first two TI-59 keycodes.

7. There is only one type of numeric data: real. Integers and reals are both considered reals in the TI-59 run-time environment. Numeric entries without

exponent will always be truncated to ten significant digits. If the entry contains an exponent or if the entry must be converted to exponent notation in order to maintain equivalence, then only eight significant digits will be saved (plus two in the exponent). These rules can have a profound effect upon precision errors in numerical computations.

8. Avoid proliferation of variable names. Variable names use registers, your most precious resource! Whenever possible reuse variable names to prevent new register assignments.
9. Do not forget to reserve registers for your own requirements. The "long" function facility always requires one in the range 10-99. The linker always requires two in the range 00-09. An additional two per segment in the range 10-99 will be taken dynamically by the linker after the cross-compiler has made all its assignments. The interface information is passed through the SCRATCH file. Never reserve the last available register, otherwise a memory overflow will never be reported in a warning message.
10. Optimize expressions using the standard rules of operator precedence. Failure to do so may result in unnecessary generation of parentheses.
11. Avoid use of the STEP option in FOR loops. Rely on the default increment (+1) whenever possible.
12. If a user-defined function is to be applied for its side effects only, then use one variable name to invoke all such function calls. For example:
00120 INVOKE = FN_ALPHA
00130 INVOKE = FN_BETA(X,Y)
00140 INVOKE = FN_GAMMA
13. Although contrary to principles of good structured programming, do not pass any parameters unless

absolutely necessary. Parameter passing uses a great deal of program steps. Furthermore, since actual parameters may be simple expressions, nesting of parentheses can become arbitrarily deep very quickly in a function call. The TI-59 places a limit of nine on the depth of parentheses nests.

14. Remember that the TI-59 allows subroutine (SBR) nesting to a maximum of six levels. This restricts the depth of recursion. However, if the recursive call always returns to the same address, the recursion will probably work. This is because the subroutine return stack will always maintain the correct return address, even if it overflows.
15. BAX59 distinguishes between upper and lower case characters in variable identifiers. Exceptions are the "E" in an exponent, the "FN_" preface of a function, and reserved words. Built-in functions must be written as they appear in the BIFNQP or BIFNLF files (currently, all upper case).
16. WBASIC trigonometric functions compute in radians. By default, TI-59 trigonometric functions compute in degrees. If trigonometric functions are translated, then prior to execution the TI-59 must be reset to the radian mode by entering "2ND RAD."
17. If you plan to modify TI-59 code that has been generated by BAX59, remember that only subroutines have relative addresses. All other addresses are absolute justified. Code insertions or deletions do not rejustify absolute addresses! Unless you are familiar enough with the calculator to know what you are doing, you will create more problems than you fix.
18. The special construct, PAUSE, is provided for assistance in debugging. This is somewhat like a message to the compiler. It translates to the TI-59

keyccdes 82 and 31, which are a void key and the "LRN" key. These keycodes cannot be entered directly into the calculator. Instead, enter "STO 31" followed by the editing sequence "BST BST NOP SST." The original "STC 31" will have been changed to "NOP 31." When encountered during run-time, these keys will interrupt execution and cause the calculator to enter its Learn Mode. Other than stopping execution, no other harmful effects result. To resume processing, simply hit "LRN" to show the contents of the current display register, and "R/S" to continue execution. This facility provides a convenient method of process suspension which corresponds physically and logically to the WBASIC source code.

19. The BAX59 CPTICN statement may provide other useful facilities for debugging. However, most of these were designed for debugging during installation of enhancements to the BAX59 Cross-Compiler. Beware of CPTICN parameters 6, 7, and 8. These tend to produce a great deal of output!
20. The BAX59 system will not execute properly unless all associated data files are available to the host operating system on which BAX59 will run. You may modify the information contained in them, however, you should not change the formats. (These files are Appendices D, E, F, G, H, and J.)
21. Do not design excessively long iterative loops or back jumps. The linker cannot handle them. If iteration is required, design loops which translate to back jumps that are well within the TI-59 memory constraints.
22. The key to successful use of the linker is to break very large programs into smaller parts which can be processed sequentially without much repetition.

APPENDIX C
CROSS-COMPILER SOURCE CODE

```

*****
*
*           BAX59 CROSS-COMPILER
*
* IMPLEMENTS A RECURSIVE DESCENT PARSER AND GENERATES
* A LINKED RECORD DATA STRUCTURE OF TI-59 CODE
* TRANSLATED FROM WEASIC LANGUAGE SOURCE CODE.
* THE DATA STRUCTURE IS USED TO RESOLVE RELATIVE TI-59
* ADDRESSES. A LISTING OF THE ORIGINAL WBASIC PROGRAM
* INCLUDING DETECTED ERRORS IS GENERATED DURING
* TRANSLATION. VARIOUS FORMS OF OUTPUT BESIDES THE
* LISTING FILE CAN BE TOGGLED ON/OFF FROM WITHIN THE
* INPUT FILE USING THE "OPTION" STATEMENT. THIS VERSION
* OF THE PROGRAM IS DESIGNED TO SUPPLEMENT A WATERLOO
* BASIC (WEASIC) LANGUAGE INTERPRETER OR COMPILER. AS
* SUCH, IT DOES NOT DETECT ALL WBASIC SYNTAX OR SEMANTIC
* ERRORS. WBASIC PROGRAMS SHOULD BE SUCCESSFULLY RUN
* IN THE WEASIC SYSTEM ENVIRONMENT PRIOR TO TRANSLATION
* WITH THIS CROSS-COMPILER. THE BAX59 SYSTEM INCLUDES
* AN INDEPENDENT LINKER (TSDRIVER) WHICH WILL PROPERLY
* SEGMENT AND ISSUE INSTRUCTIONS FOR MANUALLY LINKING
* AND EXECUTING A TI-59 PROGRAM GENERATED BY THE CROSS-
* COMPILER BUT WHICH IS TOO LARGE FOR THE CALCULATOR
* MEMORY CAPACITY.
*
*****

```

```

PROGRAM BAX59      (INPUT,   OUTPUT, BASICF, MSGF,
                  RWIENLF, LABELF, CTEXTF, BIFNOF, BIFNLF,
                  OUTFILE, LISTF,  NAMEF,  READP,  SCRATCH);

```

```

*****
*
*           SYSTEM PARAMETERS
*
*****

```

```

-----
*
*           CONST DECLARATIONS (MAIN)
*
-----

```

```

CONST  RWCHARCT   = 270: (* TOTAL # OF CHARS IN RW ARRAY *)
        RWWORDCT   = 72: (* TOTAL # OF WORDS IN RW ARRAY *)
        RWIENGCT   = 9:  (* # OF CHARS IN LONGEST RW *)
        MAXTOKLEN  = 20: (* MAX ACCUM LENGTH => MAX TOKEN *)
        MAXLINLEN  = 66: (* MAX LENGTH OF BASIC TEXT LINE *)
        MAXEASLIN  = 99999: (* MAX BASIC PROGRAM LINE NUMBER *)
        HASHBASE   = 99: (* INDEX OF LAST BUCKET (0-99) *)
        STARTREG   = 00: (* 1ST REGISTER (LOWEST NUMBER) *)
        REGEASE    = 90: (* MAX # AVAILABLE REGISTERS *)
        LBLEASE    = 72: (* MAX # OF AVAILABLE LABELS *)
        FNLEN      = 4:  (* MAX # STEPS IN QUICK FUNCTION *)
        FNLIEN     = 15: (* MAX # STEPS IN LONG FUNCTION *)
        FNIREG     = 10: (* REG USED FOR PARM OF LONG FNS *)
        FNSTACKLIM = 6:  (* MAX SBR/FN NESTING LEVEL *)
        TEXTLEN    = 20: (* MAX # CHARS IN A CODE TEXT LN *)

```

```

*****
*
*          GICBAL DECLARATIONS
*
*****

```

```

-----*
*  TI-59 KEY CODES:                OTHER SYMBOLS:  *
-----*

```

K_ZERO	=	0:	K_EE	=	52:	BLANK	=	' ':
K_1	=	1:	K_OPAREN	=	53:	ENDLIN	=	'@':
K_2	=	2:	K_CPAREN	=	54:	ENDFIL	=	'%':
K_3	=	3:	K_DIVOP	=	55:	PERIOD	=	'.':
K_4	=	4:	K_ENG	=	57:	COMMA	=	',':
K_5	=	5:	K_FIX	=	58:	USCORE	=	'_':
K_6	=	6:	K_INT	=	59:	EXCLAM	=	'!':
K_7	=	7:	K_DEG	=	60:	QUOTE	=	'"':
K_8	=	8:	K_GTO	=	61:			
K_S	=	9:	K_PGMIND	=	62:			
K_EFR	=	10:	K_EXCIND	=	63:			
K_A	=	11:	K_PRDIND	=	64:			
K_E	=	12:	K_MULTOP	=	65:			
K_CC	=	13:	K_PAUSE	=	66:			
K_D	=	14:	K_IFXEQT	=	67:			
K_E	=	15:	K_NOF	=	68:			
K_AFR	=	16:	K_OP	=	69:			
K_BFR	=	17:	K_RAD	=	70:			
K_CFR	=	18:	K_SBR	=	71:			
K_DFR	=	19:	K_STCIND	=	72:			
K_2CLR	=	20:	K_RCLIND	=	73:			
K_INV	=	22:	K_SUMIND	=	74:			
K_INV	=	23:	K_SUBOP	=	75:			
K_CE	=	24:	K_LBL	=	76:			
K_CLR	=	25:	K_IFXGET	=	77:			
K_2INV	=	27:	K_SIGMA	=	78:			
K_ICG	=	28:	K_XBAR	=	79:			
K_CP	=	29:	K_GRAD	=	80:			
K_TAN	=	30:	K_RST	=	81:			
K_XT	=	32:	K_GTOIND	=	83:			
K_XSOR	=	33:	K_OPIND	=	84:			
K_SRTX	=	34:	K_ADDOP	=	85:			
K_XINV	=	35:	K_STFLG	=	86:			
K_FGR	=	36:	K_IFFLG	=	87:			
K_FFR	=	37:	K_DMS	=	88:			
K_SIN	=	38:	K_PI	=	89:			
K_CCS	=	39:	K_LIST	=	90:			
K_IND	=	40:	K_RS	=	91:			
K_STO	=	42:	K_INV_SBR	=	92:			
K_RCL	=	43:	K_DECFPT	=	93:			
K_SUM	=	44:	K_NEG	=	94:			
K_FCWR	=	45:	K_EQUALS	=	95:			
K_CMS	=	47:	K_WRITE	=	96:			
K_EXC	=	48:	K_DSZ	=	97:			
K_EFD	=	49:	K_ADV	=	98:			
K_ABSX	=	50:	K_PRT	=	99:			

{*-----*}
{* TCKEN NUMEERS USED MOST OFTEN OUTSIDE OF MAIN DRIVER. *}
{*-----*}

ERRORTOK = 0:
CMICKEXC = 1:
EQUALTOK = 2:
PLUSTCK = 3:
MINUSTOK = 4:
MULTCK = 5:
DIVTCK = 6:
OPARENTOK = 7:
CPARENTOK = 8:
GTTCK = 9:
LTTCK = 10:
COMMATOK = 11:
EXFTOK = 15:
NOTEQTOK = 16:
GTCTCK = 17:
LTCTCK = 18:

IDENTOK = RWORDCT + 1:
NUMBERTOK = RWORDCT + 2:
ENDLINTOK = RWORDCT + 3:
ENDFILTOK = RWORDCT + 4:

TOTOK = 20:
OFTOK = 21:
CMTOKREM = 24:
NOTTOK = 28:
ANDTOK = 30:
THENTOK = 33:
ELSETOK = 34:
QUITTOK = 38:
STEPTOK = 42:
ENDLOOPTOK = 67:

```

(*)-----(*)
{*
*           TYPE DECLARATIONS (MAIN)
*
*-----*

```

```

TYPE  BASLINRNG = 0..MAXBASLIN;           (* SUBRANGES *)
      TCKENRNG  = 0..FWWORDCT + 4;
      HASHRNG   = 0..HASHBASE;
      ACCRNG    = 0..MAXTOKLEN;
      LE1RNG    = 0..LBLBASE + 1;
      REGRNG    = 0..99;
      KEYRNG    = 0..99;
      CTEXTRNG  = -2..99;

```

```

      LNSTRING  = PACKED ARRAY (.0..MAXLINLEN + 1.) OF CHAR;
      TKSTRING  = PACKED ARRAY (.1..MAXTOKLEN.) OF CHAR;

```

```

      LEVEL1    = ARRAY (.1..RWCHARCT + 1.) OF CHAR;
      LEVEL2    = ARRAY (.1..RWWORDCT + 1.) OF INTEGER;
      LEVEL3    = ARRAY (.1..RWLENGCT + 1.) OF INTEGER;

```

```

      LBLSTACK  = ARRAY (.LBLRNG.) OF INTEGER;

```

```

(*)-----(*)
{*
* DATA STRUCTURE USED FOR WBASIC READ/DATA STATEMENTS;
* ONE DATA ENTRY CONSISTS OF AN OPTIONAL SIGN (DEFAULT
* IS POSITIVE) AND AN INTEGER OR REAL NUMBER.
*-----*

```

```

      DATAITEM = RECCED
                   NUMB : TKSTRING;
                   SIGN : CHAR
                   END;
      DATASTORE = ARRAY (.1..REGBASE.) OF DATAITEM; (* DATAITEM *)

```

```

(*)-----(*)
{*
* DATA STRUCTURE WHICH HOLDS THE TEXT TRANSLATION OF
* ALL TI-59 KEY CODES READ FROM THE CTEXTF FILE;
* UNIT FIELD INDICATES INSTRUCTION TYPE:
* 0 = SINGLE STEP INSTR (INDEPENDENT)
* 1 = 2-STEP INSTR (FOLLOWED BY REG NUMBER)
* 2 = 3-STEP INSTR (FOLLOWED BY ABSOLUTE ADDR)
* 3 = 4-STEP INSTR (FOLLOWED BY REG OR FLAG NUMBER
* AND AN ABSOLUTE ADDR)
*-----*

```

```

      CODETEXT = RECORD
                   UNIT : 0..3;
                   CCDECHAR : PACKED ARRAY
                               (.1..TEXTLEN.) OF CHAR
                   END;
      CTEXTSTORE = ARRAY (.CTEXTRNG.) OF CODETEXT; (* CODETEXT *)

```

```

*-----*
* CODERCD IS A SINGLE NODE IN THE TI-59 CODE DATA *
* STRUCTURE; LINERCD IS A SINGLE NODE IN THE CHAIN OF *
* WBASIC LINE NUMBERS TO WHICH THE TI-59 CODE STRUCTURE *
* IS ATTACHED; THIS PART OF THE CODE DATA STRUCTURE IS *
* USED TO LOCATE PORTIONS OF TI-59 CODE WHICH DIRECTLY *
* CORRESPOND TO WBASIC LINE NUMBERS. *
*-----*

```

```

CODEPTR = @CODERCD;
CODERCD = RECORD
    ADDR : INTEGER;
    KEY : INTEGER;
    JREF : CODEPTR;
    SECP : CODEPTR;
    BAKF : CODEPTR;
END; (* CODERCD *)

```

```

LINEPTR = @LINERCD;
LINERCD = RECORD
    LINC : BASLINRNG;
    LPTR : LINEPTR;
    CPTR : CODEPTR;
END; (* LINERCD *)

```

```

*-----*
* SLCTRCDs ARE SYMBOL TABLE SLOTS; SLOTS ARE ATTACHED TO *
* HASH; HASH IS THE SYMBOL TABLE REPRESENTED AS A STATIC *
* ARRAY OF SLOT POINTERS; EACH SLOT POINTER IN HASH *
* REPRESENTS A SINGLE HASH BUCKET; EACH BUCKET MAY HAVE *
* ANY NUMBER OF SLOTS ATTACHED AT THE HEAD POINTER OF *
* EACH BUCKET (LIMITED ONLY BY MACHINE CAPACITY). *
* SLOTS MAY BE OF 4 DIFFERENT TYPES: VARIABLE IDS, *
* LONG FUNCTION IDS, QUICK FUNCTION IDS, AND PARAMETER *
* (OR PARAMETERLESS) FUNCTION IDS. *
*-----*

```

```

IDTYP = (VARID, FNQID, FNLID, FNPID);
SLCIPTR = @SLCTRCD;
HASH = ARRAY (.HASHRNG.) OF SLOTPTR;
OKEYSEQ = ARRAY (.1..FNQLEN.) OF KEYRNG;
LKEYSEQ = ARRAY (.1..FNLLEN.) OF KEYRNG;
SLCTRCD = RECORD
    IDENT : TKSTRING;
    SLCT : SLOTPTR;
    CASE TYP : IDTYP OF
        VARID : (REGNO : INTEGER;
                AUXREG1 : INTEGER;
                AUXREG2 : INTEGER);
        FNQID : (FNQ : OKEYSEQ);
        FNLID : (FNL : LKEYSEQ);
        FNPID : (FNREGNO : INTEGER;
                FNP : SLOTPTR;
                LEL : LBLRNG;
                FNPLINK : SLOTPTR);
END; (* SLCTRCD *)

```

```

*-----*
*                                     *
*                               VAR DECLARATIONS (MAIN)                       *
*                                     *
*-----*

```

```

*-----*
* OPTICN TOGGLES ARE BOOLEANS WHICH CAN BE SWITCHED FROM *
* WITHIN THE WBASIC SOURCE PROGRAM USING THE "OPTION" *
* STATEMENT; EXCEPT FOR ZERO (LINKER INTERFACE TOGGLE) *
* THE RULE IS THAT A '+' SETS/RESETS TOGGLE TRUE, WHILE *
* A '-' SETS/RESETS TOGGLE FALSE; DEFAULT VALUES ARE *
* INDICATED IN THE COMMENT FOLLOWING EACH DECLARATION. *
*-----*

```

```

VAR   LINK59 : BCOLEAN;          (* OPTION 0 = FALSE *)
      PC100  : BCOLEAN;          (* OPTION 1 = TRUE  *)
      OPTPAR : BCOLEAN;          (* OPTION 2 = TRUE  *)
      CPTNOP : BCOLEAN;          (* OPTION 3 = TRUE  *)
      CODUMP : BCOLEAN;          (* OPTION 4 = TRUE  *)
      SYDUMP : BCOLEAN;          (* OPTION 5 = FALSE *)
      DSDUMP : BCOLEAN;          (* OPTION 6 = FALSE *)
      TOKCUT : BCOLEAN;          (* OPTION 7 = FALSE *)
      TOKIIS : BCOLEAN;          (* OPTION 8 = FALSE *)

```

```

*-----*
* SETS USED IN VARIOUS TESTS FOR CHARS, TOKEN NUMBERS, *
* TI-59 KEY CODES, AND REGISTERS. *
*-----*

```

```

LETTERS, DIGITS, ALFANUM : SET OF CHAR;
DOUBLE1, DOUBLE2        : SET OF CHAR;
SPECIALS, SIGNS         : SET OF CHAR;
SUBERROR, CRITICAL      : SET OF CHAR;
BINCF TOKS, RELOPTOKS   : SET OF TOKENRNG;
TRAILTOKS, SIGNTOKS     : SET OF TOKENRNG;
BEGIN EXPRTOKS          : SET OF TOKENRNG;
NUMERICKEY               : SET OF KEYSRNG;
RESERVE_REG              : SET OF REGRNG;

```

```

*-----*
* RESERVED WORD TABLE CHARACTER AND INDEX ARRAYS. *
*-----*

```

```

RWCHAR : LEVEL1;
FWWCRD : LEVEL2;
RWLENG : LEVEL3;

```

```

*-----*
* SCANNER ASSOCIATED GLOBAL VARIABLES. *
*-----*

```

```

ACCUM          : TKSTRING;
ACCINX         : ACCRNG;
LINEUF         : LNSTRING;
LNBINX        : 0..MAXLINLEN + 1;
LINUM, LLINUM, CLINUM : BASLINRNG;
TCKNUM, LTCKNUM : TOKENRNG;
LCCOUNT, RCOUNT, ECOUNT : INTEGER;
FLAGCMT       : BOOLEAN;

```

```

{*-----*}
{* PARSE ASSOCIATED GLOBAL VARIABLES. *}
{*-----*}

```

```

ERRCRCT, WARNCT : INTEGER;
NEXTREG          : INTEGER;
LELCT           : LBLRNG;
RESERVECT       : REGRNG;

```

```

CTEXT           : CTEXTSTORE;
DATALIST        : DATASTORE;
LATAIX, READIX : 1..REGBASE;
INDEXERROR      : BOOLEAN;
FIRSTREAD       : BOOLEAN;

```

```

CLABEL          : LBLSTACK;

```

```

BUCKET          : HASH;
IDSLOT          : SLOTPTR;
LF, LFCUR       : LINEPTR;
LPLEAD, LPTRAIL : LINEPTR;
CF, CFCUR       : CODEPTR;
FIRSTIP, LASTLF : LINEPTR; (* MARKERS *)
BEGINCP, ENDCP  : CODEPTR; (* MARKERS *)

```

```

FNSTACK, FNLLIST : SLOTPTR;
FNSTACKCT        : INTEGER;

```

```

IFSTACK, ENDIFSTACK : CODEPTR;
LOOPSTACK, ENDICOPSTACK : CODEPTR;
FCRSTACK, NEXTSTACK : CODEPTR;

```

```

{*-----*}
{* FILES *}
{*-----*}

```

```

RWTELF, LABELF, CTEXTF : TEXT; (* INITIAL FILES *)
BIFNCF, BIFNLF         : TEXT; (* BUILT-IN FNS *)
BASICF, MSGF           : TEXT; (* INPUT FILES *)
LISTF, NAMEF, READF    : TEXT; (* OUTPUT FILES *)
OUTFILE                 : TEXT; (* TERMINAL FILE *)
SCRATCH                 : TEXT; (* LINKER INTERFACE *)

```

```

(*****
*
*           PRIMITIVE CHAR ROUTINES
*
*****

```

```

(*-01-----*)
* UPCASE CCNVERTS ANY LOWER CASE EBCDIC (OR ASCII) CHAR *
* TC UPPER CASE EQUIVALENT.
*-----*

```

```
FUNCTION UPCASE (VAR CH : CHAR) : CHAR;
```

```

BEGIN
  IF CH IN ('a'..'i') THEN
    UPCASE := CHR(ORD(CH) - ORD('a') + ORD('A'))
  ELSE IF CH IN ('j'..'r') THEN
    UPCASE := CHR(ORD(CH) - ORD('j') + ORD('J'))
  ELSE IF CH IN ('s'..'z') THEN
    UPCASE := CHR(ORD(CH) - ORD('s') + ORD('S'))
  ELSE
    UPCASE := CH
END;
(* UPPERCASE *)

```

```

(*****

```

```

(*-02-----*)
* TRANSDIGIT RETURNS THE INTEGER VALUE FOR NUMERIC CHARS *
*-----*

```

```
FUNCTION TRANSDIGIT (CH : CHAR) : INTEGER;
```

```

BEGIN
  TRANSDIGIT := ORD (CH) - CRD ('0')
END;
(* TRANSDIGIT *)

```

```

(*****

```

```

(*-03-----*)
* XNUMBER RETURNS INTEGER VALUE OF A NUMERIC CHAR STRING *
*-----*

```

```
FUNCTION XNUMBER (ACCUM:TKSTRING; ACCINX:ACCRNG) : INTEGER;
```

```
VAR I, TEMPNR : INTEGER;
```

```

BEGIN
  TEMPNR := 0;
  FOR I := 1 TO ACCINX DO
    TEMENR := TEMPNR * 10 + TRANSDIGIT (ACCUM(.I.));
  XNUMBER := TEMPNR
END;
(* XNUMBER *)

```

```

(*****

```

```
(* -04 -----*)
{* ZEROPAD WRITES INTEGERS TO AN OUTFILE WITH LEADING 0'S *}
(* -----*)
```

```
PROCEDURE ZEROPAD (VAR WFILE : TEXT; N, ZCT : INTEGER);
```

```
VAR I, TN : INTEGER;
```

```
BEGIN
```

```
  TN := N;
```

```
  REPEAT
```

```
    TN := TN DIV 10;
```

```
    ZCT := ZCT - 1
```

```
  UNTIL TN = 0;
```

```
  FOR I := 1 TO ZCT DO
```

```
    WRITE (WFILE, '0');
```

```
  IF N >= 0 THEN
```

```
    WRITE (WFILE, N:1)
```

```
  ELSE
```

```
    WRITE (WFILE, -N:1, '-')
```

```
END;
```

```
(* ZEROPAD *)
```

```

(*****
*
*           SCANNER
*
*****
)

```

```

(*-05-----*)
(* SCAN USES THE RESERVED WORD ARRAYS TO ISOLATE AND
* RETURN SINGLE TOKENS FROM THE WBASIC SOURCE FILE;
* IT ALSO REWRITES THE SOURCE CODE TO THE LISTF FILE
* ALONG WITH ANY SCANNER DETECTED ERRORS; SCAN INSERTS
* ITS OWN END-OF-LINE AND END-OF-FILE CONTROL CHARACTERS
* INTO ITS LINE BUFFER (LINBUF) AS THESE CHARS ARE
* DETECTED IN THE SOURCE FILE.
*-----*)

```

```
PROCEDURE SCAN (VAR TCKNUM : TCKENRNG);
```

```
VAR   TCHAR : CHAR;
      I      : INTEGER;
```

```

(=====*)
(*-05-01-----*)
(* LINENR RETURNS NUMBER OF THE CURRENTLY SCANNED WBASIC
* LINE; WRITES THE LINE NUMBER TO THE LISTF FILE
* PADDED WITH ZEROS.
*-----*)

```

```
FUNCTION LINENR : BASLINRNG;
```

```
VAR I : EASLINRNG;
```

```

BEGIN
  READ (EASICF, I);
  ZERCPAD (LISTF, I, 5);
  LINENR := I
END;
(* LINENR *)

```

```

(*-----*)
(*-05-02-----*)
(* RDLINE READS TEXT IMMEDIATELY FOLLOWING LINE NUMBER;
* RETURNS THE TEXT IN A MAXLINLEN CHAR BUFFER; UNUSED
* PORTION OF BUFFER IS FILLED WITH BLANKS; WRITES EACH
* CHAR OF TEXT TO THE LISTF FILE AS IT IS READ; REPORTS
* AN ERROR IF NUMBER OF CHARS EXCEEDS MAXLINLEN.
*-----*)

```

```
FUNCTION RDLINE : LNSTRING;
```

```
VAR I, J, LINLENGTH : INTEGER;
    CH : CHAR;
```

```

BEGIN
  I := 0;
  WHILE (NOT (EOLN(BASICF))) AND (I < MAXLINLEN) DO
    BEGIN
      I := I + 1;
      READ (BASICF, CH);
      RDLINE (.I.) := CH;
      WRITE (LISTF, CH)
    END;
  WRITELN (LISTF);
  LINLENGTH := I;
  LNBINX := 0;
  IF LINLENGTH < MAXLINLEN THEN (* FILL UNUSED W/ BLANKS *)
    FOR J := LINLENGTH + 1 TO MAXLINLEN DO
      RDLINE (.J.) := BLANK;

```

```

IF (LINLENGTH = MAXLINLEN) AND (NOT EOLN(BASICF)) THEN
  BEGIN
    WRITE (LISTF, '**F** SCAN ERROR...LENGTH OF TEXT ');
    WRITEIN (LISTF, 'AFTER LINUM > ', MAXLINLEN:2, ' CHARS');
    ERRORCT := ERRORCT + 1;
    REPEAT (* LOCATE THE EOLN CHAR TO RECOVER *)
      GET (BASICF)
    UNTIL EOLN(BASICF)
  END;
  RDLINE(.LINLENGTH + 1.) := ENDLIN; (* INSERT EOLN CHAR *)
  GET (EASICF); (* MOVE FILE PTR PAST PASCAL EOLN CHAR *)
  IF EOF(EASICF) THEN (* OVERWRITE EOLN CHAR IF EOF *)
    RDLINE(.LINLENGTH + 1.) := ENDFIL
END; (* RDLINE *)

```

```

(*-----*)
{* -05-03-----*}
{* GETNOELANK RETURNS FIRST NON-BLANK CHAR STARTING WITH *}
{* THE CURRENT CHAR REFERENCED BY THE LINBUF INDEX. *}
{*-----*}

```

```

FUNCTION GETNOBLANK : CHAR;
BEGIN
  WHILE LINEUF(.LNBINX.) = BLANK DO
    LNBINX := LNBINX + 1;
  GETNOELANK := LINEUF(.LNBINX.)
END; (* GETNOELANK *)

```

```

(*-----*)
{* -05-04-----*}
{* GETCHAR RETURNS THE CHAR FOLLOWING LINE BUFFER INDEX *}
{* AND INCREMENTS THE LINE BUFFER INDEX. *}
{*-----*}

```

```

FUNCTION GETCHAR : CHAR;
BEGIN
  LNBINX := LNBINX + 1;
  GETCHAR := LINEUF(.LNBINX.)
END; (* GETCHAR *)

```

```

(*-----*)
{* -05-05-----*}
{* PUTCHAR INCREMENTS THE ACCUM INDEX AND PLACES A CHAR *}
{* INTO THAT POSITION IN THE ACCUM ARRAY. *}
{*-----*}

```

```

PROCEDURE PUTCHAR (CH : CHAR);
BEGIN
  IF NOT (ACCINX >= MAXTOKLEN) THEN
    BEGIN
      ACCINX := ACCINX + 1;
      ACCUM(.ACCINX.) := CH
    END
END; (* PUTCHAR *)

```

```

(*-----*)

```

```
(* -05-06-----*)
(* PUTANDGET PUTS A CURRENT CHAR INTO ACCUM AND GETS THE *)
(* NEXT CHAR FROM THE LINE BUFFER. *)
(*-----*)
```

```
FUNCTION PUTANDGET (CH : CHAR) : CHAR;
```

```
  BEGIN
    PUTCHAR (CH);
    PUTANDGET := GETCHAR
  END; (* PUTANDGET *)
```

```
(*-----*)
```

```
(* -05-07-----*)
(* NUMSECCCL READS ALL DIGITS WHICH COMPRISE ONE UNSIGNED *)
(* INTEGER AND PUTS THEM INTO THE ACCUM IN SEQUENCE; *)
(* THE NUMBER OF DIGITS SPOOLED IS RETURNED. *)
(*-----*)
```

```
FUNCTION NUMSPOOL : INTEGER;
```

```
  VAR I : INTEGER;
  BEGIN
    IF TCHAR IN DIGITS THEN
      BEGIN
        I := 0;
        WHILE TCHAR IN DIGITS DO
          BEGIN
            TCHAR := PUTANDGET (TCHAR);
            I := I + 1
          END;
        NUMSECCCL := I
      END
    ELSE
      NUMSECCOL := 0
  END; (* NUMSPOOL *)
```

```
(*-----*)
```

```
(* -05-08-----*)
(* EXPONENT READS THE CHARS WHICH COMPRISE AN EXPONENT *)
(* PART, COUNTS THEM, AND PUTS THEM INTO THE ACCUM. *)
(*-----*)
```

```
PROCEDURE EXPONENT;
```

```
  BEGIN
    IF (UPCASE (TCHAR) = 'E') THEN
      BEGIN
        TCHAR := PUTANDGET (TCHAR);
        IF TCHAR IN SIGNS THEN
          TCHAR := PUTANDGET (TCHAR)
        ELSE
          TCHAR := '+';
        ECCUNT := NUMSECCOL
      END
    END; (* EXPONENT *)
```

```
(*-----*)
```

```

(*-05-09-----*)
{* DECIMALPT READS AND COUNTS THE DIGITS IN THE *
{* FRACTICNAL PART OF A NUMBER AND PUTS THEM INTO ACCUM. *}
(*-----*)

```

PROCEDURE DECIMALPT;

```

BEGIN
  IF TCHAR = PERIOD THEN
    BEGIN
      TCHAR := PUTANIGET (TCHAR);
      RCCUNT := NUMSECOL
    END
  END;
(* DECIMALPT *)

```

```

(*-----*)
(*-05-10-----*)
{* ADJUST IS HIGHLY CALCULATOR-DEPENDENT; ADJUSTS ALL *
{* LITERAL NUMERICS TO TI-59 FORMAT; SETS VALUES FOR *
{* LCCUNT, RCOUNT, AND ECOUNT (LEFT, RIGHT, EXPONENT). *
{* TI-59 WILL ACCEPT FROM ITS KEYBOARD A MAX OF 10 DIGITS *
{* (PLUS DECIMAL POINT AND SIGN) FOR INTEGERS OR REALS *
{* WITHOUT EXPONENT, OR 8 DIGITS (PLUS DECIMAL POINT AND *
{* SIGN) AND 2 DIGIT SIGNED EXPONENT. SINCE NUMBERS ARE *
{* SCANNED AND PUT INTO THE ACCUM AS THEY ARE READ IN *
{* WEASIC SOURCE CODE, THOSE WHICH EXCEED THE ABOVE *
{* MAXIMUMS MUST BE CCNVERTED WHILE IN THE ACCUM. *
{* THIS ROUTINE IS DIFFICULT TO UNDERSTAND, MUCH LESS *
{* VISUALIZE, WITHOUT USING A SPECIFIC EXAMPLE TO WALK- *
{* THROUGH CN PAPER. *}
(*-----*)

```

PROCEDURE ADJUST;

```

(*-05-10-01-----*)
{* MCVEXP SHIFTS THE POSITION OF THE EXPONENT PART SO *
{* THAT THE NUMBER HAS MAX OF 8 SIGNIFICANT DIGITS, SIGN, *
{* AND DECIMAL POINT (NOTE: SIGNIFICANCE IS LOST). *}
(*-----*)

```

PROCEDURE MCVEXP;

```

VAR I : INTEGER;
BEGIN
  FOR I := 1 TO (2 + ECOUNT) DO
    ACCUM (.9 + I.) := ACCUM (.LCOUNT + RCOUNT + 1 + I.);
  ACCINX := 11 + ECOUNT
END;
(* MCVEXP *)
(*-----*)

```

```

(*-05-10-02-----*)
(* ADJEXP CONVERTS NUMBERS TO EQUIVALENT TI-59 COMPATIBLE *)
(* FORM BY COORDINATING EXPONENT VALUE ADJUSTMENT WITH *)
(* DECIMAL POINT MOVEMENT AND DIGIT TRUNCATION. *)
(*-----*)

```

```
PROCEDURE ADJEXP (DIFF : INTEGER);
```

```
VAR E1, E2, EXP, NEWEXP : INTEGER;
```

```

BEGIN
  E1 := CRD (ACCUM (.12.)) - ORD ('0');
  E2 := CRD (ACCUM (.13.)) - ORD ('0');
  EXP := 10 * E1 + E2;
  IF ACCUM (.11.) = '-' THEN
    NEWEXP := EXP - DIFF;
  ELSE
    NEWEXP := EXP + DIFF;
  E1 := TRUNC (NEWEXP / 10);
  E2 := NEWEXP - (E1 * 10);
  ACCUM (.12.) := CHR (E1 + ORD ('0'));
  ACCUM (.13.) := CHR (E2 + ORD ('0'));
END;

```

```
(* ADJEXP *)
```

```
(*-----*)
```

```

BEGIN (* ADJUST MAIN *)
  IF (LCCUNT > 10) OR ((LCOUNT > 8) AND (ECOUNT <> 0)) THEN

```

```

    BEGIN
      ACCUM (.9.) := PERIOD;
      IF ECOUNT = 0 THEN
        BEGIN
          ACCUM (.10.) := 'E';
          ACCUM (.11.) := '+';
          ACCUM (.12.) := '0';
          ACCUM (.13.) := '0';
          ACCINX := 13;

```

```

        END;
      ELSE
        BEGIN
          MCVEXP;
          IF ECOUNT = 1 THEN
            BEGIN
              ACCUM (.13.) := ACCUM (.12.);
              ACCUM (.12.) := '0';
              ACCINX := 13;
            END;

```

```

          END;
          ADJEXP (LCOUNT - 8);
          LCCUNT := 8; RCOUNT := 0; ECOUNT := 2;

```

```

        END;
      ELSE IF (LCOUNT + RCOUNT > 10) OR
        ((LCOUNT + RCOUNT > 8) AND (ECOUNT <> 0)) THEN

```

```

        BEGIN
          IF ECOUNT = 0 THEN
            BEGIN
              ACCINX := 11;
              RCOUNT := 10 - LCCUNT;

```

```

            END;
          ELSE
            BEGIN
              MCVEXP;
              RCOUNT := 8 - LCCUNT;
            END;

```

```

          END;

```

```
END;
```

```
(* ADJUST *)
```

```
(*-----*)
```

```
(*--05-11-----*)
(* RWLOOKUP LOOKS UP TOKEN IN RESERVE WORD TBL BASED UPON *)
(* TCKEN LENGTH; RETURNS TOKEN NUMBER; IF NOT FOUND, *)
(* TCKNUM = IDENTOK (IE. TOKEN IS ASSUMED IDENTIFIER). *)
(*-----*)
```

```
FUNCTION FWLOOKUP : INTEGER;
```

```
VAR MATCH : BOOLEAN;
    CHINDEX, WDINDEX, LGINDEX, ACINDEX : INTEGER;
```

```
BEGIN
  LGINDEX := ACCINX;
  WDINDEX := RWLENG(.IGINDEX.);
  REPEAT
    MATCH := TRUE;
    ACINDEX := 1;
    CHINDEX := RWORD(.WDINDEX.);
    WHILE (MATCH) AND (ACINDEX <= LGINDEX) DO
      BEGIN
        IF UPCASE(ACCUM(.ACINDEX.)) <> RWCHAR(.CHINDEX.) THEN
          MATCH := FALSE
        ELSE
          BEGIN
            ACINDEX := ACINDEX + 1;
            CHINDEX := CHINDEX + 1
          END
        END;
      WDINDEX := WDINDEX + 1
    UNTIL (MATCH) OR (CHINDEX = RWORD(.RWLENG(.LGINDEX+1.)));
    IF MATCH THEN
      RWLOCKUP := WDINDEX - 1      (* BACK-UP THE WORD INDEX *)
    ELSE
      RWLOCKUP := IDENTOK
  END;
  RWLOCKUP := IDENTOK
END;
(* RWLOOKUP *)
```

```
(*-----*)
(*--05-12-----*)
(* CMTSPCL READS AND DISREGARDS THE TEXT OF COMMENTS. *)
(* FLAGCMT IS USED FOR COMMENTS CONTINUED ON NEW LINE. *)
(*-----*)
```

```
PROCEDURE CMTSPCL;
```

```
BEGIN
  FLAGCMT := FALSE;      (* RESET COMMENT CONTINUATION FLAG *)
  WHILE NCT (TCHAR IN CRITICAL) DO
    TCHAR := GETCHAR;
    IF TCHAR = 'E' THEN
      FLAGCMT := TRUE
  END;
  (* CMTSPCL *)
```

```
(*-----*)
(*--05-13-----*)
(* RECOVER SCANS AND DISREGARDS THE REMAINDER OF THE *)
(* CURRENT TOKEN AND STOPS AT START OF NEXT TOKEN. *)
(*-----*)
```

```
PROCEDURE RECOVER;
```

```
BEGIN
  WHILE NCT (TCHAR IN (CRITICAL + (.BLANK.))) DO
    TCHAR := GETCHAR      (* SKIP TO NEXT TOKEN AND RETURN *)
  END;
  (* RECOVER *)
  (*=====*)
```

```

BEGIN      (* SCAN MAIN *)

LTCKNUM := TOKNUM;      (* SAVE LAST TOKNUM IN LTOKNUM      *)
TOKNUM  := ERRORTOK;    (* INITIALIZE NEW TOKEN NUMBER *)
ACCINX  := 0;          (* INDICATES TOKEN LENGTH IN ACCUM *)
LCCUNT  := 0;          (* NO. OF DIGITS LEFT OF DECIMAL *)
RCCUNT  := 0;          (* NO. OF DIGITS RIGHT OF DECIMAL *)
ECCUNT  := 0;          (* NO. OF DIGITS IN EXPONENT *)
TCHAR   := GETNOBLANK; (* GET NEXT NON-BLANK CHAR      *)

IF (TCHAR = ENDFIL) THEN
  TCKNUM := ENDFILTCK

ELSE IF (TCHAR = '&') THEN
  BEGIN
    CLINUM := LINENR;      (* READ LINE NO. OF CONT LINE *)
    LINBUF := RDLINE;     (* READ TEXT CF CONT LINE *)
    TCHAR  := GETCHAR;    (* MOVE LNBINX PAST TRAIL "&" *)
    TCHAR  := GETNOBLANK; (* FIND LEADING "&" ON NEW LINE *)
    TCHAR  := GETCHAR;    (* MOVE LNBINX PAST CONT "&" *)
    IF FLAGCMT THEN
      CMTSPOOL              (* COMMENT CONTINUATION *)
    ELSE
      SCAN (TOKNUM)        (* SCAN NEXT TOKEN AFTER "&" *)
    END

  ELSE IF (TCHAR = ENDLIN) THEN
    BEGIN
      LLINUM := LINUM;    (* PASS LINUM TO LAST LINUM *)
      LINUM  := LINENR;   (* READ LINE NO. OF NEW LINE *)
      LINBUF := RDLINE;   (* READ TEXT OF NEW LINE *)
      TCHAR  := GETCHAR;  (* MOVE LNBINX PAST ENDLIN CHAR *)
      TCKNUM := ENDLINTOK (* ASSIGN TOKEN NO. FOR ENDLIN *)
    END

  ELSE IF (TCHAR IN LETTERS) THEN
    BEGIN
      WHILE (TCHAR IN ALFANUM) DO
        BEGIN
          TCHAR := PUTANDGET(TCHAR);
          IF TCHAR = USCORE THEN (* ASSUMES USCORE WILL *)
            TCHAR := PUTANDGET(TCHAR) (* NOT OCCUR AT END *)
          END;
          IF ACCINX <= RWLENGCT THEN
            BEGIN
              TCKNUM := RWLOOKUP;
              IF TOKNUM = CMTOKREM THEN (* LOOK FOR REM CMT *)
                CMTSPOOL
              END
            ELSE
              TCKNUM := IDENTOK
            END
          END

        ELSE IF (TCHAR IN DIGITS) THEN
          BEGIN
            LCCUNT := NUMSFICOL;
            IF TCHAR = PERIOD THEN
              DECIMALPT
            ELSE IF (UPCASE(TCHAR) = 'E') THEN
              PUTCHAR(PERICD);
              EXFCNENT;
              ADJUST;
              TCKNUM := NUMBERTOK
            END
          END
        END
      END
    END
  END

```

```

ELSE IF (TCHAR = PERIOD) THEN
  BEGIN
    DECIMALPT;
    EXECNENT;
    ADJUST;
    TCKNUM := NUMBERTOK
  END

ELSE IF (TCHAR IN DCUBLE1) THEN
  BEGIN
    TCHAR := PUTANDGET(TCHAR);
    IF (TCHAR IN DCUBLE2) THEN
      TCHAR := PUTANDGET(TCHAR);
    TOKNUM := RWLOCKUP
  END

ELSE IF (TCHAR IN SPECIALS) THEN
  BEGIN
    TCHAR := PUTANDGET(TCHAR);
    TCKNUM := RWLOCKUP;
    IF TOKNUM = CMTICXC THEN      (* LOOK FOR EXCLAM CMT *)
      CMTSPOOL
  END

ELSE IF (TCHAR IN SUBERROR) THEN
  BEGIN
    WRITE (LISTF, '**F** SCAN ERROR FOUND AT "', TCHAR);
    WRITEIN (LISTF, '"...CHAR NOT IN THIS SUBSET');
    ERRCRCT := ERRCRCT + 1;
    RECOVER
  END

ELSE
  BEGIN
    WRITE (LISTF, '**F** SCAN ERROR FOUND AT "', TCHAR);
    WRITEIN (LISTF, '"...UNRECOGNIZABLE CHAR');
    ERRCRCT := ERRCRCT + 1;
    RECOVER
  END;

FOR I := (ACCINX + 1) TO MAXTOKLEN DO
  ACCUM(.I.) := BLANK; (* BLANK OUT REMAINDER OF ACCUM *)

(*-----*)
(* DEBUGGING TOOL:  LISTS TOKNUM AND TOKEN AS IT IS READ. *)
IF TOKCUT THEN (* OPTION 7 *)
  WRITEIN (OUTFILE, ' ':6, TCKNUM:2, ' |', ACCUM, '|');
IF TCKLIS THEN (* OPTION 8 *)
  WRITEIN (LISTF, ' ':6, TOKNUM:2, ' |', ACCUM, '|')
(*-----*)

END; (* SCAN *)

```

```

{*****}
{*
{*          ERROR/LINE END HANDLING ROUTINES
{*
{*****}

```

```

{*06-----*}
{* PRECOVER SCANS A LINE AND DISREGARDS TOKENS UNTIL IT  *}
{* FINDS A COMMENT, AN END OF LINE, OR AN END OF FILE.  *}
{*-----*}

```

PROCEDURE PRECOVER;

```

BEGIN
  WHILE NOT (TOKNUM IN TRAILTOKS) DO
    SCAN (TCKNUM)
  END;

```

(* PRECOVER *)

```

{*****}

```

```

{*20-*}
PROCEDURE GENKEY (OPCCDE : INTEGER);

```

FORWARD;

```

{*****}

```

```

{*07-----*}
{* PERROR IS THE GENERAL PURPOSE ERROR HANDLER WHICH:  *}
{* GENERATES SPACE FOR REGISTER OR ADDRESS INSERTION IN *}
{* ORDER TO PREVENT THE CODEDUMP ROUTINE FROM CAUSING A *}
{* SYSTEM ERROR DUE TO INVALID TI-59 CODE GENERATION;  *}
{* ANNOTATES THE LISTING FILE WITH THE ERROR LOCATION;  *}
{* INCREMENTS THE ERRCR COUNT; RECOVERS TO END OF LINE. *}
{*-----*}

```

PROCEDURE PERROR;

```

BEGIN
  GENKEY (-1); { * GENERATE REG/ADDR SPACE TO PROTECT CODE * }
  GENKEY (-1); { * DUMP ROUTINE FROM OPERATING SYS ERROR. * }
  WRITELN (LISTF, '**F** FATAL ERROR FOUND AT ', ACCUM, '');
  ERRCRCT := ERRCRCT + 1;
  PRECOVER

```

(* PERRCR *)

```

{*****}

```

```

{*08-----*}
{* PSUBERROR DISTINGUISHES AN ERROR WHICH IS A RESULT OF *}
{* USING A WEASIC COMMAND NOT IN THIS IMPLEMENTATION.  *}
{*-----*}

```

PROCEDURE PSUBERROR;

```

BEGIN
  WRITELN (LISTF, '**F** SUBSET ERROR FOUND AT ', ACCUM, '');
  ERRCRCT := ERRCRCT + 1;
  PRECOVER

```

(* PSUBERRCR *)

```

{*****}

```

```

(*-09-----*)
(* PWARN IS USED TO ICCATE THE CAUSE OF A WARNING FOR THE *)
(* USER AND TO INCREMENT THE WARNING MSG COUNT; NOTE THAT *)
(* NCRMAI CCMPIATION CONTINUES. *)
(*-----*)

```

PROCEDURE PWARN;

```

BEGIN
  WRITELN (LISTF, '***W** WARNING TRIGGERED AT "', ACCUM, '"');
  WARNCT := WARNCT + 1
END;
(* PWARN *)

```

(*****)

```

(*-10-----*)
(* CIOSELINE IS A GENERAL PURPOSE PROCEDURE USED WHEN THE *)
(* END OF A LINE IS EXPECTED BUT IT IS NOT KNCWN WHETHER *)
(* CR NOT THE LINE BUFFER INDEX IS IN FRONT OF OR AT THE *)
(* END OF LINE/FILE CHAR; MAY ALSO BE A COMMENT PRIOR TO. *)
(*-----*)

```

PROCEDURE CIOSELINE;

```

BEGIN
  IF NOT (TCKNUM IN TRAILTOKS) THEN
    SCAN (TCKNUM);
  IF NOT (TCKNUM IN TRAILTOKS) THEN
    PERRCR
END;
(* CIOSELINE *)

```

```

(*****
*
*           SYMBOL TABLE MANAGEMENT ROUTINES
*
*****

```

```

(*-11-----*)
* HASHVAL RETURNS THE HASH VALUE OF THE STRING CONTAINED *
* IN THE ACCUM (SUM CF ORD VALUES OF CHARS MOD HASHEASE) *
*-----*

```

```

FUNCTION HASHVAL (ACCUM:TKSTRING; ACCINX:INTEGER) : HASHRNG;
VAR  HASHSUM, I : INTEGER;

```

```

BEGIN
  HASHSUM := 0;
  FOR I := 1 TO ACCINX DO
    HASHSUM := HASHSUM + ORD (ACCUM(.I.));
  HASHVAL := HASHSUM MOD (HASHEASE + 1)
END;                                     (* HASHVAL *)

```

```

(*****

```

```

(*-12-----*)
* GETSLOT HASHES ID STRING IN ACCUM INTO SYMBOL TABLE: *
* INSERTS NEW SLOT INTO CORRECT HASH BUCKET AND ENTERS *
* IDENTIFIER NAME INTO IDENT FIELD; RETURNS SLOT POINTER *
* TO THE NEW SLOT BUT VARIANT TAG TYP IS UNDECLARED. *
*-----*

```

```

FUNCTION GETSLOT (ACCUM:TKSTRING; ACCINX:ACCRNG) : SLOTPTR;

```

```

VAR  CURHASH : HASHRNG;
     IDSLCT  : SLOTPTR;

BEGIN
  CURHASH := HASHVAL (ACCUM, ACCINX);
  NEW (IDSLCT);
  IDSLCT?.IDENT := ACCUM;
  IDSLCT?.SLOT := BUCKET (.CURHASH.);
  BUCKET (.CURHASH.) := IDSLCT;
  GETSLOT := IDSLCT
END;                                     (* GETSLOT *)

```

```

(*****

```

```

(*-13-----*)
* FN_CHK RETURNS TRUE/FALSE AS TO WHETHER OR NOT AN *
* ACCUM STRING DESIGNATES A USER DEFINED FUNCTION OR NOT *
*-----*

```

```

FUNCTION FN_CHK (ACCUM : TKSTRING) : BOOLEAN;

```

```

BEGIN
  FN_CHK := FALSE;
  IF (UPCASE {ACCUM {.1.}} = 'F' AND
      (UPCASE {ACCUM {.2.}} = 'N' AND
       (UPCASE {ACCUM {.3.}} = USCORE))
  THEN
    FN_CHK := TRUE
END;                                     (* FN_CHK *)

```

```

(*****

```

```

(*-14-----*)
(* NEWREG RETURNS THE VALUE OF THE NEXT REGISTER WHICH *)
(* AVAILABLE FOR USE AS VARIABLE STORAGE; RESERVED REGS *)
(* ARE SKIPPED AND A COUNT IS MAINTAINED TO IDENTIFY THE *)
(* POINT AT WHICH TOO MANY REGISTERS HAVE BEEN USED. *)
(* NOTE THAT IF THE LAST REG IS RESERVED AN OVERFLOW WILL *)
(* NOT BE DETECTED AND PROCESSING WILL CONTINUE. NOTE *)
(* ALSO THAT A MEMORY OVERFLOW DOES NOT STOP THE PARSER *)
(* FROM ANALYSIS AND CODE GENERATION, HOWEVER THE REG *)
(* SUMMARY MAY NOT REFLECT ACCURATE REGISTER INFO. *)
(*-----*)

```

FUNCTION NEWREG : INTEGER;

```

BEGIN
  WHILE NEXTREG IN RESERVE REG DO
    NEXTREG := NEXTREG + 1;
    NEWREG := NEXTREG;
    IF NEXTREG = (REGBASE + STARTREG) THEN
      BEGIN
        (* NOTE THAT IF LAST REG IS RESERVED, THEN *)
        PWARN; (* THIS WARNING WILL NOT BE TRIGGERED *)
        WRITE (LISTF, '***** MEMORY OVERFLOW...> ');
        WRITE (LISTF, REGBASE:1);
        WRITELN (LISTF, ' VARIABLE NAMES IN USE...REUSING. ');
        NEXTREG := STARTREG (* RESET THE REGISTER STACK *)
      END
    ELSE
      NEXTREG := NEXTREG + 1
  END; (* NEWREG *)

```

(*****)

```

(*-15-----*)
(* NEWLBL RETURNS THE KEY CCDE FOR THE NEXT LABEL ON THE *)
(* LABEL STACK; IF A LABEL STACK OVERFLOW OCCURS, THE *)
(* SUMMARY ITEM WHICH INDICATES NUMBER OF LABELS USED MAY *)
(* NOT REFLECT ACCURATE INFORMATION. *)
(*-----*)

```

FUNCTION NEWLBL : LBIENG;

```

BEGIN
  NEWLBL := CLABEL (.LBLCT.);
  LBLCT := LBLCT + 1;
  IF LBLCT = LBLBASE + 1 THEN
    BEGIN
      PWARN;
      WRITE (LISTF, '***** LABEL OVERFLOW...> ');
      WRITE (LISTF, LBLBASE:1);
      WRITELN (LISTF, ' IN USE...RESET TO 0. ');
      LBLCT := 1
    END;
  END; (* NEWLBL *)

```

(*****)

```

(*-16-----*)
(* FNSTACKLOOK SEARCHES THE FNP ACTIVATION STACK FOR THE *)
(* IDENTIFIER IN ACCUM AND RETURNS A POINTER TO ITS SLOT *)
(* IF IT IS A FORMAL PARAMETER IN AN ACTIVE FNP, OTHER- *)
(* WISE THE POINTER RETURNED IS NIL. *)
(*-----*)

```

```

FUNCTION FNSTACKLOOK (ACCUM : TKSTRING) : SLOTPTR;

```

```

VAR FLOCK, PARMPTR, TPMPTR : SLOTPTR;
    FOUND : BOOLEAN;

```

```

BEGIN

```

```

    FLOCK := FNSTACK;

```

```

    FOUND := FALSE;

```

```

    PARMPTR := NIL;

```

```

    TPMPTR := NIL;

```

```

    WHILE (FLOCK <> NIL) AND (NOT FOUND) DO

```

```

        BEGIN

```

```

            PARMPTR := FLOCK@.FNP;

```

```

            IF PARMPTR <> NIL THEN

```

```

                REPEAT

```

```

                    TPMPTR := PARMPTR;

```

```

                    FCUND := (PARMPTR@.IDENT = ACCUM);

```

```

                    PARMPTR := PARMPTR@.SLOT

```

```

                UNTIL (FOUND) OR (PARMPTR = NIL);

```

```

            FLOCK := FLOCK@.FNPLINK

```

```

        END;

```

```

    FNSTACKLOCK := PARMPTR;

```

```

    IF FOUND THEN

```

```

        FNSTACKLOCK := TPMPTR

```

```

END;

```

```

(* FNSTACKLOCK *)

```

```

(*****

```

```
(*-17-----*)
(* IDICCKUP FIRST SEARCHES ACTIVE FNP STACK (FORMAL      *)
(* PARAMETERS), THEN THE SYM TBL UNTIL IT FINDS THE     *)
(* IDENTIFIER NAME CURRENTLY IN ACCUM; RETURNS POINTER TO *)
(* THE SICT FOR THAT IDENTIFIER; CREATES AND ENTERS A    *)
(* SLCT FOR THE IDENTIFIER IF IT DOES NOT YET EXIST.     *)
(*-----*)
```

```
FUNCTION IDLOOKUP (ACCUM:TKSTRING; ACCINX:ACCRNG) : SLOTPTR;
VAR  HICOK, TLOCK : SLOTPTR;
```

```
(*-----*)
(*-17-01-----*)
(* ENTERID INSERTS AN APPROPRIATE SLOT INTO THE SYM TBL  *)
(* FOR THE IDENTIFIER IN ACCUM AND RETURNS A POINTER TO  *)
(* THAT SICT.                                           *)
(*-----*)
```

```
FUNCTION ENTERID (ACCUM:TKSTRING; ACCINX:ACCRNG) : SLOTPTR;
VAR  IDSICT : SLCTPTR;
```

```
BEGIN
  IDSLOT := GETSLOT (ACCUM, ACCINX);
  IF NOT FN CHK (ACCUM) THEN
    WITH IDSLOT@ DO
      BEGIN
        TYP := VARID;
        REGNO := NEWREG;
        AUXREG1 := -1; (* USED FOR INDEX VARS TO *)
        AUXREG2 := -1; (* FOR-NEXT LOOPS ONLY. *)
        WRITE (NAMEF, ' ':8);
        ZERCPAD (NAMEF, REGNO, 2);
        WRITELN (NAMEF, ' ':3, IDENT)
      END
    ELSE
      WITH IDSLOT@ DO
        BEGIN
          TYP := FNPID;
          FNP := NIL;
          FNPLINK := NIL;
          FNREGNO := NEWREG;
          LEL := NEWLBI;
          WRITELN (NAMEF, ' ':5, FNREGNO:5, ' ':3, IDENT)
        END
      ENTERID := IDSLOT
    END;
  (* ENTERID *)
  (*-----*)
```

```

BEGIN (* IDLOOKUP MAIN *)
  TLCK := FNSTACKLOCK (ACCUM);
  IF TLOCK = NIL THEN
    BEGIN
      HLCCK := BUCKET (.HASHVAL (ACCUM, ACCINX).);
      IF HLCCK <> NIL THEN
        BEGIN
          TLOOK := HLOOK;
          HLOOK := HLCCK@.SLOT;
          WHILE (HLOOK<>NIL) AND (TLOOK@.IDENT<>ACCUM) DO
            BEGIN
              HLOOK := HLOOK@.SLOT;
              TLOOK := TLOOK@.SLOT
            END;
          IF TLOOK@.IDENT = ACCUM THEN
            IDLOOKUP := TLOOK
          ELSE
            IDLOOKUP := ENTERID (ACCUM, ACCINX)
          END
        ELSE
          IDLCOKUP := ENTERID (ACCUM, ACCINX)
        END
      ELSE
        IDLCCKUP := TLOOK
      END;
    END;
  END;
  (* IDLOOKUP *)

```

```

{*****}
{*
{*          CCDE DATA SIRUCTURE MANAGEMENT ROUTINES          *}
{*
{*****}

```

```

{*--18-----*}
{* NEWCODE RETURNS A POINTER TO A NEW CODE DATA NODE;      *}
{* FIELDS ARE INITIALIZED TO A DEFAULT VALUE (EXCEPT KEY) *}
{*-----*}

```

```
FUNCTION NEWCODE (OPCCDE : INTEGER) : CODEPTR;
```

```
VAR CP : CODEPTR;
```

```

BEGIN
  NEW (CP);
  CP@.SECF := NIL;
  CP@.JMFP := NIL;
  CP@.EAKF := NIL;
  CP@.ADDR := -1;
  CP@.KEY  := OPCODE;
  NEWCODE := CP;
END;

```

```
(* NEWCODE *)
```

```
{*****}
```

```

{*--19-----*}
{* PUTKEY INSERTS A CODE DATA NODE INTO THE CODE DATA      *}
{* STRUCTURE IN FRONT OF CPCUR BUT AFTER ENDCP WHICH IS      *}
{* TO SAY THAT THE NODE IS THE NEW END CODE DATA NODE.     *}
{*-----*}

```

```
PROCEDURE PUTKEY (VAR HOLDP : CODEPTR);
```

```

BEGIN
  IF HOLDP = NIL THEN
    HOLDP := NEWCODE (-1);
    ENDCP@.SEQP := HOLDP;
    HOLDP@.SECF := CPCUR;
    ENDCP := ENDCP@.SEQP;
END;

```

```
(* PUTKEY *)
```

```
{*****}
```

```

{*--20-----*}
{* GENKEY FILLS THE CPCUR REFERENCED NODE WITH THE OPCODE  *}
{* PASSED TO IT, AND ATTACHES ANOTHER EMPTY NODE TO THE   *}
{* CODE DATA STRUCTURE AFTER THE ONE JUST FILLED.         *}
{*-----*}

```

```
PROCEDURE GENKEY; (* FORWARD DECLARATION WITH *)
```

```
(* ERROR HANDLING ROUTINES *)
```

```
VAR CP : CCDEPTR;
```

```

BEGIN
  CPCUR@.KEY := OPCODE;
  CPCUR@.SECF := NEWCODE (-1);
  CPCUR := CPCUR@.SECF;
  ENDCP := ENDCP@.SEQP;
END;

```

```
(* GENKEY *)
```

```
{*****}
```

```

(*-21-----*)
(* INSERTKEY IS USED AFTER ALL CODE HAS BEEN GENERATED TO *)
(* INSERT LABELS TO GCSUB REFERENCES; NOTE THAT ENDCP *)
(* MUST NOT BE MOVED OR USED BY THIS PROCEDURE SINCE IT *)
(* NOW RESIDES AT THE END OF THE CODE STRUCTURE AND WILL *)
(* BE USED TO FLAG THE END OF THE STRUCTURE TO TRAVERSING *)
(* PCINTER PROCEDURES. *)
(*-----*)

```

```

PROCEDURE INSERTKEY (CPCODE : INTEGER; VAR LOCUS : CODEPTR);
VAR CP : CCDEPTR;
BEGIN
  CP := NEWCCDE (OPCODE);
  CP@.SEQP := LOCUS;
  LOCUS := CP
END; (* INSERTKEY *)

```

```

(******)

```

```

(*-22-----*)
(* GETNEWHDR CONSTRUCTS AND RETURNS A POINTER TO A LINE *)
(* AND CCDE DATA NODE PAIR; ALL FIELDS OF BOTH NODES ARE *)
(* INITIALIZED; THIS PAIR OF NODES IS USED TO HEAD THE *)
(* CHAIN OF CODE GENERATED FOR A PARTICULAR WBASIC SOURCE *)
(* LINE. (NOTE THAT NEWCODE INITIALIZES CODE NODE) *)
(*-----*)

```

```

FUNCTION GETNEWHDR (LINUM : BASLINRNG) : LINEPTR;
VAR LP : LINEPTR;
    CP : CODEPTR;
BEGIN
  NEW (LP);
  LP@.CPTR := NEWCODE (-1);
  LP@.LINC := LINUM;
  LP@.LPTR := NIL;
  GETNEWHDR := LP
END; (* GETNEWHDR *)

```

```

(******)

```

```

(*-23-----*)
(* PUTNEWHDR INSERTS A HEADER (LINE/CODE DATA NODE PAIR) *)
(* INTO THE CODE DATA STRUCTURE AT THE POSITION OF THE *)
(* CURRENT LINE MARKING POINTER, LPCUR. NOTE THAT THE *)
(* BAKP IS USED HERE TO MARK THE LOCATION OF THE START OF *)
(* CODE CORRESPONDING TO A NEW SOURCE LINE. *)
(*-----*)

```

```

PROCEDURE PUTNEWHDR (VAR LPCUR, LP : LINEPTR);
BEGIN
  LP@.LPTR := LPCUR@.LPTR;
  LPCUR@.LPTR := LP;
  LPCUR := LP;
  ENDCP@.SEQP := LP@.CPTR;
  CPCUR := LP@.CPTR;
  CPCUR@.BAKP := ENDCP
END; (* PUTNEWHDR *)

```

```

(******)

```

```

*-24-----*
* SETLINE COORDINATES THE SET UP OF THE CODE DATA *
* STRUCTURE FOR THE BEGINNING OF CODE GENERATED FOR A *
* NEW WEASIC SOURCE LINE; IF LINE NUMBER HEADER NODES *
* ALREADY EXIST BECAUSE FORWARD JUMP REFERENCES REQUIRED *
* THEIR EXISTENCE, THEN THESE NODES ARE CHECKED FIRST *
* FOR THE CURRENT LINE NUMBER. *
-----*

```

```
PROCEDURE SETLINE (VAR LPCUR, LP : LINEPTR);
```

```
VAR LNUMBER : BASLINRNG;
```

```
BEGIN
```

```
  LNUMBER := LINUM;
```

```
  IF TOKNUM = ENDLINTOK THEN
```

```
    LNUMBER := LLINUM;
```

```
  IF LPCUR@.LPTR <> NIL THEN
```

```
    BEGIN
```

```
      IF (LPCUR@.LPTR@.LINO) = LNUMBER THEN
```

```
        BEGIN
```

```
          LPCUR := LPCUR@.LPTR;
```

```
          ENDCP@.SEQP := LPCUR@.CPTR;
```

```
          CPCUR := LPCUR@.CPTR;
```

```
          CPCUR@.BAKF := ENDCP
```

```
        END
```

```
      ELSE
```

```
        BEGIN
```

```
          LP := GETNEWHDR (LNUMBER);
```

```
          PUTNEWHDR (LPCUR, LP)
```

```
        END
```

```
      END
```

```
    ELSE
```

```
      BEGIN
```

```
        LP := GETNEWHDR (LNUMBER);
```

```
        PUTNEWHDR (LPCUR, LP)
```

```
      END
```

```
END;
```

```
(* SETLINE *)
```

```
(*****)
```

```
(* -25 -----*)
(* SETJMPEXT SETS THE EXTERNAL JMP PTR FROM CFCUR@.JMPP *)
(* TO THE WEASIC LINUM INDICATED BY THE GOTO OR GOSUB *)
(* -----*)
```

```
PROCEDURE SETJMPEXT (LINUM : BASLINRNG) ;
```

```

BEGIN
  IF LINUM > LPCUR@.LINC THEN
    BEGIN (* FORWARD JUMP *)
      LPLEAD := LPCUR@.LPTR;
      LPTRAIL := LPCUR@;
      WHILE (LPTRAIL@.LINO < LINUM) AND (LPLEAD <> NIL) DO
        BEGIN
          LPLEAD := LPLEAD@.LPTR;
          LPTRAIL := LPTRAIL@.LPTR
        END;
        IF LPTRAIL@.LINO = LINUM THEN
          BEGIN
            CFCUR@.JMPP := LPTRAIL@.CPTR; (* SET JMPP PTR *)
            LPTRAIL@.CPTR@.ADDR := 0 (* MARK JMPP TERMINAL *)
          END
        ELSE
          BEGIN
            LPTRAIL@.LPTR := GETNEWHDR (LINUM);
            LASTLP := LPTRAIL@.LPTR;
            CFCUR@.JMPP := LASTLP@.CPTR; (* SET JMPP PTR *)
            LASTLP@.CPTR@.ADDR := 0 (* MARK JMPP TERMINAL *)
          END
        END
      END
    END
  ELSE
    BEGIN (* BACKWARD JUMP *)
      LPLEAD := FIRSTIP;
      WHILE (LPLEAD@.LINO <> LINUM) AND (LPLEAD <> NIL) DO
        LPLEAD := LPLEAD@.LPTR;
        IF LPLEAD@.LINC = LINUM THEN
          BEGIN
            CFCUR@.JMPP := LPLEAD@.CPTR; (* SET JMPP PTR *)
            LPLEAD@.CPTR@.ADDR := 0 (* MARK JMPP TERMINAL *)
          END
        END
      END (* ASSUMES THAT A LINO = LINUM ALWAYS EXISTS ! *)
    END
  END; (* SETJMPEXT *)

```

```

(*****
*)
*)          FUNCTION CALL ROUTINES
*)
(*****)

(*-40-*)
PROCEDURE PEXPR;                                FORWARD;

(*-41-*)
PROCEDURE PPRIMARY;                            FORWARD;

(*****)

(*-26-----*)
*) GENFNC GENERATES THE CODE FOR SHORT, SIMPLE CALCULATOR *)
*) ARITHMETIC FUNCTIONS ("QUICK" FUNCTIONS). *)
*)-----*)

PROCEDURE GENFNQ (OPND : SLOTPTR);
VAR   I : 1..FNQLEN;
BEGIN
  SCAN (TCKNUM);
  IF TCKNUM <> OPARENTOK THEN                    (* ' ( ' *)
    PERRCR
  ELSE
    BEGIN
      SCAN (TOKNUM);
      PEXER;
      I := 1;
      WITH CEND@ DO
        REPEAT
          GENKEY (FNQ(.I.));
          I := I + 1;
        UNTIL (I >= FNQLEN) OR (FNQ(.I.) = K_NOP)
      END
    END;
END;                                             (* GENFNQ *)

(*****)

(*-27-----*)
*) CHKFNLLIST SEARCHES THE FNL USE LIST TO DETERMINE IF *)
*) THE FNL HAS EVER BEEN CALLED BEFORE; IF NOT, THEN IT *)
*) MUST BE ADDED TO THE USE LIST. *)
*)-----*)

PROCEDURE CHKFNLLIST (VAR IDSLOT : SLOTPTR);
VAR   LISTPTR, HOLDPTR : SLOTPTR;
      USED : BOOLEAN;
BEGIN
  LISTPTR := FNLLIST;                            (* GET THE LONG FN LIST *)
  IF LISTPTR <> NIL THEN                          (* TRAVERSE THE FNLLIST *)
    REPEAT
      USED := (IDSLOT@.IDENT = LISTPTR@.IDENT);
      LISTPTR := LISTPTR@.FNLLINK;
    UNTIL (USED) OR (LISTPTR = NIL);
    IF LISTPTR = NIL THEN                          (* IF NOT FOUND ON LIST, THEN *)
      BEGIN                                       (* ADD THIS LONG FUNCTION TO THE FNLLIST *)
        HOLDPTR := FNLLIST;
        FNLLIST := IDSLOT;
        IDSLOT@.FNLLINK := HOLDPTR
      END
    END;
END;                                             (* CHKFNLLIST *)

(*****)

```

```

(*-28-----*)
{* GENFNL GENERATES THE SBR CALL TO A FNL BODY AND THEN *}
{* CALLS CN PROCEDURE CHKFNLIST TO SEE IF THE FNL HAS *}
{* BEEN USED BEFORE. *}
-----*)

```

```
PROCEDURE GENFNL (VAR IDSLOT : SLOTPTR);
```

```

BEGIN
  SCAN (TOKNUM);
  IF TOKNUM <> OPARENTOK THEN (* ' ( ' *)
    PERRCR
  ELSE
    BEGIN
      SCAN (TOKNUM);
      PEXER;
      GENKEY (K STO);
      GENKEY (FNLREG);
      GENKEY (K SBR);
      GENKEY (IDSLOT.LBL);
      CHKFNLIST (IDSLOT)
    END
  END;
(* GENFNL *)

```

```

(*****

```

```

(*-29-----*)
{* NEWPARM IS CALLED WHEN A FIRST CALL TO A PARAMETER FN *}
{* IS ENCOUNTERED IN THE WBASIC SOURCE FILE; AT THIS TIME *}
{* NO REGISTERS HAVE BEEN DESIGNATED FOR THE FORMAL FN *}
{* PARMS; THIS PROCEDURE CREATES THE SYM TBL ENTRY AND *}
{* DESIGNATES THE RESPECTIVE REGISTER FOR THE NEW PARM *}
{* ENCOUNTERED IN THE FN CALL; NOTE THAT THE FORMAL PARM *}
{* IDENT NAME IS NOT YET KNOWN AND WILL BE ENTERED ONLY *}
{* AFTER THE FN DEFINITION STATEMENT IS ENCOUNTERED LATER *}
-----*)

```

```
FUNCTION NEWPARM : SICTPTR;
```

```
VAR PARMSICT : SLOTPTR;
```

```

BEGIN
  NEW (PARMSLOT);
  WITH PARMSLOT DO
    BEGIN
      TYP := VARID;
      REGNC := NEWREG;
      WRITELN (NAMEP, ' :5,REGNO:5, ' :5, ' (FN PARAMETER) ');
      SICT := NIL
    END;
  NEWPARM := PARMSLOT
END;
(* NEWPARM *)

```

```

(*****

```

```

(*-30-----*)
(* GENPARM GENERATES THE FORMAL PARAMETER LIST FOR *)
(* PARAMETER FUNCTIONS; IT ALSO GENERATES THE CODE WHICH *)
(* WILL EVALUATE AN ACTUAL PARAMETER AND STORE IT IN THE *)
(* FORMAL PARAMETER STORAGE LOCATION PRIOR TO FN EXECUTE. *)
(*-----*)

```

```

PROCEDURE GENPARM (VAR IDSLOT : SLOTPTR);
VAR PARMETR : SLOTPTR;
BEGIN
  SCAN (TOKNUM);
  IF TOKNUM = OPARENICK THEN
    BEGIN
      PARMETR := IDSLOT@.FNP;
      IF PARMETR = NIL THEN
        BEGIN
          PARMETR := NEWPARM;
          IDSLOT@.FNE := PARMETR
        END;
      REPEAT
        SCAN (TOKNUM);
        PEXER; (* STOP AT EACH ',' AND LAST ')' *)
        GENKEY (K_STC);
        GENKEY (PARMETR@.REGNO);
        IF TOKNUM = CCMMATOK THEN
          BEGIN
            IF PARMETR@.SLOT = NIL THEN
              PARMETR@.SLOT := NEWPARM;
            PARMETR := PARMETR@.SLOT (* NEXT PARAMETER *)
          END
        UNTIL (TOKNUM = CPARENTOK) (* PEXPR WILL FIND ')' *)
              OR (TOKNUM IN TRAILTOKS) (* OR WILL FIND END *)
      END
    END;
END; (* GENPARM *)

```

```

(******)

```

```

(*-31-----*)
(* GENFNP GENERATES CCDE SEQUENCE WHICH CALLS A PARM FNP. *)
(*-----*)

```

```

PROCEDURE GENFNP (VAR IDSLOT : SLOTPTR);
BEGIN
  GENPARM (IDSLOT);
  GENKEY (K_CPAREN);
  GENKEY (K_SBR);
  GENKEY (IDSLOT@.LBI);
  GENKEY (K_CPAREN)
END; (* GENFNP *)

```

```

(*****
*)
*           FUNCTION DEFINITION ROUTINES
*
(*****

```

```

*-32-----*
* PUSHFN PUSHES A FNP SLOT ONTO THE FNP ACTIVATION STACK *
*-----*

```

```

PROCEDURE PUSHFN (VAR FNSLOT : SLOTPTR);
BEGIN
  FNSLOT@.FNPLINK := FNSTACK;
  FNSTACK := FNSLOT;
  FNSTACKCT := FNSTACKCT + 1;
  IF FNSTACKCT > FNSTACKLIM THEN
    BEGIN
      FWARN;
      WRITE (LISTF, '***** SER STACK OVERFLOW...> ');
      WRITEIN (LISTF, FNSTACKLIM:1, ' RETURN ADDRESSES. ');
    END
  END;
(* PUSHFN *)

```

```

(*****
*-33-----*
* PCPFN POPS A FNP SLOT OFF TOP OF FNP ACTIVATION STACK. *
*-----*

```

```

PROCEDURE PCPFN;
VAR  HOLDFTR : SLOTPTR;
BEGIN
  HOLDFTR := FNSTACK;
  FNSTACK := FNSTACK@.FNPLINK;
  HOLDFTR@.FNPLINK := NIL;
  FNSTACKCT := FNSTACKCT - 1;
  IF FNSTACKCT < 0 THEN
    BEGIN
      FWARN;
      WRITE (LISTF, '***** ATTEMPT TO POP RETURN ADDR ');
      WRITEIN (LISTF, 'FROM EMPTY STACK...RESET CT = 0');
      FNSTACKCT := 0;
    END
  END;
(* POPFN *)

```

```

(*****

```

```

(*-34-----*)
(* FILLPARIIDS READS THE FORMAL PARAMETER IDENTIFIER *)
(* NAMES IN THE FNP DEF STATEMENT AND FILLS THE IDENT *)
(* FIELD OF THE RESPECTIVE PARAMETER SLOT (ATTACHED TO *)
(* RESPECTIVE FNP SLOT IN SYM TBL). *)
(*-----*)

```

```
PROCEDURE FILLPARIIDS (VAR FNSLOT : SLOTPTR);
```

```
VAR FNPARM : SLCTPTR;
```

```

BEGIN
  IF FNSLCT@.FNP = NIL THEN (* IN CASE FNP HAS NOT *)
    FNSLCT@.FNP := NEWPARM; (* BEEN CALLED AT ALL *)
    FNPARM := FNSLCT@.FNP;
    SCAN (TCKNUM);
    IF TCKNUM <> IDENTCK THEN
      PERRCR
    ELSE
      BEGIN
        WHILE TOKNUM <> CPARENTOK DO
          BEGIN
            FNEARM@.IDENT := ACCUM;
            SCAN (TOKNUM);
            IF TOKNUM = COMMATOK THEN
              BEGIN
                SCAN (TCKNUM);
                IF FNPARM@.SLOT = NIL THEN (* FNP HAS NOT *)
                  FNSLCT@.SLOT := NEWPARM; (* BEEN CALLED *)
                FNPARM := FNPARM@.SLOT
              END
            END;
          END
        SCAN (TOKNUM) (* SCAN TOKEN AFTER ' ' *)
      END
    END;
  END;
END; (* FILLPARIIDS *)

```

```
(*****)
```

```

(*-35-----*)
(* PFNEND GENERATES CODE FOR THE END OF A FUNCTION BODY *)
(* DEFINITION; IT INCLUDES THE HOUSE-KEEPING REQUIRED TO *)
(* RESET THE SCOPE AND VISIBILITY OF VARIABLE NAMES IN *)
(* THE SYMBOL TABLE THROUGH THE FN ACTIVATION STACK. *)
(*-----*)

```

```
PROCEDURE PFNEND;
```

```

BEGIN
  GENKEY (K_RCL);
  GENKEY (FNSTACK@.FNREGNO);
  GENKEY (K_INVSR);
  POPFN;
  IF NOT (TCKNUM IN TRAILTOKS) THEN (* GUARD AGAINST OVER *)
    SCAN (TCKNUM) (* SCANNING END LINE IF CALLED BY PDEF. *)
  END; (* PFNEND *)

```

```
(*****)
```

```

*-36-----*
* PDEF GENERATES THE CODE WHICH DEFINES THE SCOPE AND *
* VISIBILITY FOR VARIABLE NAMES; RESETS THE VALUE OF THE *
* REGISTER IN WHICH THE FUNCTION VALUE IS RETURNED. *
*-----*

```

```
PROCEDURE PDEF;
```

```
VAR FNSICT : SLCTPTR;
```

```

BEGIN
  SCAN (TCKNUM);
  IF NOT FN_CHK (ACCUM) THEN
    PERRCR
  ELSE
    BEGIN
      FNSICT := IDLOCKUP (ACCUM, ACCINX);
      GENKEY (K_LBL);
      GENKEY (FNSLOT@.LBL);
      GENKEY (K_ZERO); (* MUST ZERO THE VALUE OF THE *)
      GENKEY (K_STO); (* REGISTER IN WHICH THE FN *)
      GENKEY (FNSLOT@.FNREGNO); (* VALUE IS RETURNED. *)
      PUSHFN (FNSLOT);
      SCAN (TOKNUM);
      IF TCKNUM = OPARENTOK THEN (* LOOKING FOR PARMS *)
        FILIPARMIDS (FNSLOT);
      IF TCKNUM = EQUALTOK THEN (* LOOK FOR ONE LINE FN *)
        BEGIN
          SCAN (TOKNUM);
          PEXPR;
          PFNEND (* GENERATE THE RETURN FROM ONE LINE FN *)
        END;
      CIRCSELINE
    END
  END;
END; (* PDEF *)

```

```

(*****

```

```

*-37-----*
* GETFNLS IS CALLED AFTER ALL OTHER CODE HAS BEEN *
* GENERATED; IT GENERATES THE CODE FOR THE BODIES OF ALL *
* BUILT-IN LONG FUNCTIONS WHICH HAVE BEEN CALLED AT *
* LEAST CNCE AND ARE, THUS, ON THE FNLLIST. *
*-----*

```

```
PROCEDURE GETFNLS;
```

```

VAR LISTPTR : SLOTPTR;
    I : 1..FNLEN + 1;

```

```

BEGIN
  LISTPTR := FNLLIST; (* GET LONG FN LIST *)
  WHILE LISTPTR <> NIL DO
    BEGIN (* INSERT CODE FOR NEXT LONG FN ON FNLLIST *)
      GENKEY (K_LBL);
      GENKEY (LISTPTR@.LBL);
      I := 1;
      WHILE (I <= FNLEN) AND
        (LISTPTR@.FNL(.I.) <> K_NOP) DO
        BEGIN
          GENKEY (LISTPTR@.FNL(.I.));
          I := I + 1
        END;
      GENKEY (K_INVSER);
      LISTPTR := LISTPTR@.FNLLINK (* NEXT LONG FN ON LIST *)
    END
  END;
END; (* GETFNLS *)

```

```

(*****
*)
*)      EXPRESSION GENERATOR ROUTINES
*)
(*****

```

```

(*-38-----*)
(* GENID GENERATES CCCE FOR A VARIABLE OR FUNCTION IDENT. *)
(*-----*)

```

PROCEDURE GENID;

VAR CFNE : SLOTPTR;

BEGIN

OPND := IDLOOKUP (ACCUM, ACCINX);

CASE OPND@.TYP OF

VARIC : BEGIN

IF OPND@.REGNO = -314 THEN (* REGNO FOR PI *)

GENKEY (K_PI)

ELSE

BEGIN

GENKEY (K_RCI);

GENKEY (OPND@.REGNO)

END

END;

FNOID : GENFNO (CFNE);

FNLIC : GENFNL (CFNE);

FNPID : GENFNP (CFNE);

END (* CASE *)

END;

(* GENID *)

```

(*****

```

```

(*-39-----*)
(* GENNUM GENERATES II-59 EQUIVALENT CODE FOR A LITERAL *)
(* NUMERIC (BOTH INTEGER AND REAL). *)
(*-----*)

```

PROCEDURE GENNUM;

VAR I, DECPLOC, ESIGNLOC : INTEGER;

BEGIN

DECPLOC := LCCUNT + 1;

FOR I := 1 TO LCOUNT DO

GENKEY (TRANSDIGIT (ACCUM(.I.)));

IF RCCUNT > 0 THEN

BEGIN

GENKEY (K_DECPT);

FOR I := (DECPLOC + 1) TO (DECPLOC + RCOUNT) DO

GENKEY (TRANSDIGIT (ACCUM(.I.)))

END;

IF ECCUNT > 0 THEN

BEGIN

ESIGNLOC := LCCUNT + 1 + RCOUNT + 2;

IF ACCUM(.DECPLOC.) <> PERIOD THEN

ESIGNLOC := ESIGNLOC - 1;

GENKEY (K_EE);

IF ACCUM(.ESIGNLOC.) = '-' THEN

GENKEY (K_NEG);

FOR I := (ESIGNLOC + 1) TO (ESIGNLOC + ECOUNT) DO

GENKEY (TRANSDIGIT (ACCUM(.I.)))

END

END;

(* GENNUM *)

```

(*****

```

```
(*--40-----*)
(* PEXPR PARSSES AND GENERATES CODE FOR EXPRESSIONS. *)
(*-----*)
```

```
PROCEDURE PEXPR; (* FORWARD DECLARATION WITH *)
(* FUNCTION CALL ROUTINES *)
```

```
BEGIN
  GENKEY (K_OPAREN);
  PPRIMARY:-
  WHILE TCKNUM IN BINOPTCKS DO
    BEGIN
      CASE TCKNUM OF
        FIUSTOK : GENKEY (K_ADDOP); (* '+' *)
        MINUSTOK : GENKEY (K_SUBCP); (* '-' *)
        MULTOK : GENKEY (K_MULTOP); (* '*' *)
        DIVTOK : GENKEY (K_DIVOP); (* '/' *)
        EXP1OK : GENKEY (K_POWR); (* '**' *)
      END; (* CASE *)
    IF NOT (TCKNUM IN TRAILTOKS) THEN BEGIN
      SCAN (TCKNUM);
      PPRIMARY
    END;
  END;
  GENKEY (K_CPAREN)
END; (* PEXPR *)
```

```
(*****)
```

```
(*--41-----*)
(* PPRIMARY PARSSES AND GENERATES CODE FOR A PRIMARY ITEM *)
(* EXPECTED AS PART OF EXPRESSIONS. *)
(*-----*)
```

```
PROCEDURE PPRIMARY; (* FORWARD DECLARATION WITH *)
(* FUNCTION CALL ROUTINES *)
```

```
BEGIN
  CASE TCKNUM OF
    FIUSTCK : BEGIN (* '+' *)
      SCAN (TOKNUM);
      PPRIMARY
    END;
    MINUSTCK : BEGIN (* '-' *)
      SCAN (TOKNUM);
      PPRIMARY;
      GENKEY (K_NEG)
    END;
    OPARENTCK : BEGIN (* '(' *)
      SCAN (TOKNUM);
      PEXPR;
      SCAN (TOKNUM)
    END;
    IDENTCK : BEGIN
      GENID;
      IF (TOKNUM=IDENTOK) OR (TOKNUM=CPARENTOK)
      THEN SCAN (TOKNUM)
    END;
    NUMBERTCK : BEGIN
      GENNUM;
      SCAN (TOKNUM)
    END
  END (* CASE *)
END; (* PPRIMARY *)
```

```
(*****)
```

```
(* -42-----*)
(* PCCNDITICN PARSES AND GENERATES CODE FOR BOOLEAN EXPRS *)
(*-----*)
```

```
PROCEDURE FCCNDITION;
```

```
VAR REICF : TOKENRNG;
    INVERT : BOOLEAN;
```

```
BEGIN
```

```
  INVERT := FALSE;
```

```
  SCAN (TCKNUM);
```

```
(* SCAN FOR "NOT" *)
```

```
  IF TOKNUM = NOTTOK THEN
```

```
    BEGIN
```

```
      INVERT := TRUE;
```

```
      SCAN (TOKNUM)
```

```
    END;
```

```
  PEXFR;
```

```
  GENKEY (K_X T);
```

```
  IF INVERT THEN
```

```
    CASE TOKNUM OF
```

```
      EQUALTCK : TOKNUM := NOTECTOK;
```

```
      NOTECTOK : TOKNUM := EQUALTOK;
```

```
      GTTCK : TOKNUM := ITTECTOK;
```

```
      GTECTCK : TOKNUM := LTTOK;
```

```
      ITTCK : TOKNUM := GTECTOK;
```

```
      LTECTCK : TOKNUM := GTTOK;
```

```
    END; (* CASE *)
```

```
  REICF := TCKNUM;
```

```
(* BEGIN NEXT EXPR *)
```

```
  SCAN (TCKNUM);
```

```
  PEXFR;
```

```
  CASE REICF OF
```

```
    EQUALTOK : BEGIN
```

```
      GENKEY (K_INV);
```

```
      GENKEY (K_IFXEQT)
```

```
    END;
```

```
    NTECTIOK : GENKEY (K_IFXEQT);
```

```
    GTTCK : GENKEY (K_IFXGET);
```

```
    GTECTCK : BEGIN
```

```
      GENKEY (K_X T);
```

```
      GENKEY (K_INV);
```

```
      GENKEY (K_IFXGET)
```

```
    END;
```

```
    ITTECTCK : BEGIN
```

```
      GENKEY (K_INV);
```

```
      GENKEY (K_IFXGET)
```

```
    END;
```

```
    ITTCK : BEGIN
```

```
      GENKEY (K_X T);
```

```
      GENKEY (K_IFXGET)
```

```
    END
```

```
  END (* CASE *)
```

```
(* PCONDITION *)
```

```
END;
```

```

(*****
{*
{*          LOOPING ROUTINES
{*
(*****

```

```

*-43-----*
{* PUSHCCDE PUSHES THE RCODE DATA NODE ONTO THE LOOP/IF
{* STACK DESIGNATED BY STACK.
{*-----*

```

```

PROCEDURE PUSHCODE (RCODE : CODEPTR; VAR STACK : CODEPTR);
BEGIN
  RCODE@.SECP := STACK;
  STACK := RCODE
END;                                     (* PUSHCODE *)

```

```

(*****
*-44-----*
{* POPCODE PCPS AND RETURNS THE CODE DATA NODE ON THE TOP
{* OF THE LCCP/IF STACK DESIGNATED.
{*-----*

```

```

FUNCTION PCPCODE (VAR STACK : CODEPTR) : CODEPTR;
BEGIN
  IF STACK = NIL THEN
    BEGIN
      WRITE (LISTF, '***** INCORRECT NESTING OF CONTRCL');
      WRITEIN (LISTF, ' STATEMENTS (IF, FOR, CR LOOP).');
      PERROR;
      POPCODE := NIL
    END
  ELSE
    BEGIN
      POPCODE := STACK;
      STACK := STACK@.SEQP
    END
END;                                     (* POPCODE *)

```

```

(*****
*-45-----*
{* SETFWDJMP IS USED TO SET THE JUMP POINTER (JMPP) OF
{* THE CURRENT CODE DATA NODE POINTING TO THE MOST RECENT
{* CODE DATA NODE ON THE DESIGNATED LOOP/IF STACK; THE
{* POTENTIAL ABSOLUTE ADDRESS SPACE IS GENERATED WITH THE
{* ASSUMPTION THAT THE CODE DATA NODE IN THE STACK TO
{* WHICH THE FIRST ADDRESS SPACE NODE IS POINTING WILL
{* LATER BE POPPED AND INSERTED INTO THE CODE AT THE
{* APPROPRIATE POSITION.
{*-----*

```

```

PROCEDURE SETFWDJMP (STACK : CODEPTR);
BEGIN
  CPCUR@.JMPP := STACK; (* SET JMPP TO NODE JUST PUSHED *)
  STACK@.ADDR := 0;     (* MARK THE TERMINAL NODE OF JUMP *)
  GENKEY (-2);         (* GEN SPACE FOR ABSOLUTE ADDRESS *)
  GENKEY (-2)
END;                                     (* SETFWDJMP *)

```

```

(*****

```

```

(*-46-----*)
(* SETBAKJUMP IS SIMILAR TO SETFWDJMP EXCEPT IN THIS CASE *)
(* THE FIRST NODE OF A POTENTIAL ADDRESS SPACE PAIR IS *)
(* PUSHED ONTO THE DESIGNATED LOOP/IF STACK AFTER ITS *)
(* JMPF HAS BEEN SET TO A CCDE DATA NODE INSERTED AS AN *)
(* ANCHOR FOR THIS BACK JUMP; THE POTENTIAL ADDRESS SPACE *)
(* NODE WILL LATER BE POPPED AND INSERTED (ALONG WITH ITS *)
(* THE INSERTION OF ANOTHER NODE TO COMPOSE AN ADDR PAIR) *)
-----*)

```

```
PROCEDURE SETBAKJMP (VAR STACK : CODEPTR);
```

```
VAR JCODE : CODEPTR;
```

```
BEGIN
```

```
  JCCDE := NEWCODE (-2);
```

```
  JCODE@.JMPP := CPCUR;
```

```
  CPCUR@.ADIR := 0;
```

```
  GENKEY (K_NOP);
```

```
  PUSHCODE (JCODE, STACK)
```

```
END;
```

```
(* SETBAKJMP *)
```

```
*****
```

```

(*-47-----*)
(* PLOOP GENERATES CCDE FOR THE LOOP COMMAND; *)
(* IT SETS UP THE START OF A LOOP CONSTRUCT BY GENERATING *)
(* AN ANCHOR NODE FOR THE BACK JUMP IN THE LOOP. *)
-----*)

```

```
PROCEDURE PLOOP;
```

```
BEGIN
```

```
  SETBAKJMP (LOOPSTACK);
```

```
  PUSHCODE (NEWCODE (K_NOP), ENDLLOOPSTACK);
```

```
  SCAN (TCKNUM);
```

```
  CLOSELINE
```

```
END;
```

```
(* PLOOP *)
```

```
*****
```

```

(*-48-----*)
(* PWHILE GENERATES CCDE FOR THE WHILE COMMAND; *)
(* IT IS SIMILAR TO PLOOP EXCEPT IT INSERTS CODE TO *)
(* EVALUATE A BOOLEAN EXPRESSION (CONDITION). *)
-----*)

```

```
PROCEDURE PWHILE;
```

```
BEGIN
```

```
  SETBAKJMP (LOOPSTACK);
```

```
  PCONDITION;
```

```
  PUSHCODE (NEWCODE (K_NOP), ENDLLOOPSTACK);
```

```
  SETFWDJMP (ENDLOOPSTACK)
```

```
END;
```

```
(* PWHILE *)
```

```
*****
```

```

(*-49-----*)
* PENDLOCF POPS AND INSERTS CODE WHICH HAD BEEN STACKED *
* EARLIER AS A RESULT OF THE START OF A LOOP CONSTRUCT. *
(*-----*)

```

```
PROCEDURE PENDLOCF;
```

```
VAR JCODE : CODEPTR;
```

```
BEGIN
```

```

  IF TCKNUM = ENDLOOPSTOK THEN
    GENKEY (K_GTO);
    JCODE := POPCODE (LOPSTACK);
    PUTKEY (JCODE);
    GENKEY (-2);
    JCODE := POPCODE (ENDLOOPSTACK);
    PUTKEY (JCODE);
    CLCSELINE

```

```
END; (* PENDLOCF *)
```

```
(*****)
```

```

(*-50-----*)
* PUNTIL GENERATES CODE TO EVALUATE A BOOLEAN EXPRESSION *
* AND CALLS PENDLOCF TO CLCSE OUT THE LOOP. *
(*-----*)

```

```
PROCEDURE PUNTIL;
```

```
BEGIN
```

```

  PCCNDITION;
  PENDLOCF

```

```
END; (* PUNTIL *)
```

```
(*****)
```

```

(*-51-----*)
* PNEXT GENERATES CODE FOR THE NEXT COMMAND. *
* THIS ROUTINE IS WEAK IN SYNTAX ERROR CHECKING. *
(*-----*)

```

```
PROCEDURE PNEXT;
```

```

VAR ISLCT : SLOTPTR;
    JCODE : CODEPTR;

```

```
BEGIN
```

```

  SCAN (TCKNUM);
  ISLCT := IDLOOKUP (ACCUM, ACCINX);
  IF ISLCT@.AUXREG2 = -1 THEN
    GENKEY (K_1)
  ELSE
    BEGIN
      GENKEY (K_RCL);
      GENKEY (ISLOT@.AUXREG2)
    END;
  GENKEY (K_SUM);
  GENKEY (ISLOT@.REGNC);
  GENKEY (K_GTO);
  JCODE := POPCODE (FORSTACK);
  PUTKEY (JCODE);
  GENKEY (-2);
  JCODE := POPCODE (NEXTSTACK);
  PUTKEY (JCODE);
  CLCSELINE

```

```
END; (* PNEXT *)
```

```
(*****)
```

```

(*-52-----*)
(* PFOR GENERATES CODE FOR THE FCR COMMAND. THIS ROUTINE *)
(* (AND THE PNEXT ROUTINE) IS WEAK IN SYNTAX ERROR CHECK- *)
(* ING. THERE ARE MANY PLACES WHERE SIMPLE CHECKS FOR *)
(* CORRECT SYNTAX COULD HAVE BEEN PERFORMED BUT WERE NOT *)
(* BECAUSE OF COMPLEXITY SUCH CHECKS WOULD HAVE INDUCED. *)
*-----*

```

```
PROCEDURE PFOR;
```

```
VAR ISICT : SLOTPTR;
```

```

BEGIN
  SCAN (TCKNUM);
  ISICT := IDLOOKUP (ACCUM, ACCINX);
  ISICT@.AUXREG1 := NEWREG;
  SCAN (TCKNUM);
  SCAN (TCKNUM);
  PEXER;
  GENKEY (K STO);
  GENKEY (ISLOT@.REGNO);
  SCAN (TCKNUM);
  PEXER;
  GENKEY (K STO);
  GENKEY (ISLOT@.AUXREG1);
  IF TCKNUM = STEPTOK THEN
    BEGIN
      SCAN (TOKNUM);
      PEXER;
      GENKEY (K STO);
      ISICT@.AUXREG2 := NEWREG;
      GENKEY (ISLOT@.AUXREG2)
    END;
  PUSHCODE (NEWCODE (-2), FCRSTACK);
  FORSTACK@.JMPP := CPCUR;
  GENKEY (K RCL);
  GENKEY (ISLOT@.REGNO);
  GENKEY (K X T);
  GENKEY (K RCL);
  GENKEY (ISLOT@.AUXREG1);
  GENKEY (K INV);
  GENKEY (K IFXGET);
  PUSHCODE (NEWCODE (K NOP), NEXTSTACK);
  CPCUR@.JMPP := NEXSTACK;
  GENKEY (-2);
  GENKEY (-2);
  CLCSELINE
END;

```

```
(* PFOR *)
```

```

(*****
*
*           IF-BRANCHING ROUTINES
*
*****)

```

```

(*-53-----*)
* QUITERROR IS CALLED WHENEVER A QUIT STATEMENT IS
* ENCOUNTERED WHILE NOT WITHIN THE SCOPE OF A LOOP.
*-----*)

```

PROCEDURE QUITERROR;

```

BEGIN
  WRITE (LISTF, '***** ATTEMPT TO "QUIT" WHILE NOT ');
  WRITELN (LISTF, 'INSIDE A LOOP. ');
  PERROR
END; (* QUITERRCR *)

```

(*****)

```

(*-54-----*)
* PQUIT GENERATES POTENTIAL ADDRESS SPACE WHOSE JMPP
* POINTS TO THE MOST CURRENT CODE NODE ON THE ENDLOOP
* STACK; THUS, CONTROL WILL LEAVE THE MOST CURRENTLY
* EXECUTING LOOP DURING TI-59 EXECUTION. NOTE THAT THIS
* IMPLEMENTATION WILL NOT ALLOW LINE# TO FOLLOW 'QUIT'
*-----*)

```

PROCEDURE PQUIT;

```

BEGIN
  IF ENDLCOFSTACK = NIL THEN
    QUITERRCR
  ELSE
    BEGIN
      GENKEY (K_GTO);
      SETFWLJMP (ENDLCOFSTACK);
      SCAN (TOKNUM);
      IF TCKNUM = NUMEERTOK THEN
        BEGIN
          PSUBERROR;
          WRITE (LISTF, '***** "QUIT" DOES NOT ACCEPT ');
          WRITELN (LISTF, 'LINE NUMBERS THIS IMPLEMENT. ');
        END
      ELSE
        CLOSeline
    END
  END; (* PQUIT *)

```

(*****)

```

(*-56-*)
PROCEDURE PTHENLINE; FORWARD;

```

(*****)

```

(*-55-----*)
(* PTHENELSE DETERMINES WHETHER THE ELSE BRANCH OF AN *)
(* IF-THEN-ELSE IS LINE-ORIENTED (LINE#) OR LOOP-ORIENTED *)
(* ('QUIT'); APPROPRIATE ROUTINE IS CALLED TO SET JUMP. *)
(*-----*)

```

PROCEDURE PTHENELSE;

```

BEGIN
  SCAN (TCKNUM);
  IF TCKNUM = NUMBERTOK THEN
    BEGIN
      GENKEY (K_GTO);
      PTEENLINE
    END
  ELSE IF TCKNUM = QUITTOK THEN
    PQUIT
  ELSE
    BEGIN
      WRITE (LISTF, '***** "IF-THEN-ELSE" LIMITED TO ');
      WRITEIN (LISTF, '"QUIT" OR LINE NUMBERS. ');
      PERROR
    END
END;
(* PTHENELSE *)

```

(*****)

```

(*-56-----*)
(* PTHENLINE SETS THE LINE-ORIENTED JUMPS FOR THE IF-THEN *)
(* OR IF-THEN-ELSE STATEMENTS. *)
(*-----*)

```

PROCEDURE PTHENLINE;

```

BEGIN
  SETJMPFEXT (XNUMBER (ACCUM, ACCINX));
  GENKEY (-2);
  GENKEY (-2);
  SCAN (TCKNUM);
  IF TOKNUM = ELSETOK THEN
    PTHENELSE
  ELSE
    CLOSELINE
END;
(* PTHENLINE *)

```

(*****)

```

(*-57-----*)
(* PTHENQUIT SETS THE LOOP-ORIENTED JUMPS FOR IF-THEN OR *)
(* IF-THEN-ELSE STATEMENTS. *)
(*-----*)

```

PROCEDURE PTHENQUIT;

```

BEGIN
  IF ENDICOPSTACK = NIL THEN
    QUITERROR
  ELSE
    BEGIN
      SETFWLJMP (ENDICOPSTACK);
      SCAN (TOKNUM);
      IF TOKNUM = ELSETOK THEN
        PTHENELSE
      ELSE
        CLOSELINE
    END
END;
(* PTHENQUIT *)

```

(*****)

```

(*-58-----*)
{* PIF DETERMINES THE TYPE OF 'IF' STATEMENT; IT WILL *}
{* CALL THE REQUIRED SET ROUTINES FOR UNSTRUCTURED JUMPS *}
{* (LINE OR LOOP ORIENTED) OR PERFORM THE SET UP ITSELF *}
{* FOR STRUCTURED JUMPS. *}
(*-----*)

```

PROCEDURE PIF;

```

BEGIN
  PCONDITION;
  IF TOKNUM = THENOK THEN
    BEGIN
      SCAN (TOKNUM);
      IF TOKNUM = NUMBERTOK THEN
        ETHENLINE
      ELSE IF TOKNUM = QUITOK THEN
        ETHENQUIT
      ELSE
        PERROR
    END
  ELSE
    BEGIN
      PUSHCODE (NEWCCDE (K NOP), ENDIFSTACK);
      CPCUR@.BAKP := ENDIFSTACK;
      ENDIFSTACK@.BAKP := CECUR;
      PUSHCCDE (NEWCODE (K NOP), IFSTACK);
      SETIFWJMP (IFSTACK);
      CXCSELINE
    END
  END;

```

(* PIF *)

(*****)

```

(*-59-----*)
{* ELSE ADJUST PERFORMS HOUSE-KEEPING ON THE VARIOUS 'IF' *}
{* STACKS DEPENDENT UPON THE FORM OF THE STRUCTURED 'IF' *}
{* STATEMENT ENCOUNTERED; IF-ENDIF REQUIRES A DIFFERENT *}
{* SEQUENCE OF PUSH/POP STACK THAN DOES IF-ELSE-ENDIF OR *}
{* IF-ELSEIF-ELSE-ENLIF. *}
(*-----*)

```

PROCEDURE ELSE_ADJUST;

```

BEGIN
  WITH ENDIFSTACK@ DC
    BEGIN
      IF BAKP <> NIL THEN
        BEGIN
          BAKP@.BAKP := NIL;
          BAKP@.JMPP@.BAKP := NIL;
          BAKP := NIL
        END
      END
    END
  END;

```

(* ELSE_ADJUST *)

(*****)

```
(*-60-----*)
(* PELSEIF PERFORMS A SEQUENCE OF STACK MANIPULATIONS IN *)
(* ORDER TO GENERATE THE ADDRESS SPACES AND JUMPS WHICH *)
(* IMPLEMENT THE ELSEIF CONSTRUCT. *)
(*-----*)
```

```
PROCEDURE PELSEIF;
```

```
VAR JCCODE : CODEPTR;
```

```
BEGIN
  ELSE ADJUST;
  GENKEY (K_GTO);
  SETFWDJMP (ENDIFSTACK);
  JCCODE := POPCODE (IFSTACK);
  PUTKEY (JCCODE);
  PCCNDITION;
  PUSHCODE (NEWCCODE (K_NOP) , IFSTACK);
  SETFWDJMP (IFSTACK);
  CLOSELINE
```

```
END; (* PELSEIF *)
```

```
(*****)
```

```
(*-61-----*)
(* PELSE IS SIMILAR TO ELSEIF EXCEPT IT DOES NOT PARSE/ *)
(* GENERATE CODE TO EVALUATE A BOOLEAN EXPRESSION. *)
(*-----*)
```

```
PROCEDURE PELSE;
```

```
VAR JCCODE : CODEPTR;
```

```
BEGIN
  ELSE ADJUST;
  GENKEY (K_GTO);
  SETFWDJMP (ENDIFSTACK);
  JCCODE := POPCODE (IFSTACK);
  PUTKEY (JCCODE);
  SCAN (TCKNUM);
  CLOSELINE
```

```
END; (* PELSE *)
```

```
(*****)
```

```

(*-62-----*)
(* PENDIF CLOSES UP THE SCOPE OF A STRUCTURED 'IF' *)
(* CCNSTRUCT BY POPPING THE APPROPRIATE STACKS AND *)
(* INSERTING AND DISCARDING CODE WHICH HAD BEEN STACKED; *)
(* DISCARDING/INSERTING IS DEPENDENT UPON THE PARTICULAR *)
(* TYPE OF 'IF' CONSTRUCT (IF-ENDIF OR IF-ELSE-ENDIF). *)
(*-----*)

```

```
PROCEDURE PENDIF;
```

```
VAR DUMPC, JCODE : CODEPTR;
```

```

BEGIN
  WITH ENDIFSTACK@ DO
    BEGIN
      IF EAKP <> NIL THEN
        BEGIN
          (* NC ELSE/ELSEIF HAS BEEN SEEN *)
          EAKP@.EAKP := NIL; (* NULLIFY POINTERS *)
          BAKP := NIL;
          DUMPC := PCFCODE (ENDIFSTACK); (* CLEAR STACK *)
          DUMPC@.ADDR := -1;
          JCODE := PCFCODE (IFSTACK); (* INSERT ENDIF *)
          PUTKEY (JCODE)
        END
      ELSE
        (* ELSE/ELSEIF HAS BEEN SEEN *)
        BEGIN
          JCODE := PCFCODE (ENDIFSTACK); (* INSERT ENDIF *)
          JCODE@.ADDR := 0; (* MARK TERMINAL NODE OF JUMP *)
          PUTKEY (JCODE) (* ELSE ADJUST HAS ALREADY *)
        END
      END
    END
  END;
  SCAN (TCKNUM);
  CLCSELINE
END; (* PENDIF *)

```

```

(*****
*
*           I/C COMMAND ROUTINES
*
*****)

```

```

(**-84-**)
PROCEDURE WRITLN (VAR WFILE, MSGFILE : TEXT;
                  MSG_NO : INTEGER);

```

```

(*****

```

```

*-63-----*)
* PDATA GENERATES SINGLE NCP TO PROTECT FROM LINE-
* ORIENTED JUMP REFERENCES; THIS COMMAND IS INTENDED FOR
* USE AT THE START OF A PROGRAM SINCE ITS USE WITHIN
* LCOPS, SER'S, OR FUNCTIONS WOULD RENDER THE DATA TO
* READ MAPPING MEANINGLESS. THIS ROUTINE READS THE DATA
* VALUES FOUND AS ITS PARAMETERS, COUNTS THEM, AND
* STORES THEM IN AN ARRAY OF RECORDS WHICH IS ACCESSED
* BY SUBSEQUENT READ COMMANDS.
*-----*)

```

```

PROCEDURE PLATA;

```

```

VAR DATASIGN : CHAR;

```

```

BEGIN
  GENKEY (K NOP);
  SCAN (TCKNUM);
  WHILE (TOKNUM IN SIGNTOKS) OR (TOKNUM = NUMBERTOK) DO
    BEGIN
      DATASIGN := BLANK;
      IF TOKNUM IN SIGNTOKS THEN
        BEGIN
          IF TOKNUM = MINUSTOK THEN
            DATASIGN := '-';
          SCAN (TOKNUM)
        END;
      IF DATAIX = REGEASE + 1 THEN
        BEGIN
          PWARN;
          WRITE (LISTF, '***** EXCEEDED DATASTORE ');
          WRITE (LISTF, 'CAP = ', REGBASE : 1);
          WRITELN (LISTF, '...RESET DATA INDEX TO 1. ');
          DATAIX := 1
        END;
      DATAIX (.DATAIX.) .NUMB := ACCUM;
      DATAIX (.DATAIX.) .SIGN := DATASIGN;
      DATAIX := DATAIX + 1;
      SCAN (TOKNUM);
      IF TOKNUM = COMMATOK THEN
        SCAN (TOKNUM)
    END;
  CLCSELINE
END;

```

```

(* PDATA *)

```

```

(*****

```

```

(*-64-----*)
(* PREAD ONLY GENERATES A MCP INSTRUCTION TO ALLOW FOR *)
(* LINE-ORIENTED JUMP REFERENCES; OTHERWISE, THIS CMMAND *)
(* WRITES THE READF FILE WHICH INDICATES DATA VALUES FOR *)
(* RESPECTIVE REGISTERS AND THEIR WBASIC VARIABLE NAMES; *)
(* THE READF FILE IS USED TO INPUT DATA PRIOR TO PROGRAM *)
(* EXECUTION ON THE II-59, THUS, SAVING PROGRAM STEPS. *)
(* THE CCNSTRUCT IS INTENDED FOR USE AT THE START OF A *)
(* PROGRAM. IF NESTED WITHIN THE PROGRAM, LOOPS, SBR'S, *)
(* AND FN'S WOULD RENDER THE DATA/READ MAP MEANINGLESS. *)
(*-----*)

```

```

PROCEDURE PREAD;
VAR  IDSICT : SLCTPTR;
BEGIN
  GENKEY (K NOP);
  IF FIRSTREAD THEN
    BEGIN
      REWRITE (READF, 'NAME=READF.WBASIC.A');
      WRITLN (READF, MSGF, 9);
      FIRSTREAD := FALSE
    END;
  IF NOT INDEXERROR THEN
    BEGIN
      SCAN (TOKNUM);
      WRITELN (READF);
      WHILE TOKNUM = IDENTOK DO
        BEGIN
          IDSLOT := IDLOOKUP (ACCUM, ACCINX);
          WITH IDSLOT DO
            BEGIN
              IF READIX >= DATAIX THEN
                BEGIN
                  PWARN;
                  WRITE (LISTF, '***** READ PAST DATA ');
                  WRITE (LISTF, 'INDEX...IGNORING ');
                  WRITELN (LISTF, 'SUBSEQUENT READ/DATA. ');
                  WRITELN (READF);
                  WRITE (READF, '***** READ PAST DATA ');
                  WRITE (READF, 'INDEX...SUBSEQUENT ');
                  WRITELN (READF, 'READ/DATA IGNORED. ');
                  INDEXERROR := TRUE;
                  PRECOVER
                END
              ELSE
                BEGIN
                  WRITE (READF, ' ':5);
                  WRITE (READF, DATALIST (.READIX.) .SIGN);
                  WRITE (READF, DATALIST (.READIX.) .NUMB);
                  WRITE (READF, ' ':2);
                  ZENCPAD (READF, REGNO, 2);
                  WRITELN (READF, ' ':3, IDENT);
                  READIX := READIX + 1;
                  SCAN (TOKNUM);
                  IF TOKNUM = COMMATOK THEN
                    SCAN (TCKNUM)
                END
            END
          END
        END
      END
    END
  ELSE
    PRECOVER;
  CISCSELINE
END;

```

(* PREAD *)

(*****)

```

(*-65-----*)
(* PRESTORE GENERATES A SINGLE NOP TO PROTECT FROM LINE- *)
(* ORIENTED JUMP REFERENCES. IN THIS IMPLEMENTATION, *)
(* THIS CCNSTRUCT IS NOT OF GREAT VALUE SINCE DATA/READ *)
(* STATEMENTS ARE SUGGESTED FOR USE AT THE START OF A *)
(* PROGRAM ONLY; THIS ROUTINE RESETS THE READ INDEX TO *)
(* ITS INITIAL VALUE (=1). *)
(*-----*)

```

PROCEDURE PRESTORE;

```

BEGIN
  GENKEY (K_NOP);
  READIX := -1;
  SCAN (TCKNUM);
  CLCSELINE
END;
(* PRESTORE *)

```

(*****)

```

(*-66-----*)
(* PINPUT PARSES A LIMITED FORM OF THE WBASIC "INPUT" *)
(* STATEMENT; THE LIST OF INPUT PARAMETERS MAY CONSIST OF *)
(* VARIABLE NAMES ONLY. *)
(*-----*)

```

PROCEDURE PINPUT;

```

VAR  TENDIG : -1..9;
     INPVAR : SLCTPTR;

```

```

BEGIN
  SCAN (TCKNUM);
  GENKEY (K_CE);
  TENDIG := -1; (* FLAG CHECKS IF INPUT VARS ARE LISTED *)
  WHILE TCKNUM = IDENTCK DO
    BEGIN
      INPVAR := IDLOCKUP (ACCUM, ACCINX);
      TENDIG := INPVAR@.REGNO DIV 10;
      GENKEY (TENDIG); (* REG IN WHICH INPUT TO BE STORED *)
      GENKEY (INPVAR@.REGNO - (TENDIG * 10));
      GENKEY (K_INT); (* CLOSES DISPLAY REG *)
      GENKEY (K_RS);
      GENKEY (K_STO);
      GENKEY (INPVAR@.REGNO);
      SCAN (TCKNUM);
      IF TOKNUM = COMMATOK THEN (* PARAMETERS SEPARATED *)
        (* BY COMMAS OR BLANKS. *)
      END;
      IF TENDIG = -1 THEN (* GENERATES A R/S IF "INPUT" *)
        (* IS USED WITHOUT A VAR LIST *)
      GENKEY (K_RS);
      CLCSELINE
    END;
  END;
(* PINPUT *)

```

(*****)

```

(*-67-----*)
{* PPRINT PARSES A LIMITED FORM OF THE WBASIC "PRINT" *}
{* STATEMENT; IT ALLCWS EXPRESSIONS, VARIABLE NAMES, AND *}
{* LITERAL NUMERICS IN THE LIST OF PARAMETERS. *}
(*-----*)

```

```
PROCEDURE PPRINT;
```

```

BEGIN
  SCAN (TCKNUM);
  WHILE TCKNUM IN BEGIN_EXPRTOKS DO
    BEGIN
      PEXPR;
      IF PC100 THEN (* WITH PC100 *)
        GENKEY (K_PRT)
      ELSE (* WITHOUT PC100 *)
        BEGIN
          GENKEY (K_PAUSE);
          GENKEY (K_RS)
        END;
      IF TOKNUM = COMMATOK THEN (* CAN SEPARATE ITEMS BY *)
        SCAN (TOKNUM) (* COMMAS OR BLANKS. *)
      END;
      IF PC100 THEN (* WITH PC100 *)
        GENKEY (K_ADV);
      CLOSELINE
    END;
  END; (* PPRINT *)

```



```

(*-69-----*)
{* PNOLET PARSES AND GENERATES CODE FOR AN ASSIGNMENT *}
{* STATEMENT WHICH DOES NOT BEGIN WITH THE 'LET' COMMAND. *}
(*-----*)

```

```
PROCEDURE PNOLET;
```

```
VAR RESULT : SLOTPTR;
```

```

BEGIN
  RESULT := IDLOOKUP (ACCUM, ACCINX);
  SCAN (TCKNUM);
  IF TCKNUM <> EQUALTOK THEN (* '=' *)
    PERROR
  ELSE
    BEGIN
      SCAN (TOKNUM);
      PEXER;
      GENKEY (K STO);
      IF RESULT@.TYP = VARID THEN
        GENKEY (RESULT@.REGNO)
      ELSE IF RESULT@.TYP = FNPID THEN
        GENKEY (RESULT@.FNREGNO)
      ELSE
        PERROR
    END
END; (* PNOLET *)

```

```
(*****)
```

```

(*-70-----*)
{* PLET PARSES AND GENERATES CODE FOR A 'LET' STATEMENT. *}
(*-----*)

```

```
PROCEDURE PLET;
```

```

BEGIN
  SCAN (TCKNUM);
  PNOLET
END; (* PLET *)

```

```
(*****)
```

```

(*-71-----*)
{* PREM GUARDS AGAINST USE OF A GOTO DIRECTED TO A REM BY *}
{* CAUSING GENERATION OF A NODE (LOADED W/ A NOP INSTRU) *}
{* WHICH CAN BE REFERENCED BY A JMPP POINTER; THE SCANNER *}
{* HAS RESPONSIBILITY FOR SKIPPING OVER THE CMT TEXT. *}
(*-----*)

```

```
PROCEDURE PREM;
```

```

BEGIN
  GENKEY (K NOP);
  SCAN (TCKNUM)
END; (* PREM *)

```

```
(*****)
```

```

*-72-----*
* PGOTO GENERATES THE TI-59 GTO STATEMENT AND ITS *
* POTENTIAL ADDRESS SPACE; THE JUMP POINTER FROM THE 1ST *
* NODE OF THIS ADDRESS SPACE IS POINTED TO THAT NODE IN *
* THE CODE DATA STRUCTURE WHICH IS THE START (OR, IN THE *
* CASE OF FORWARD JUMPS, THE POTENTIAL START) OF CODE *
* GENERATED FOR THE WBASIC LINE NUMBER REFERENCED IN THE *
* GOTO COMMAND. *
-----*

```

PROCEDURE PGOTO;

```

BEGIN
  GENKEY (K GTO);
  SCAN (TCKNUM);
  IF TCKNUM <> NUMBERTOK THEN
    PERRCR
  ELSE
    BEGIN
      SETJMFEXT (XNUMBER (ACCUM, ACCINX));
      GENKEY (-2);
      GENKEY (-2);
    END;
  CLCSELINE
END; (* PGOTO *)

```

(*****)

```

*-73-----*
* PGOSUB GENERATES A CALL TO A SUBROUTINE REFERENCED BY *
* WBASIC LINE NUMBER; NOTE THAT ALTHOUGH WBASIC CALLS *
* SUBROUTINES BY LINE NUMBER, THE TI-59 CODE GENERATED *
* CALLS A SUBROUTINE BY A LABEL NAME; AN EXTERNAL JUMP *
* IS SET (AS IN THE GOTO), HOWEVER, RESOLUTION OF THE *
* JUMP WILL BE MADE BY INSERTING THE LABEL USED IN THE *
* CALL IN FRONT OF THE NODE REFERENCED BY THE JMP; THIS *
* INSERTION IS DONE AFTER ALL CODE HAS BEEN GENERATED. *
* NOTE THAT THIS ROUTINE NEITHER CHECKS FOR NOR DOES IT *
* KNOW OF THE EXISTENCE OF A RETURN STATEMENT IN THE *
* SEQUENCE OF SOURCE CODE ASSUMED TO BE THE GOSUB BODY; *
* IF THE USER DOES NOT PROVIDE A RETURN STATEMENT, THEN *
* NO CORRESPONDING TI-59 INVSBR (SBR RETURN) WILL BE *
* GENERATED, AND THE SBR RETURN REGISTER IN THE *
* CALCULATOR WILL NEVER BE CLEARED OF THAT SBR CALL. *
-----*

```

PROCEDURE PGOSUB;

```

BEGIN
  GENKEY (K SBR);
  SCAN (TCKNUM);
  IF TCKNUM <> NUMBERTOK THEN
    PERRCR
  ELSE
    BEGIN
      SETJMFEXT (XNUMBER (ACCUM, ACCINX));
      GENKEY (NEWLBL)
    END;
  CLCSELINE
END; (* PGOSUB *)

```

(*****)

```

*-74-----*
* PRETURN GENERATES THE RETURN FROM A SUBROUTINE. *
* STRUCTURED PROGRAMMING DISCIPLINE DEMANDS A RETURN FOR *
* EACH SUBROUTINE CALL; NOTE THAT THE TI-59 HAS A LIMIT *
* OF SBR RETURN ADDRESSES WHICH CAN BE STACKED; THE USER *
* SHOULD REMEMBER THAT THE WBASIC RETURN STATEMENT IS *
* THE ONLY ONE WHICH WILL GENERATE THE TI-59 INVSBR. *
* FOR A GCSUB GENERATED SBR CALL (FUNCTIONS GENERATE SBR *
* AND INVSBR ALSO, BUT THEY DO THIS AS A RESULT OF THE *
* FNEND STATEMENT OF A ONE-LINE FUNCTION). *
-----*

```

PROCEDURE PRETURN;

```

BEGIN
  GENKEY (K_INVSBR);
  CLOSILINE
END; (* PRETURN *)

```

(*****)

```

*-75-----*
* PPAUSE GENERATES (82) (31) WHICH ARE ACTUALLY A VOID *
* CODE AND THE 'LRN' KEY; WHEN ENTERING HIS PROGRAM INTO *
* THE CALCULATOR THE USER MUST ENTER 'STO 31' INSTEAD OF *
* (82) (31) WHICH CANNOT BE ENTERED DIRECTLY ANYWAY; THEN *
* THE USER MUST BACKSTEP AND CHANGE THE ORIGINAL 'STO 31' *
* BY ISSUING THE FOLLOWING EDITING KEY STROKE SEQUENCE *
* TO THE CALCULATOR IMMEDIATELY AFTER ENTERING THE *
* 'STO 31': BST,BST,NOP,SST. THIS WILL REVISE THE *
* ORIGINAL 'STO 31' TO 'NOP 31'; WHEN ENCOUNTERED BY THE *
* CALCULATOR THESE 2 INSTRUCTIONS WILL STOP EXECUTION BY *
* SHIFTING THE CALCULATOR INTO THE LEARN (LRN) MODE; *
* IN ORDER TO RESUME EXECUTION, THE USER MUST ENTER *
* 'LRN' (PLACING THE DISPLAY REG BACK INTO VIEW) *
* FOLLOWED BY 'R/S' (WHICH RESUMES THE PROCESSING MODE); *
* THIS INTERRUPTION OF EXECUTION DOES NOT CAUSE ANY SIDE *
* EFFECTS AND PROVIDES AN ACCURATE INDICATION OF THE *
* OCCURRENCE OF ANY 'PAUSE' STATEMENTS PLACED IN THE *
* WBASIC SOURCE CODE; THIS IMPLEMENTATION OF THE 'PAUSE' *
* INSTRUCTION PROVIDES A CONVENIENT AND RECOGNIZABLE *
* DEBUGGING/TRANSLATION TOOL WHICH CARRIES A LOW *
* OVERHEAD IN TERMS OF REGISTER/PROGRAM STEP USE. *
-----*

```

PROCEDURE PPAUSE;

```

BEGIN
  GENKEY (82); (* VOID *)
  GENKEY (31); (* LEARN *)
  CLOSILINE
END; (* PPAUSE *)

```

(*****)

```

*-76-----*
{* PSTOP GENERATES CCDE WHICH CAUSES THE TI-59 TO HALT *}
{* EXECUTION AND DISPLAY '888' THUS SIGNALING THAT A *}
{* PROGRAM STOP HAS BEEN ENCOUNTERED INSTEAD OF A DATA *}
{* INPUT OR MAGNETIC CARD LINKING INSTRUCTION. *}
-----*

```

```
PROCEDURE PSTOP;
```

```
VAR I : 1..4;
```

```
BEGIN
```

```
  GENKEY (K_CE);
  FOR I := 1 TO 3 DO
    GENKEY (8);
  GENKEY (K_RS);
  CLOSELINE
```

```
END; (* PSTOP *)
```

```
(*****)
```

```

*-77-----*
{* PEND ASSUMES THAT THE END OF THE WBASIC SOURCE FILE *}
{* HAS BEEN ENCOUNTERED AND WILL INSERT THE END OF FILE *}
{* CHAR INTO THE TOKEN STREAM, CAUSING IMMEDIATE *}
{* TERMINATION OF THE COMPILATION PROCESS. *}
-----*

```

```
PROCEDURE PEND;
```

```
BEGIN
```

```
  GENKEY (K_NOP);
  TOKNUM := -ENDFILTOK
```

```
END; (* PEND *)
```

```

{*****}
{*
CODE RESOLUTION ROUTINES
*}
{*****}

```

```

{-78-----*}
{* PUTGOSUBLEBL USES PROCEDURE INSERTKEY TO ENTER THE LBL *}
{* REFERENCED BY THE GOSUB CALL INTO THE CODE SEQUENCE AT *}
{* LOCATION POINTED TO BY THE JMP (LBLP). *}
{*-----*}

```

```

PROCEDURE PUTGOSUBLEBL (LBL : LBLRNG; VAR LBLP : CODEPTR);
BEGIN
  WITH IELP@.BAKP@ DO
    BEGIN
      INSERTKEY (LBL, SEQP);
      INSERTKEY (K_LBL, SEQP);
    END
END;
(* PUTGOSUBLEBL *)

```

```

{*****}

```

```

{-79-----*}
{* FINDGOSUBLEBL SEARCHES THE CODE DATA STRUCTURE TO FIND *}
{* SBR CALLS FOR WHICH THE JMP HAS BEEN SET; THESE WILL *}
{* CORRESPOND TO BASIC GOSUB STATEMENTS; THE JMP IS *}
{* FOLLOWED AND THE CORRECT LABEL IS INSERTED INTO THE *}
{* CODE SEQUENCE USING THE BAKP AND PROCEDURE PUTGOSUBLEBL *}
{*-----*}

```

```

PROCEDURE FINDGOSUBLEBL (VAR START : CODEPTR);
VAR TRAVELP, TAILP : CODEPTR;
BEGIN
  TRAVELP := START@.SEQP;
  TAILP := START;
  WHILE TRAVELP <> ENDCP@.SEQP DO
    BEGIN
      WITH TRAVELP@ DC
        IF (JMPP <> NIL) AND (TAILP@.KEY = K_SBR) THEN
          BEGIN
            (* FIRST CHECK FOR REDUNDANT GOSUB CALL *)
            IF JMPP@.BAKP@.SEQP@.KEY = K_LBL THEN
              KEY := JMPP@.BAKP@.SEQP@.SEQP@.KEY
            ELSE
              PUTGOSUBLEBL (KEY, JMPP); (* INSERT A LABEL *)
              JMPP@.ADDR := -1; (* UNMARK JMPP ADDR *)
              JMPP := NIL (* RESET JMPP TO NIL *)
          END;
        TAILP := TRAVELP;
        TRAVELP := TRAVELP@.SEQP
    END
END;
(* FINDGOSUBLEBL *)

```

```

{*****}

```

```

(*-80-----*)
(* OSQPAREN (OPTIMIZE SQUEEZE PARENTHESES) REMOVES *)
(* UNNECESSARY PARENTHESES (IN PAIRS) FROM THE CODE DATA *)
(* STRUCTURE FOR THE MOST COMMON CASES, NAMELY '(RCL NN)' *)
(* AND '<LITERAL NUMERIC>' *)
(*-----*)

```

```
PROCEDURE OSQPAREN (START : CODEPTR);
```

```
VAR CPEN, CLOSE, TAILP, MOVEP : CODEPTR;
    CPENCT, CLOSECT : INTEGER;
```

```
(*-----*)
```

```

(*-80-01-----*)
(* CCUNTREP COUNTS THE NUMBER OF SEQUENTIAL OCCURENCES OF *)
(* KEYC AT A PARTICULAR LOCATION IN THE CODE DATA STRUCTR *)
(* STRUCTURE; NOTE THAT IT ALSO CHECKS FOR JMPP POINTERS *)
(* TO THESE KEYS. *)
(*-----*)

```

```
FUNCTION CCUNTREP (VAR MOVEP:CODEPTR; KEYC:INTEGER):INTEGER;
```

```
VAR COUNT : INTEGER;
```

```

BEGIN
  COUNT := 0;
  WHILE (MOVEP@.KEY = KEYC) AND (MOVEP@.ADDR = -1) DO
    BEGIN
      MOVEP := MOVEP@.SEQP;
      COUNT := COUNT + 1;
    END;
  COUNTREP := COUNT
END;
(* COUNTREP *)

```

```
(*-----*)
```

```

(*-80-02-----*)
(* NUMBERUN MOVES ITS POINTER PARAMETER PASSED ANY NODE *)
(* WHICH CONTAINS A NUMERIC LITERAL KEY CODE AND HAS NO *)
(* POINTER REFERENCE; IT IS ASSUMED HERE THAT NO JUMP *)
(* POINTER IS EVER SET IN THE MIDDLE OF A NUMERIC LITERAL *)
(* KEY SEQUENCE, ELSE PART OF THE NUMBER MAY BE REMOVED. *)
(*-----*)

```

```
PROCEDURE NUMBERUN (VAR MOVEP : CODEPTR);
```

```

BEGIN
  WHILE (MOVEP@.KEY IN NUMERICKEY) AND (MOVEP@.ADDR = -1) DO
    MOVEP := MOVEP@.SEQP;
  END;
(* NUMBERUN *)

```

```
(*-----*)
```

```

(*-80-03-----*)
(* REMOVEPAREN TAKES PAIRS OF NODES OUT OF THE CODE DATA *)
(* STRUCTURE; NOTE THAT THIS PROCEDURE DOES NOT KNOW WHAT *)
(* CODE IT IS REMOVING; THAT IS DEFINED BY OSQPAREN. *)
(*-----*)

```

```

PROCEDURE REMOVEPAREN (VAR OPEN, CLOSE : CODEPTR;
                      OPENCT, CLOSECT : INTEGER);
BEGIN
  REPEAT
    OPEN@.SEQP := OPEN@.SEQP@.SEQP;
    OPENCT := OPENCT - 1;
    CLOSE@.SEQP := CLOSE@.SEQP@.SEQP;
    CLOSECT := CLOSECT - 1
  UNTIL (OPENCT = 0) OR (CLOSECT = 0)
END;
(* REMOVEPAREN *)

```

```

(*-----*)

```

```

BEGIN (* CSQPAREN MAIN *)
  MOVEP := START;
  WHILE MCVEP@.SEQP <> NIL DO
    BEGIN
      IF (MOVEP@.KEY = K_OPAREN) AND (MOVEP@.ADDR = -1) THEN
        BEGIN
          OPEN := TAILP; (* SET OPEN PTR *)
          OPENCT := CCUNTRP (MOVEP, K_OPAREN);
          IF (MOVEP@.KEY=K_RCL) AND (MOVEP@.ADDR=-1) THEN
            BEGIN
              CLOSE := MOVEP@.SEQP; (* SET CLOSE PTR *)
              MOVEP := MOVEP@.SEQP@.SEQP; (* MOVE AHEAD *)
              CLOSECT := CCUNTRP (MOVEP, K_CPAREN);
              IF CLOSECT > 0 THEN (* IF EXTRAS, DELETE *)
                REMOVEPAREN (OPEN, CLOSE, OPENCT, CLOSECT)
            END
          ELSE IF (MOVEP@.KEY IN NUMERICKEY) AND
                 (MCVEP@.ADDR = -1) THEN
            BEGIN
              WHILE MCVEP@.SEQP@.KEY IN NUMERICKEY DO
                MOVEP := MOVEP@.SEQP; (* PASS OVER NUMBER *)
                CLOSE := MOVEP; (* SET CLOSE PTR *)
                MOVEP := MOVEP@.SEQP; (* MOVE AHEAD *)
                CLOSECT := CCUNTRP (MOVEP, K_CPAREN);
                IF CLOSECT > 0 THEN (* IF EXTRAS, DELETE *)
                  REMOVEPAREN (OPEN, CLOSE, OPENCT, CLOSECT)
            END
          END;
          TAILP := MOVEP;
          MCVEP := MOVEP@.SEQP
        END
      END;
    END;
  END;
(* OSQPAREN *)

```

```

(*****

```

```

(*-81-----*)
(* OSCNCF (OPTIMIZE SQUEEZE NOP) LOCATES ALL 'NOP' KEY *)
(* CODES, RESETS POINTER REFERENCES TO THEM IF THEY EXIST, *)
(* AND THEN PINCHES THEM OUT OF THE CODE DATA STRUCTURE. *)
(*-----*)

```

```

PROCEDURE OSCNOP (VAR START : CODEPTR);

```

```

VAR   CUR   : CODEPTR;
      I     : 0..3;
      INDEX : CTEXTNG;

```

```

BEGIN
  CUR := START;
  WHILE CUR <> NIL DC
  BEGIN
    IF CUR@.JMPP <> NIL THEN
      WHILE (CUR@.JMPP@.KEY = K NOP)
      AND (CUR@.JMPP <> ENDCP) DO
        BEGIN
          CUR@.JMPP@.ADDR := -1;
          CUR@.JMPP@.SEQP := CUR@.JMPP@.SEQP;
          CUR@.JMPP@.ADDR := 0;
        END;
        CUR := CUR@.SEQP
      END;
      CUR := START;
      WHILE CUR@.SEQP <> NIL DO
      BEGIN
        INDEX := CUR@.KEY;
        FOR I := 1 TO (CTEXT(.INDEX).UNIT) DO
          CUR := CUR@.SEQP;
          IF (CUR@.SEQP@.KEY = K NOP) AND (CUR@.SEQP@.ADDR = -1)
          THEN CUR@.SEQP := CUR@.SEQP@.SEQP
          ELSE
            CUR := CUR@.SEQP
          END;
        END;
      END;
    END;
  END;
  (* SQUEEZE OUT NOPS *)
  (* FIX THE INDEX TO CTEXT *)
  (* BYPASS REG/ADDR SPACES *)
  (* REMOVE NOP *)
  (* NEXT NODE *)
  (* OSCNOP *)
  (*****)

```

```
(*--82-----*)
(* RESOLVE ADDR FILLS THE ADDR FIELDS OF ALL TI-59 CODE *)
(* NODES LINKED IN THE CODE DATA STRUCTURE, AND THEN *)
(* FILLS THE KEY FIELDS OF NODES WHICH HAVE NON-NIL *)
(* JMPP'S WITH THE ABSOLUTE ADDR POINTED TO BY THOSE *)
(* JMPP'S; JMPP'S ARE THEN SET BACK TO NIL. *)
(*-----*)
```

```
PROCEDURE RESOLVE_ADDR (START : CODEPTR);
```

```
VAR TRAVEL : CODEPTR;
    I : INTEGER;
```

```
(*-----*)
```

```
(*--82-01-----*)
(* INSERT JMPADDR CONVERTS THE ADDR FOUND AT THE NODE *)
(* REFERENCED BY JMPP PTR INTO A TI-59 MACHINE CODE ADDR *)
(* (2 INTEGERS IN RANGE 0..99), AND INSERTS IT INTO THE *)
(* THE KEY FIELDS (OCCUPIED BY -2'S) OF THE NODES FROM *)
(* WHICH THE JMPP ORIGINATES. *)
(*-----*)
```

```
PROCEDURE INSERT_JMPADDR (JADDR : INTEGER);
```

```
VAR HIPART, LOPART : INTEGER;
```

```
BEGIN
  HIPART := JADDR DIV 100;
  LOPART := JADDR - HIPART * 100;
  TRAVEL@.KEY := HIPART;
  TRAVEL@.SEQP@.KEY := LOPART;
END;
(* SPLIT ADDR INTO *)
(* HI/LO PARTS; *)
(* OVERWRITE NOP'S *)
(* W/ ABS ADDR'S. *)
(* INSERT_JMPADDR *)
```

```
(*-----*)
```

```
BEGIN (* RESOLVE_ADDR MAIN *)
  TRAVEL := START;
  I := 0;
  WHILE TRAVEL <> ENCCP@.SEQP DO (* INSERT ABSOLUTE ADDR *)
    BEGIN
      TRAVEL@.ADDR := I;
      TRAVEL := TRAVEL@.SEQP;
      I := I + 1;
    END;
  TRAVEL := START;
  WHILE TRAVEL <> ENCCP@.SEQP DO (* FIND/JUSTIFY JMP ADDR *)
    BEGIN
      WITH TRAVEL@ DC
        IF JMPP <> NIL THEN (* FIND JMPP'S WHICH ARE SET *)
          BEGIN
            INSERT_JMPADDR (JMPP@.ADDR);
            JMPP := NIL (* SET JMPP BACK TO NIL *)
          END;
      TRAVEL := TRAVEL@.SEQP;
    END;
  END;
END; (* RESOLVE_ADDR *)
```

```

{*****}
{*
      OUTPUT DUMP ROUTINES
*}
{*****}

```

```

{* -83 ------*}
{* FINDMSG LOCATES THE START OF THE CORRECT MESSAGE IN *}
{* THE MSGFILE. *}
{* ------*}

```

```

PROCEDURE FINDMSG (VAR MSGFILE : TEXT; VAR ESCHAR : CHAR;
                  MSG_NO : INTEGER);
VAR CH : CHAR;
    I : INTEGER;

```

```

BEGIN
  RESET (MSGFILE, 'NAME=MSGF.PASCAL.A');
  READLN (MSGFILE, ESCHAR);
  REPEAT
    READ (MSGFILE, CH);
    IF CH = ESCHAR THEN (* CHECK FOR ESCAPE CHAR & MSG NO *)
      READLN (MSGFILE, I)
    ELSE
      READLN (MSGFILE)
  UNTIL (EOF (MSGFILE)) OR
        ((CH = ESCHAR) AND (I = MSG_NO))
END; (* FINDMSG *)

```

```

{*****}
{* -84 ------*}
{* WRITLN WRITES A FULL MESSAGE FROM '$N' TO $N' AS FOUND *}
{* IN THE MSGFILE. *}
{* ------*}

```

```

PROCEDURE WRITLN; (* FWD DECL WITH I/O COMMAND ROUTINES *)
VAR CH, ESCHAR : CHAR;
    I : INTEGER;

```

```

BEGIN
  FINDMSG (MSGFILE, ESCHAR, MSG_NO);
  REPEAT
    READ (MSGFILE, CH);
    IF CH=ESCHAR THEN (* CHECK FOR EMBEDDED ESCAPE CHARS *)
      READLN (MSGFILE, I) (* AND DISCARD IF FOUND. *)
    ELSE
      WRITE (WFILE, CH);
      IF ECIN (MSGFILE) THEN (* NEXT LINE *)
        BEGIN
          READLN (MSGFILE);
          WRITELN (WFILE)
        END
      END
  UNTIL (EOF (MSGFILE)) OR
        ((CH = ESCHAR) AND (I = MSG_NO))
END; (* WRITLN *)

```

```

{*****}

```

```
(* -85 ----- *)
{* WRIT WRITES A ONE-LINE MESSAGE OR THE FIRST LINE OF A *}
{* MESSAGE FROM THE MSGFILE. *}
(* ----- *)
```

```
PROCEDURE WRIT (VAR WFILE, MSGFILE : TEXT;
                MSG_NO : INTEGER);
VAR CH, ESCHAR : CHAR;
    I : INTEGER;
```

```
BEGIN
  FINDMSG (MSGFILE, ESCHAR, MSG_NO);
  REPEAT
    READ (MSGFILE, CH);
    WRITE (WFILE, CH)
  UNTIL EOLN (MSGFILE)
END; (* WRIT *)
```

```
(*****)
```

```
(* -86 ----- *)
{* REPORT CCMPTES AND WRITES THE REGISTER/LABEL SUMMARY. *}
(* ----- *)
```

```
PROCEDURE REPORT (VAR WFILE : TEXT);
```

```
VAR LTCTAL : LBLRNG;
    RTCTAL : INTEGER;
```

```
BEGIN
  WRITLN (WFILE, MSGF, 3);
  WRITELN (WFILE, ERRCRCT:7, ' FATAL ERRORS. ');
  WRITELN (WFILE, WARNCT:7, ' WARNING MSGS. ');
  IF ERRCRCT > 0 THEN (* CALCULATIONS INCOMPLETE *)
    WRITLN (WFILE, MSGF, 14);
  RTCTAL := NEXTREG - STARTREG;
  LTCTAL := LBLCT - 1;
  WRITELN (WFILE);
  WRITELN (WFILE, NEXTREG:1, ' IS NEXT AVAILABLE REGISTER ');
  WRITELN (WFILE, ' TOTAL REGISTERS RESERVED = ', RESERVECT:1);
  WRITELN (WFILE, ' TOTAL REGISTERS USED = ', RTOTAL:1);
  WRITELN (WFILE, ' TOTAL LABELS USED = ', LTOTAL:1);
  WRITLN (WFILE, MSGF, 4)
END; (* REPORT *)
```

```
(*****)
```

```

(*-87-----*)
(* CODEDUMP WRITES THE TI-59 CODE STORED IN THE CODE DATA *)
(* STRUCTURE AND APPLIES THE CTEXTF FILE TO EACH STEP TO *)
(* PRODUCE THE LITERAL TEXT OF FOR THE KEY STROKES. *)
(*-----*)

PROCEDURE CODEDUMP (VAR WFILE : TEXT; VAR TICODE : CODEPTR);
VAR CUR, HCLD : CODEPTR;
    I : 0..3;

(*-----*)
(*-87-01-----*)
(* WRCODE WRITES THE NUMERICAL FORM OF TI-59 ADDR AND KEY *)
(*-----*)

PROCEDURE WRCODE (VAR CUR : CODEPTR);
BEGIN
    WRITE (WFILE, ' ':5);
    ZEROPAD (WFILE, CUR@.ADDR, 3);
    WRITE (WFILE, ' ':3);
    ZEROPAD (WFILE, CUR@.KEY, 2);
    WRITE (WFILE, ' ':3)
END; (* WRCODE *)

(*-----*)

BEGIN (* CODEDUMP MAIN *)
    CUR := TICODE;
    WRITLN (WFILE, MSGF, 5); (* HEADER MSG *)
    WRITE (WFILE, '$'); (* '$' MUST BE WRITTEN HERE, *)
    (* ELSE WILL INTERFERE W/ WRITLN *)
    WRITLN (WFILE, MSGF, 6); (* BEGIN CODE MSG *)
    WHILE CUR@.SEQ# <> NIL DO
        BEGIN
            WRCCDE (CUR);
            WRITLN (WFILE, CTEXT (.CUR@.KEY.).CODECHAR);
            IF CUR@.KEY IN (.K SBR, K LBL.) THEN
                BEGIN (* MUST NOT TAKE SBR'S OR LBL'S LITERALLY *)
                    CUR := CUR@.SEQ#;
                    WRCODE (CUR);
                    WRITLN (WFILE, CTEXT (.CUR@.KEY.).CODECHAR)
                END
            ELSE
                BEGIN
                    HCLD := CUR;
                    FOR I := 1 TO (CTEXT (.HOLD@.KEY.).UNIT) DO
                        BEGIN (* UNIT FIELD DEFINES TYPE INSTRUCTION *)
                            CUR := CUR@.SEQ#;
                            WRCODE (CUR);
                            ZEROPAD (WFILE, CUR@.KEY, 2);
                            WRITLN (WFILE)
                        END
                    END;
                    CUR := CUR@.SEQ#
                END
        END;
    END;
END; (* CODEDUMP *)

(*****

```

```
(*--88-----*)
(* LINK INTERFACE CREATES THE SCRATCH FILE WHICH PROVIDES *)
(* THE LINKER WITH ALL THE INFORMATION IT MUST HAVE TO *)
(* SEGMENT THE TI-59 CODE; ENTRIES IN SCRATCH ARE IN THE *)
(* FORM OF SUB-FILES (MESSAGES) DELIMITED BY "$N". *)
(*-----*)
```

```
PROCEDURE LINK_INTERFACE;
```

```
(*--88-01-----*)
(* LOGTO IS USED BY LINK INTERFACE TO READ AND WRITE *)
(* FILES TO THE SCRATCH FILE (COPY). *)
(*-----*)
```

```
PROCEDURE LOGTO (VAR WFILE, RFILE : TEXT; MSGNO : INTEGER);
VAR CH : CHAR;
```

```
BEGIN
WRITE (WFILE, '$', MSGNO: 1); (* WRITE MSG DELIMITER *)
WHILE NCT EOF (RFILE) DO
BEGIN
WHILE NOT EOLN (RFILE) DO
BEGIN
READ (RFILE, CH); (* COPY THE FILE TO SCRATCH *)
WRITE (WFILE, CH)
END;
WRITELN (WFILE);
IF NCT EOF (RFILE) THEN
READLN (RFILE)
END;
WRITELN (WFILE, '$', MSGNO: 1); (* WRITE MSG DELIMITER *)
WRITELN (WFILE) (* LOGTO *)
END;
```

```
(*-----*)
```

```
BEGIN (* LINK_INTERFACE MAIN *)
REWRITE (SCRATCH, 'NAME=SCRATCH.PASCAL.A');
WRITELN (SCRATCH, '$1'); (* NEXT REGISTER = MSG $1 *)
WRITELN (SCRATCH, NEXTREG: 1, ' IS NEXT AVAILABLE REG. ');
WRITELN (SCRATCH, '$1'); WRITELN (SCRATCH);
WRITELN (SCRATCH, '$2'); (* TI-59 CODE = MSG $2 *)
CODEDUMP (SCRATCH, BEGINCF);
WRITELN (SCRATCH, MSGF, 7); (* END CODE MSG *)
WRITELN (SCRATCH, '$2'); WRITELN (SCRATCH);
RESET (NAMEF, 'NAME=NAMEF.WBASIC.A'); (* REG/NAME MAP *)
LOGTO (SCRATCH, NAMEF, 3); (* = MSG $3 *)
IF NCT FIRSTREAD THEN (* DATA/READ MAP = MSG $4 *)
BEGIN
RESET (READF, 'NAME=READF.WBASIC.A');
LOGTO (SCRATCH, READF, 4)
END
END;
```

```
(* LINK_INTERFACE *)
```

```
(*****)
```

```

(*-89-----*)
{* SYMBELDUMP IS A SPECIAL PURPOSE ROUTINE USED FOR *}
{* DEBUGGING; IT WILL DUMP THE ENTIRE CONTENTS OF THE *}
{* COMPILER SYMBOL TABLE BUCKET BY BUCKET; THIS ROUTINE *}
{* IS TOGGLED USING OPTION NUMBER 5. *}
-----*)

```

```

PROCEDURE SYMBELDUMP (VAR WFILE : TEXT; BUCKET : HASH);
VAR
  I      : INTEGER;
  LOCK  : SLOTPTR;
BEGIN
  WRITLN (WFILE, MSGF, 10);          (* HEADER MSG *)
  FOR I := 0 TO HASHEASE DO
    IF BUCKET(.I.) <> NIL THEN      (* SKIP EMPTY BUCKETS *)
      BEGIN
        ZERCPAD (WFILE, I, 2);
        WRITLN (WFILE, MSGF, 11);  (* BUCKET BOUNDARY MSG *)
        LOCK := BUCKET(.I.);
        REPEAT                      (* UNTIL LOCK = NIL *)
          WRITE (WFILE, ' ':11, LOCK@.IDENT, ' ':1);
          WITH LOCK@ DO
            CASE TYP OF
              VARID : BEGIN
                IF REGNO < 0 THEN    (* PI = -314 *)
                  WRITELN (WFILE, '.. CONSTANT');
                ELSE
                  BEGIN
                    ZEROPAD (WFILE, REGNO, 2);
                    WRITELN (WFILE, ' GLOBAL VAR');
                    IF AUXREG1<>-1 THEN (* IF USED *)
                      BEGIN
                        WRITE (WFILE, ' ':32);
                        ZEROPAD (WFILE, AUXREG1, 2);
                        WRITELN (WFILE, ' AUXREG 1');
                      END;
                    IF AUXREG2<>-1 THEN (* IF USED *)
                      BEGIN
                        WRITE (WFILE, ' ':32);
                        ZEROPAD (WFILE, AUXREG2, 2);
                        WRITELN (WFILE, ' AUXREG 2');
                      END;
                  END;
                END;
              END;
              FNQID : WRITELN (WFILE, '.. QUICK FN');
              FNLID : BEGIN
                ZEROPAD (WFILE, FNLREG, 2);
                WRITELN (WFILE, ' LONG FN');
              END;
              FNPID : BEGIN
                ZEROPAD (WFILE, FNREGNO, 2);
                WRITELN (WFILE, ' PARAMETER FN');
              END;
            END;
          (* CASE *)
          LOCK := LOCK@.SLOT
        UNTIL LOCK = NIL
        END;
      WRITLN (WFILE, MSGF, 12)      (* END SYMBEL MSG *)
    END;
  (* SYMBELDUMP *)
  (*****)

```

```

(*-90-----*)
(* SEARCH IS A SPECIAL PURPOSE DEBUGGING TOOL; *)
(* THIS PROCEDURE FOLLOWS AND PRINTS THE CONTENTS OF ALL *)
(* POINTERS IN THE CODE DATA STRUCTURE (LINE AND CODEPTR) *)
(* THIS ROUTINE CAN BE TOGGLED USING OPTION NUMBER 6. *)
(*-----*)

```

```

PROCEDURE SEARCH (VAR WFILE : TEXT; LSTART : LINEPTR);
VAR LPSEARCH : LINEPTR; CODE : CODEPTR;
BEGIN
  WRITLN (WFILE, MSGF, 13); (* HEADER MSG *)
  LPSEARCH := LSTART;
  REPEAT
    WRITE (WFILE, 'LINUM = ');
    ZEROPAD (WFILE, LPSEARCH@.LINO, 5); (* WBASIC LINE NO *)
    WRITELN (WFILE);
    CODP := LPSEARCH@.CPTR;
    REPEAT (* TI-59 CODE ATTACHED TO WBASIC LINE NO *)
      WRITE (WFILE, ':2');
      ZEROPAD (WFILE, CODP@.ADDR, 3);
      WRITE (WFILE, ':2');
      ZEROPAD (WFILE, CODP@.KEY, 2);
      WRITELN (WFILE);
    CODP := CODP@.SEOP
  UNTIL (CODP = LPSEARCH@.LPTR@.CPTR) OR (CODP = NIL);
  LPSEARCH := LPSEARCH@.LPTR
  UNTIL LPSEARCH@.LINO = MAXBASLIN (* MAXBASLIN IS END *)
END; (* SEARCH *)

```

```

{*****}
{*
  INITIALIZATION ROUTINE
*}
{*****}

```

```

{*--91-----*}
{* INITIALIZE SETS UP ALL FILES, DATA STRUCTURES, SETS, *}
{* AND INITIAL VARIABLE VALUES REQUIRED TO BEGIN THE *}
{* READING AND COMPILATION OF THE WBASIC SOURCE CODE, AND *}
{* THE OUTPUT OF THE TRANSLATED TI-59 CODE AND LISTINGS. *}
{*-----*}

```

```
PROCEDURE INITIALIZE;
```

```
VAR I : INTEGER;
```

```
(*-----*)
```

```

{*--91-01-----*}
{* LOADRW READS THE RWTBLF FILE (RESERVED WORD TABLE) AND *}
{* LOADS THE RESERVED WORD CHAR/INDEX ARRAYS; NOTE THAT *}
{* THE ARRAYS ARE STATIC FIXED AND ARE DEFINED BY THE *}
{* SYSTEM PARAMETERS RWCHARCT, RWORDCT, RWLENGCT IN THE *}
{* CONSTANT DECLARATION BLOCK AT THE FRONT OF THE PROGRAM *}
{*-----*}

```

```
PROCEDURE LOADRW (VAR RWTBLF : TEXT);
```

```

VAR CHINX, STARTCHINX : 0..RWCHARCT + 1;
    WINX              : 0..RWORDCT + 1;
    LINX, LENG       : 0..RWLENGCT + 1;
    CH                : CHAR;

```

```

BEGIN
  LINX := 0; (* INIT LENGTH INDEX *)
  CHINX := 1; (* INIT CHAR INDEX *)
  WHILE NCT EOF(RWTBLF) DO
    BEGIN
      STARTCHINX := CHINX;
      READ (RWTBLF, WINX); (* READ WORD INDEX (INTEGER) *)
      READ (RWTBLF, CH, CH); (* READ OFF 2 BLANK SPACES *)
      RWCRC (.WINX.) := CHINX;
      REPEAT (* READ CHARS OF ONE WORD INTO CHAR ARRAY *)
        READ (RWTBLF, RWCHAR (.CHINX.));
        CHINX := CHINX + 1;
      UNTIL EOLN (RWTBLF);
      REACLN (RWTBLF); (* NEXT WORD *)
      LENG := CHINX - STARTCHINX;
      IF LENG > LINX THEN (* IF LENGTH CHANGE, THEN *)
        BEGIN (* INDEX ITS LOCATION IN *)
          LINX := LENG; (* THE LENGTH ARRAY. *)
          RWLENG (.LINX.) := WINX;
        END
      END;
      RWCHAR (.RWCHARCT + 1.) := BLANK; (* SET DELIMITERS FOR *)
      RWCRC (.RWORDCT + 1.) := RWCHARCT + 1; (* ARRAYS AND *)
      RWLENG (.RWLENGCT + 1.) := RWORDCT + 1; (* INDICES *)
    END;
  END; (* LOADRW *)

```

```
(*-----*)
```

```

(*-91-02-----*)
{* LOADLIB READS PREDEFINED FUNCTION LIBRARIES IN BIFNLF *}
{* AND BIFNCF FILES; MAKES APPROPRIATE SYM TBL ENTRIES. *}
(*-----*)

```

```

PROCEDURE LOADLIB (VAR LIBFILE : TEXT; FNTYPE : IDTYP;
                  SEQLEN : INTEGER);

```

```

VAR  IDSICT : SLCTPTR;
     I : INTEGER;

```

```

BEGIN
  READLN (LIBFILE); READLN (LIBFILE); (* SKIP HEAD LINES *)
  WHILE NCT EOF(LIBFILE) DO
    BEGIN
      ACCINX := 0; (* INIT ACCUM INDX *)
      REPEAT (* READ NAME OF FN *)
        ACCINX := ACCINX + 1;
        READ (LIBFILE, ACCUM(.ACCINX.));
      UNTIL ACCUM(.ACCINX.) = BLANK; (* TO 1ST BLANK *)
      FOR I := ACCINX TO MAXTOKLEN DO (* FILL REST BLANK *)
        ACCUM(.I.) := BLANK;
      ACCINX := ACCINX - 1; (* SET INDEX BACK TO NAME LEN *)
      IDSLOT := GETSLOT(ACCUM, ACCINX); (* ENTER IN SYMTEL *)
      IDSLOT@.TYP := FNTYPE; (* SET IDENT TYPE *)
      FOR I := 1 TO SEQLEN DO (* READ KEY CODES *)
        CASE FNTYPE OF
          FNCID : READ (LIBFILE, IDSLOT@.FNQ(.I.));
          FNLLD : BEGIN
                     IDSLOT@.FNLLINK := NIL;
                     READ (LIBFILE, IDSLOT@.FNL(.I.))
                   END
        END; (* CASE *)
      READLN (LIBFILE) (* SKIP TO NEXT IN *)
    END
  END; (* LOADLIB *)

```

```

(*-----*)

```

```

(*-91-03-----*)
{* LCADCTEXT READS THE CTEXTF FILE AND LOADS THE DATA *}
{* STRUCTURE WHICH WILL PROVIDE THE TRANSLATIONS OF TI-59 *}
{* KEY CCIES DURING THE FINAL CODE DUMP. *}
(*-----*)

```

```

PROCEDURE LCADCTEXT;

```

```

VAR  I : INTEGER;
     J, K : 1..TEXTLEN + 1;
     CH : CHAR;

```

```

BEGIN
  READLN (CTEXTF); READLN (CTEXTF);
  WHILE NCT EOF(CTEXTF) DO
    BEGIN
      READ (CTEXTF, I, CTEXT(.I.).UNIT); (* SKIP TWO BLANKS *)
      READ (CTEXTF, CH, CH);
      J := 1;
      WHILE NOT EOLN(CTEXTF) DO
        BEGIN
          READ (CTEXTF, CTEXT(.I.).CODECHAR(.J.));
          J := J + 1;
        END;
      FOR K := J TO TEXTLEN DO
        CTEXT(.I.).CODECHAR(.K.) := BLANK;
      READLN (CTEXTF);
    END
  END;

```

```

END; (* LCADCTEXT *)

```

```

(*-----*)

```

```

BEGIN      (* INITIALIZE MAIN *)

(*-----*)
(* OPEN ALL FILES AND WRITE OUTPUT FILE HEADERS. *)
(*-----*)

TERMOUI (OUTFILE);
RESET (FASICF, 'NAME=BASICF.WBASIC.A');
RESET (MSGF, 'NAME=MSGF.PASCAL.A');
RESET (RWTBLF, 'NAME=RWTBLF.PASCAL.A');
RESET (LABELF, 'NAME=LABELF.PASCAL.A');
RESET (CTEXTF, 'NAME=CTEXTF.PASCAL.A');
RESET (EIFNOF, 'NAME=BIFNOF.PASCAL.A');
RESET (EIFNLF, 'NAME=BIFNLF.PASCAL.A');
REWRITE (LISTF, 'NAME=LISTF.WBASIC.A');
REWRITE (NAMEF, 'NAME=NAMEF.WBASIC.A');
WRITLN (LISTF, MSGF, 2); (* HEADER MSG TO LISTF *)
WRITLN (OUTFILE, MSGF, 1); (* TERMINAL INVOKE MSG *)
WRITLN (NAMEF, MSGF, 8); (* HEADER MSG TO NAMEF *)

(*-----*)
(* INITIALIZE OPTION TOGGLES *)
(*-----*)

LINK59 := FALSE; (* OPTION 0 *)
PC100 := TRUE; (* OPTION 1 *)
OPTPAR := TRUE; (* OPTION 2 *)
OPTNOP := TRUE; (* OPTION 3 *)
CODUMP := TRUE; (* OPTION 4 *)
SYDUMP := FALSE; (* OPTION 5 *)
DSDUMP := FALSE; (* OPTION 6 *)
TOKOUT := FALSE; (* OPTION 7 *)
TOKLIS := FALSE; (* OPTION 8 *)

(*-----*)
(* INITIALIZE RESERVED WORD ARRAY INDEXES. *)
(*-----*)

LOADRW (RWTBLF);

(*-----*)
(* INITIALIZE CHARACTER SETS. *)
(*-----*)

LETTERS := ('A'..'I') + ('J'..'R') + ('S'..'Z');
DIGITS := ('0'..'9');
ALFANUM := LETTERS + DIGITS;
SIGNS := ('+'..'>');
DOUBLE1 := ('<'..'>');
DOUBLE2 := ('='..'!');
SPECIALS := ('+'..'>') + ('='..'!') + ('/'..'&') + (','..');
SUBERRCR := ('$'..'&');
CRITICAL := (.ENDLIN.) + (.ENDFIL.) + ('&');
TRAILTCKS := (.CMTCKEXC.) + (.ENDLINTOK.) + (.ENDFILTOK.);
BINCPICKS := (.PLUSTOK.) + (.MINUSTOK.) + (.MULTOK.);
DIVICK := (.DIVICK.);
REICPICKS := (.EQUALTOK.) + (.NOTEQTOK.) + (.GTEQTCK.);
LTECTOK := (.LTECTOK.) + (.LTTOK.) + (.GTTOK.);
NUMERICKEY := (.K_DECPT.) + (.K_EE.) + (.K_NEG.);
SIGNTCKS := (.PLUSTCK.) + (.MINUSTOK.);
BEGIN_EXPRTOKS := SIGNTOKS + (.IDENTOK.) + (.OPARENTOK.);

```

```

{*-----*}
{* INITIALIZE HASH TABLE AND REGISTER COUNT. *}
{*-----*}

FOR I := 0 TO HASHEASE DO
    BUCKET(.I.) := NIL;
NEXTREG := STARTREG;

{*-----*}
{* INITIALIZE ARRAY HOLDING OUTPUT TEXT OF TI-59 CODE. *}
{*-----*}

LOADCTEXT;

{*-----*}
{* INITIALIZE BUILT-IN FUNCTION LIBRARY. *}
{*-----*}

LOADLIB (EIFNOF, FNCID, FNLEN);
LOADLIB (EIFNLF, FNIID, FNLEN);

{*-----*}
{* ENTER 'PI' = 3.14159265359 IN SYMBOL TABLE. *}
{*-----*}

ACCUM(.1.) := 'P'; ACCUM(.2.) := 'I';
ACCINX := 2;
FOR I := 3 TO MAXTCKLEN DC
    ACCUM(.I.) := BLANK;
IDSLOT := GETSLOT (ACCUM, ACCINX);
IDSLOT@.TYP := VARID;
IDSLOT@.REGNO := -314; (* SPECIAL REGNO FOR 'PI' *)

{*-----*}
{* INITIALIZE LABEL STACK (ARRAY OF INTEGER KEY CODES). *}
{*-----*}

READLN (LAEELF); (* SKIP HEAD LINE *)
FOR I := 1 TO LBLBASE DO
    READ (LABELF, CLAEEL(.I.));
LBLCT := 1;

{*-----*}
{* INITIALIZE RESERVED REGISTER SET. *}
{*-----*}

READLN (LAEELF); READLN (LABELF); READLN (LABELF);
RESERVECT := 0;
RESERVE REG := (.); (* INITIALIZE TO EMPTY SET *)
WHILE NOT EOF (LAEELF) DO
    BEGIN
        WHILE NOT EOLN (LABELF) DO
            BEGIN
                READ (LABELF, I);
                RESERVE REG := RESERVE REG + (.I.); (* MAKE SET *)
                RESERVECT := RESERVECT + 1 (* COUNT MEMBERS *)
            END;
        READLN (LABELF)
    END;

{*-----*}
{* INITIALIZE FNF ACTIVATION STACK AND FNL USE LIST. *}
{*-----*}

FNSTACKCT := 0;
FNSTACK := NIL;
FNLLIST := NIL;

```

```
{*-----*}
{* INITIALIZE FIRST CALL TO SCAN. *}
```

```
LINEUF(.0.) := BLANK;
LINEUF(.1.) := ENDLIN;
TOKNUM := ENDLINTCK;
LNBINX := 0;
ERRCRCT := 0;
WARNCT := 0;
FLAGCMT := FALSE;
LINUM := 0;
LLINUM := 0;
CLINUM := 0;
```

```
{*-----*}
{* INITIALIZE LINKED DATA STRUCTURE FOR TI-59 CODE. *}
```

```
FIRSTLP := GETNEWHDR (LINUM); (* SET A COMMON REF NODE *)
LPCUR := FIRSTLP; (* ANCHOR ALL MARKER PTRS TO IT *)
LASTLP := FIRSTLP;
ENDCP := FIRSTLP@.CPTR;
BEGINCF := ENDCP;
CPCUR := ENDCP;
SETLINE (LPCUR, LP); (* SET UP FOR MAIN PROCEDURE LABEL *)
GENKEY (K LBL);
GENKEY (NEWLBL); (* MAIN PROCEDURE = LBL A *)
BEGINCF := BEGINCP@.SEQP; (* BYPASS THE HEADER NODE *)
```

```
{*-----*}
{* INITIALIZE LOOP/BRANCH STACKS. *}
```

```
IFSTACK := NIL;
ENDIFSTACK := NIL;
LOOPSTACK := NIL;
ENDLOOPSTACK := NIL;
FORSTACK := NIL;
NEXTSTACK := NIL;
```

```
{*-----*}
{* INITIALIZE READ/DATA STATEMENT INDEXES/FLAGS. *}
```

```
READIX := 1;
DATAIX := 1;
INDEXERROR := FALSE;
FIRSTREAD := TRUE
```

END;

(* INITIALIZE *)

```

*****
*
*           FAX59:  MAIN DRIVER
*
*****

```

```
BEGIN (* FAX59 MAIN *)
```

```
INITIALIZE;
```

```
REPEAT (* UNTIL TCKNUM = ENDFILTOK *)
```

```

SCAN (TCKNUM); (* SCAN FIRST WORD OF NEW LINE *)
IF ERRORCT = 0 THEN (* PARSEING IS DICONTIUNED AFTER *)
  BEGIN (* FIRST FATAL ERROR ENCOUNTERED *)
    SETLINE (LPCUR, LP); (* NEW WBASIC LINE NO & LINE *)
    CASE TOKNUM OF (* RECURSIVE DESCENT PARSE PRCC *)

```

```

-----
* KEYWORDS MARKED IN RIGHT CMT COLUMN BY ASTERISKS MUST
* ALWAYS RESULT IN A PARSE ERROR IF USED AS A COMMAND
* (IE. 1ST WORD ON A LINE) REGARDLESS OF IMPLEMENTATION:
* ** IMPLEMENTED IN THIS SUBSET
* *** NOT IMPEMENTED IN THIS SUBSET
-----

```

```

ERRORCT      : BEGIN END; (* SCAN ERROR *)
CMTOKEXC     : PREM; (* EXCLAM *)
2,3,4,5,6    : PERROR; (* 1-CHAR SYMBOLS *)
7,8,9,10,11  : PERROR; (* 1-CHAR SYMBOLS *)
12,13,14     : BEGIN END; (* SCAN ERROR *)
15,16,17,18  : PERROR; (* 2-CHAR SYMBOLS *)

19           : EIF; (* IF *)
20           : PERROR; (* TO **)
21           : PERROR; (* OR *** *)
22           : PSUBERROR; (* ON *)
23           : ELET; (* LET *)
CMTOKREM     : PREM; (* REM *)
25           : EFOR; (* FOR *)
26           : FEND; (* END *)
27           : EDEF; (* DEF *)
28           : PERROR; (* NOT **)
29           : PSUBERROR; (* DIM *)
30           : PERROR; (* AND *** *)
31,32        : PSUBERROR;
33           : PERROR; (* THEN **)
34           : ELSE; (* ELSE *)
35           : PGOTO; (* GOTO *)
36           : FLOOP; (* LOOP *)
37           : ENEXT; (* NEXT *)
38           : FQUIT; (* QUIT *)
39           : PSTOP; (* STOP *)
40           : EDATA; (* DATA *)
41           : EREAD; (* READ *)
42           : PERROR; (* STEP **)
43,44,45     : PSUBERROR;
46           : FENDIF; (* ENDF *)
47           : PFNEND; (* FNEND *)
48           : FGOSUB; (* GOSUB *)
49           : PINPUT; (* INPUT *)
50           : PUNTIL; (* UNTIL *)
51           : EWHILE; (* WHILE *)
52           : FPAUSE; (* PAUSE *)
53           : EPRINT; (* PRINT *)
54,55,56,57,58 : PSUBERROR;

```

```

59      : ELSEIF;          (* ELSEIF *)
60      : RETURN;         (* RETURN *)
61      : FOPTION;        (* OPTION *)
62,63,64,65,66 : PSUBERROR; (* ENDLOOP *)
67      : FENDLOOP;      (* RESTORE *)
68      : FRESTORE;      (* *)
69,70,71,72    : PSUBERROR; (* *)

IDENTOK : FNOLET;        (* NO LET *)
NUMBERTOK : FPEROR;     (* NUMERIC *)
ENDLINTOK : GENKEY (K_NOP); (* BLANK LN *)
ENDFILTOK : GENKEY (K_NOP); (* END FILE *)

```

```

      END (* CASE *)
END

```

```

UNTIL ICKNUM = ENDFILTOK;

```

```

GETFNLS; (* INSERT LONG FN BODIES *)
ENDCP@.SEQP := NEWCCDE (-1); (* CLOSE CODE SEQUENCE *)
LPCUR@.IPTR := GETNEWHDR (MAXBASLIN); (* CLOSE LINE SEQ *)

```

```

FINDGOSUBLEL (BEGINCP); (* INSERT LABELS FOR SER *)
IF OPTFAR THEN (* OPTION 2 *)
  OSQPAREN (BEGINCP); (* OPTIMIZE PARENTHESES *)
IF CEINCP THEN (* OPTION 3 *)
  CSCNCP (BEGINCP); (* OPTIMIZE (OUT) NOP'S *)
RESOLVE_ADDR (BEGINCP); (* OVERLAY ABSOLUTE ADDR *)

```

```

REPORT (LISTF); (* ERROR/REG/LBL SUMMARY *)
REPORT (OUTFILE);
IF CODUMP THEN (* OPTION 4 *)
  CODEDUMP (LISTF, BEGINCP); (* WRITE TRANSLATED CODE *)
IF LINK59 THEN (* CREATE SCRATCH FILE *)
  LINK_INTERFACE; (* FOR LINKER INTERFACE *)

```

```

-----
(* DEBUGGING TOOL: DUMPS EACH SLOT OF EACH BUCKET IN THE *)
(* SYMBOL TABLE TO NAME FILE. *)
IF SYDUMP THEN (* OPTION 5 *)
  SYMBDUMP (LISTF, BUCKET);
-----

```

```

-----
(* DEBUGGING TOOL: DUMPS ENTIRE CODE DATA STRUCTURE *)
(* INCLUDING LINE AND CODE NODES. *)
IF DSDUMP THEN (* OPTION 6 *)
  SEARCH (LISTF, FIRSTLP)
-----

```

```

END; (* EAX59 *)

```

```

*****
*****

```

APPENDIX D

RWTBLF FILE--ORDERED RESERVED WORDS

1	!
2	=
3	+
4	-
5	*
6	/
7	(
8)
9	<
10	>
11	.
12	:
13	#
14	\$
15	%
16	<>
17	=
18	<=
19	IF
20	IC
21	CR
22	CM
23	IF
24	IF
25	IF
26	IF
27	DEF
28	NC
29	DI
30	AND
31	MAT
32	USE
33	THE
34	IF
35	GO
36	TO
37	OF
38	THE
39	ST
40	OF
41	DATA
42	READ
43	ST
44	IF
45	OF
46	END
47	IF
48	END
49	IF
50	UNTIL
51	WHILE
52	UNTIL
53	PRINT
54	CHAIN
55	IN
56	THE
57	IF
58	IF
59	IF

60 RETURN
61 CFTICN
62 LINEUT
63 FEMCVE
64 RENAME
65 RESUME
66 UNLCCK
67 ENDLOOP
68 RESTORE
69 SCRATCH
70 TAGSORT
71 ENDGUESS
72 RANDOMIZE

APPENDIX E

LABEL FILE--TI-59 LABELS/RESERVED REGISTERS

KEY CODES FOR TI-59 LABELS:

11	12	13	14	15	16	17	18	19	10
20	22	23	24	25	27	28	29	30	32
33	34	35	36	37	38	39	42	43	44
45	47	48	49	50	52	53	54	55	57
58	59	60	61	65	66	67	68	69	70
71	75	76	77	78	79	80	81	85	86
87	88	89	90	91	93	94	95	96	97
98	99								

REGISTERS RESERVED BY USER:

00 01 02 03 04 05 06 07 08 09 10

APPENDIX F

BIFNOF/BIFNLF FILES--BUILT-IN FUNCTIONS

BUILT-IN "QUICK" FUNCTION NAMES AND TI-59 KEY CODES:

ABS	50	68	68	68
ACOS	27	39	68	68
ASIN	27	38	68	68
ATN	27	30	68	68
COS	39	68	68	68
COT	30	35	68	68
CSC	38	35	68	68
EXP	22	23	68	68
FP	27	59	68	68
IP	59	68	68	68
LOG	23	68	68	68
LOG10	28	68	68	68
SEC	39	35	68	68
SIN	38	68	68	68
SQR	34	68	68	68
TAN	30	68	68	68

BUILT-IN "LCNG" FUNCTION NAMES AND TI-59 KEY CODE SEQUENCES:

RND 36 15 10 43 10 36 15 15 36 15 71 88 68 68 68

APPENDIX G
 CTEXTF FILE--TI-59 KEYCODE TRANSLATIONS

TI-59 KEY CCDE TEXT.....

-2	0	UNRESOLVED ADER \$\$\$\$
-1	0	UNFILLED CODE \$\$\$\$\$\$
00	0	0
01	0	1
02	0	2
03	0	3
04	0	4
05	0	5
06	0	6
07	0	7
08	0	8
09	0	9
10	0	2ND F'
11	0	A
12	0	B
13	0	C
14	0	D
15	0	E
16	0	2ND A'
17	0	2ND E'
18	0	2ND C'
19	0	2ND D'
20	0	2ND CLR
21	0	2ND \$\$\$\$ ERROR \$\$\$\$
22	0	INV
23	0	LN X
24	0	CF
25	0	CLR
26	0	2ND \$\$\$\$ ERROR \$\$\$\$
27	0	2ND INV
28	0	2ND LCG
29	0	2ND CP
30	0	2ND TAN
31	0	LEN (DEBUGGING TOOL)
32	0	X<=>T
33	0	X**2
34	0	SQRT (X)
35	0	1/X
36	0	2ND FGM
37	0	2ND P=>R
38	0	2ND SIN
39	0	2ND COS
40	0	2ND IND
41	0	SSI \$\$\$\$ ERROR \$\$\$\$
42	1	STC
43	1	RCI
44	1	SUM
45	0	Y**X
46	0	INS \$\$\$\$ ERROR \$\$\$\$
47	0	2ND CMS
48	0	2ND EXC
49	1	2ND PRD
50	0	X
51	0	ESI \$\$\$\$ ERROR \$\$\$\$
52	0	EE
53	0	(
54	0)

55	0	/	DEL	\$\$\$\$	ERROR	\$\$\$\$
56	0		2ND	ENG		
57	0		2ND	FIX		
58	0		2ND	INT		
59	0		2ND	DEG		
60	0		GTC			
61	2		2ND	PGM	2ND	IND
62	1		2ND	EXC	2ND	IND
63	1		2ND	PRD	2ND	IND
64	1					
65	0	*				
66	0		2ND	PAUSE		
67	2		2ND	X=T		
68	0		2ND	NOP		
69	1		2ND	CF		
70	0		2ND	RAD		
71	2		SER			
72	1		SIC	2ND	IND	
73	1		RCI	2ND	IND	
74	1		SUM	2ND	IND	
75	0	-				
76	0		2ND	LBL		
77	2		2ND	X>=T		
78	C		2ND	SUMMATION		
79	0		X-EAR			
80	0		2ND	GRAD		
81	0		RST			
82	0		\$\$\$	VOID CODE	\$\$\$\$	
83	1		GTC	2ND	IND	
84	1		2ND	CP	2ND	IND
85	0	+				
86	1		2ND	STFLG		
87	3		2ND	IFFLG		
88	0		2ND	D.MS		
89	0		2ND	FI		
90	0		2ND	LIST		
91	0		R/S			
92	0		INV	SBR		
93	0	.				
94	0	+/-				
95	C	=				
96	0		2ND	WRITE		
97	3		2ND	ISZ		
98	0		2ND	ADV		
99	0		2ND	FRT		

APPENDIX I
LINKER SOURCE CODE

```

*****
* PURPOSE: THIS PROGRAM TAKES AS INPUT A TI-59 PROGRAM. *
* IT SEGMENTS THE PROGRAM SO THAT IT WILL FIT *
* INTO THE TI-59 CALCULATOR. INSTRUCTIONS AND *
* CODE LISTINGS ARE PROVIDED AS OUTPUT. *
* COMMENT: PROGRAM MAY LOOP INFINITELY IF SMALL; *
* LIMIT IS USED BECAUSE OF DIVIDE *
* ALGORITHM. *
* COMMENT: FILEDEFS WERE USED FOR THIS PROGRAM *
* CONSEQUENTLY THEY WERE NOT DEFINED *
* IN THE PROGRAM. SPECIFIC FILEDEFS *
* FOLLOW: SCRATCH--"SCRATCH PASCAL" *
* PASSED FROM COMPILER *
* OUTFILE--"ANY DESIRED NAME" *
* YOUR OUTPUT FILE *
* TEMPFILE--"ANY DESIRED NAME" *
* A TEMPORARY SCRATCH PAD *
* MESSAGEFILE--MESSAGEFILE FILE *
* LINKER'S MESSAGES *
*****

```

PROGRAM TSDRIVER (INPUT,OUTPUT) ;

```

*****
* DECLARATIONS: *
*****

```

```

(*-----*)
CONST
  FJUMFCONST = 10;      (* NUM STEPS FOR F JUMP CODE *)
  SERCCNST = 15;       (* NUM STEPS FOR SBR BRK CCDE *)
  SERCCNTCONST = 7;    (* NUM STEPS FOR SBR BRK RTN *)

  STO = 42;            (* TI59 KEYCODES *)
  LEL = 76;
  RCLIND = 73;
  STOIND = 72;
  CF = 69;
  DECIMAL = 93;
  RS = 91;
  CE = 24;            (* END KEYCODES *)

  DISPLAYREGSTORE = 00; (* TEMP STORE OF THE DISPLAY *)
  RTNREGNUM = 6;      (* NUMBER OF MANUAL RETURN REGISTERS *)
  MANRINREG = 08;    (* MANUAL SER RETURN REGISTER *)

  NCNE = 101;         (* MESSAGE NUMS *)
  ASTER = 102;       (* SEE MESSAGEFILE FOR TRANSLATION *)
  YES = 103;
  MCDEFMETS = 100;
  RINRGTOP = 104;
  CCDENUM = 0;
  STOINRG = 105;
  PGMEARTIS = 106;
  PARINUMIS = 107;
  MCDN = 108;
  CARD1 = 109;
  CARD2 = 110;

```

AD-A132 172

DDSIGN AND IMPLEMENTATION OF A BASIC CROSS-COMPILER AND
VIRTUAL MEMORY MA. (U) NAVAL POSTGRADUATE SCHOOL
MONTEREY CA M R KINDL ET AL. JUN 83

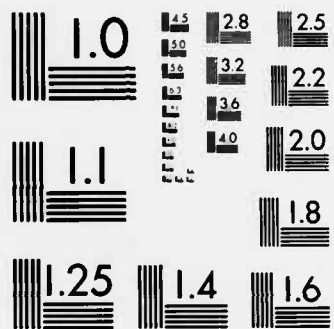
3/4

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```

SIDE1 = 111;
SIDE2 = 112;
EAXINSTR = 81;
SPECIFICS = 82;
ENDLEI = 83;
FAILINSR = 84;
UNSEGCOLLBL = 5;
PSEQ = 6;
PMANRTN = 7;
PFWDJ = 8;
PSBRINV = 9;
REGMAP = 3;
LATAREAD = 4;
RGCI = 1;
ALPHAB1 = 99; (* END MESSAGE NUMS *)
(*-----*)

TYPE
(*-----*)

LABELS = PACKED ARRAY (.1 .. 15.) OF CHAR;
TYPELABELS = ARRAY (.0..99.) OF LABELS; (* T159 KEYS *)
(*-----*)

CODEPTR = @CODERCD; (* THIS RECORD IS BUILT_CODE *)
CODERCD = RECORD
    AADDR:INTEGER;
    MEMNUM:INTEGER;
    RADDF:INTEGER;
    KEY:INTEGER;
    JMP:CCDEPTR;
    SEQ:CODEPTR;
END; (* SHOULD HAVE MADE A VARI *)
(*-----*)

INSTR_SET = SET CF 0 .. 99 ; (* RANGE INSTRUCTION SET*)
(*-----*)

NCDE = (TABLE, SBRPTR, SBRBREAK, FWD_JUMP, MEMODULE, CODE) ;
TBLPTR = @NODES;
NCDES = RECORD
    CASE TAG: NODE OF
        TABLE: (NEST:INTEGER;
            START_ADDR:INTEGER;
            STOP_ADDR:INTEGER;
            LENGTH:INTEGER;
            INCLUDED:BOOLEAN;
            COALESCED:BOOLEAN;
            SBRLIST:TBLPTR;
            NUM_F:INTEGER;
            F_JUMPLIST:TBLPTR;
            TABLELIST:TBLPTR); (* NEXT TABLE *)
        SBRPTR: (SBR:TBLPTR;
            FRM:INTEGER;
            NEXT_SBR:TBLPTR); (* NEXT PTR *)
        MEMODULE: (MEMNUM:INTEGER;
            OFFSET:INTEGER;
            HIGHOFFSET:INTEGER;
            LOWOFFSET:INTEGER;
            RETURNCODE_NEEDED:BOOLEAN;
            SEGTBLS:TBLPTR; (* TABLE *)
            CODELIST:TBLPTR;
            NEXT:TBLPTR);

```

```
CODE: (ADDRESS:INTEGER;
      ABS_ADDR:INTEGER;
      KEYCODE:INTEGER;
      SEQUENTIAL:TBLPTR);
```

```
SBREREA: (SBRZ:TBLPTR);
```

```
PWD_JUMP: (JUMP_ADDRFR:INTEGER;
           JUMP_ADDRTO:INTEGER;
           JUMP_ADDRTO1:INTEGER; (*HUNDREDS*)
           JUMP_ADDRTO2:INTEGER; (*TEN/UNIT*)
           MEM_ADDR:INTEGER; (*MEMNUM*)
           JUMP_INTADDRTO1:INTEGER; (*LOCAL*)
           JUMP_INTADDRTO2:INTEGER;
           NEXT_FJUMP:TBLPTR);
```

```
END;
```

```
(*----- (* SEGMENT TABLE STRUCTURE *)
VAR
CUTFILE:TEXT; (* OUTPUT FILE *)
TEMPFILE:TEXT; (* NEST DIAGS TEMP FILE *)
SCRATCH:TEXT; (* INFORMATION FROM COMPILER FILE *)
MESSAGEFILE:TEXT; (* MESSAGE INPUT FILE *)
(*-----)
PARTIICN:REAL; (* CALCULATOR PARTITION INFO *)
REGCCUNT:INTEGER;
GOOD_SEGMENT:BOOIFAN;
SERINVNEST:INTEGER; (* SBR NEST LEVEL CHECK *)
NUMBANKS,PARI_NUM:INTEGER;
LIMIT:INTEGER; (* MEMORY SIZE LIMIT *)
(*-----)
BUILT_CCDE,CURCP:CODEPTR; (* CODE TBL VARS*)
BUILT_CCDE_COUNT:INTEGER;
HDRPTR,SEG_TBL:TBLPTR; (* TABLE VARS *)
(*-----)
STEP_0,STEP_1,STEP_2,STEP_3:INSTR_SET; (* INS SET VAR *)
(*-----)
TILEL:TYPELABELS; (* PROGRAM LABELS *)
(*-----)
```

```

(*****
* PROCEDURES AND FUNCTIONS:
*****
)

```

```

(=====
* FOLLOWING ROUTINES ARE USED AS UTILITIES SUCH AS PRINT *
* AND SCRATCHFILE AND MESSAGE FILE MANIPULATORS
=====
)

```

```

(-----
* DUMP SEGTEBL: DUMPS THE SEGMENT TABLE. USED FOR DEEUG *
* AND IS NOT CALLED IN THIS PROGRAM.
-----
)

```

```

PROCEDURE DUMP_SEGTEL(VAR OUTFILE:TEXT; HDRPTR:TBLPTR);
VAR F_JMPLINK,SBRLINK,SBR,CURTP,SBRTP:TBLPTR;
BEGIN
  SBRTP:=HDRPTR;
  WHILE SBRTP <> NIL DO
  BEGIN
    WRITELN (OUTFILE);
    WRITELN (OUTFILE);
    WRITELN (OUTFILE, '=====');
    WRITE (OUTFILE, ' SBR CODE NUMBER ');
    WRITELN (OUTFILE, SBRTP@.STOP_ADDR:2);
    WRITELN (OUTFILE, '=====');
    CURTP:=SBRTP@.TABLELIST;
    WHILE CURTP <> NIL DO
    BEGIN
      WITH CURTP@ DO
      BEGIN
        WRITELN (OUTFILE);
        WRITELN (OUTFILE, 'NEST', NEST:3);
        WRITELN (OUTFILE, 'START', START_ADDR:4);
        WRITELN (OUTFILE, 'STOP', STOP_ADDR:4);
        WRITELN (OUTFILE, 'LENGTH', LENGTH:5);
      END;
      SBRLINK:=CURTP@.SERLIST;
      F_JMPLINK:=CURTP@.F_JUMPLIST;
      WHILE (SBRLINK <> NIL) OR (F_JMPLINK <> NIL) DO
      BEGIN
        IF SBRLINK <> NIL THEN
        BEGIN
          SBR:=SBRLINK@.SBR;
          CASE SBR@.TAG OF
            TABLE: SBR:=SBR;
            SBRBREAK:
              BEGIN
                SBR:=SBR@.SBRZ;
                WRITELN(OUTFILE, '*** BREAK ***');
              END;
          END;
          WRITE (OUTFILE, 'SBR INVOKE FROM',
            SBRLINK@.FROM:5, ' TO ',
            SBR@.START_ADDR:5, ' *** ');
          SBRLINK:=SBRLINK@.NEXT_SBR;
        END
        ELSE
          WRITE (OUTFILE, ' ');
        WRITE (OUTFILE, ' *** ');
        IF F_JMPLINK <> NIL THEN
        BEGIN
          WRITE (OUTFILE, 'JUMP FROM ',
            F_JMPLINK@.JUMP_ADDRFR:5, ' TC ',
            F_JMPLINK@.JUMP_ADDRTO:5);
          F_JMPLINK:=F_JMPLINK@.NEXT_FJUMP;
        END;
        WRITELN (OUTFILE);
      END;
    END;
  END;

```

```

        CURIP:=CURTP@.TABLELIST;
        END;
        SBRTIP:=SBRTF@.SBRLIST;
    END;
    END;
    (* DUMP_SEGTBL TEST ROUTINE *)
    -----

    (*-----*)
    (* DUMP MEMODULENODES: PRINTS OUT THE CONTENTS OF THE *)
    (* MEMODULENODE LIST FORMED. THIS IS A DEBUGGING *)
    (* ROUTINE AND IS NOT INVOKED IN THE PROGRAM. *)
    (*-----*)
    PROCEDURE DUMP_MEMODULENODES (HEAD_MEMODULE:TBLPTR);
    VAR S:TBLPTR;
    BEGIN
        S:=HEAD_MEMODULE;
        WHILE S<>NIL DO
            BEGIN
                WITH S@ DO
                    BEGIN
                        WRITELN (OUTFILE);
                        WRITE (OUTFILE, 'MEMNUM OFFSET HIGH LOW');
                        WRITELN (OUTFILE, ' SEGTBLSTART');
                        WRITELN (OUTFILE, MEMNUM:6, OFFSET:8, HIGHOFFSET:6,
                                LOWOFFSET:5, SEGTBL S@.START_ADDR:10);
                        WRITELN (OUTFILE);
                    END;
                S:=S@.NEXT;
            END;
        END;
    END;
    (* DUMP_MEMODULENODES *)
    -----

    (*-----*)
    (* WRITE LEADZERO: PADS INTEGER FIELD WITH LEADING *)
    (* ZEROS *)
    (*-----*)
    PROCEDURE WRITE_LEADZERO (VAR OUTFILE:TEXT; NUM, FLD:INTEGER);
    VAR I, TN:INTEGER;
    BEGIN
        TN:=NUM;
        REPEAT
            TN:=TN DIV 10;
            FLD:=FLD-1;
        UNTIL (TN=0);
        FOR I:=1 TO FLD DO
            WRITE (OUTFILE, '0');
        WRITE (OUTFILE, NUM:1);
    END;
    (* WRITE_LEADZERO *)
    -----

```

```

{ *-----* }
{ * WRITELEL: WRITES OUT THE TI-59 CODED LABELS * }
{ *-----* }
PROCEDURE WRITELBL (VAR OUTFILE:TEXT; CODESS:INTEGER);
  BEGIN
    WRITELN (OUTFILE,TILBL (.CODESS.));
  END;
(*-----*) (* WRITELEL *)

```

```

{ *-----* }
{ * WRITECCDES: WRITES THE ADDRESS AND KEYCODE TO LINE * }
{ *-----* }
PROCEDURE WRITECODES (VAR OUTFILE:TEXT; CUR:CODEPTR);
  BEGIN
    WRITE LEADZERO (OUTFILE,CUR@.AADDR,3);
    WRITE (OUTFILE,');
    WRITE LEADZERO (OUTFILE,CUR@.KEY,2);
    WRITE (OUTFILE,');
  END;
(*-----*) (* WRITECODES *)

```

```

{ *-----* }
{ * WRITENUM: WRITES KEYC DE AS A NUMBER NOT A LABEL * }
{ *-----* }
PROCEDURE WRITENUM (VAR OUTFILE:TEXT; CUR:CODEPTR);
  BEGIN
    WRITE LEADZERO (OUTFILE,CUR@.KEY,2);
  END;
(*-----*) (* WRITENUM *)

```

```

{ *-----* }
{ * HANDLE_#STEPS: PRINTS OUT DIFFERENT CASES OF CODES, * }
{ * EG. WHETHER ONE OR TWO STEP INSTRUCTION. * }
{ * USED FOR CODEPTR TYPE OF NODES. * }
{ *-----* }
PROCEDURE HANDLE_0STEP (VAR OUTFILE:TEXT;
  VAR CUR:CODEPTR);
  BEGIN
    WRITE (OUTFILE,');
    WRITECODES (OUTFILE,CUR);
    WRITE (OUTFILE,');
    WRITELBL (OUTFILE,CUR@.KEY);
  END;

PROCEDURE HANDLE_1STEP (VAR OUTFILE:TEXT;
  VAR CUR:CODEPTR);
  BEGIN
    CUR:=CUR@.SEC;
    WRITE (OUTFILE,');
    WRITECODES (OUTFILE,CUR);
    WRITE (OUTFILE,');
    WRITENUM (OUTFILE,CUR);
    WRITELN (OUTFILE);
  END;
(*-----*) (* HANDLE_1STEP *)

```

```

PROCEDURE HANDLE_2STEP (VAR OUTFILE:TEXT;
                        VAR CUR:CODEPTR);
  VAR I:INTEGER;
  BEGIN
    FOR I:= 1 TO 2 DO
      BEGIN
        CUR:=CUR@.SEQ;
        WRITE (OUTFILE, '                                ');
        WRITECODES (OUTFILE,CUR);
        WRITE (OUTFILE, ' ');
        WRITENUM (OUTFILE,CUR);
        WRITELN (OUTFILE);
      END;
    END;
  (* HANDLE_2STEP *)

PROCEDURE HANDLE_3STEP (VAR OUTFILE:TEXT;
                        VAR CUR:CODEPTR);
  VAR I:INTEGER;
  BEGIN
    FOR I:=1 TO 3 DO
      BEGIN
        CUR:=CUR@.SEQ;
        WRITE (OUTFILE, '                                ');
        WRITECODES (OUTFILE,CUR);
        WRITE (OUTFILE, ' ');
        WRITENUM (OUTFILE,CUR);
        WRITELN (OUTFILE);
      END;
    END;
  (* HANDLE_3STEP *)
(*-----*)

(*-----*)
(* PRINT CODELIST: PRINTS OUT THE TI-59 CODE FOR *)
(* CODEPTR NODES CNLY. *)
(*-----*)
PROCEDURE PRINT_CODELIST (VAR OUTFILE:TEXT;
                          VAR BUILTCODE:CODEPTR);
  VAR CUR:CODEPTR;
  BEGIN
    CUR:=BUILTCODE;
    WHILE CUR <> NIL DO
      BEGIN
        HANDLE_0STEP (OUTFILE,CUR);
        IF CUR@.KEY IN (.71,76.) THEN
          BEGIN
            CUR:=CUR@.SEQ;
            HANDLE_0STEP (OUTFILE,CUR);
          END
        ELSE
          BEGIN
            IF CUR@.KEY IN STEP 1 THEN
              HANDLE_1STEP (OUTFILE,CUR);
            IF CUR@.KEY IN STEP 2 THEN
              HANDLE_2STEP (OUTFILE,CUR);
            IF CUR@.KEY IN STEP 3 THEN
              HANDLE_3STEP (OUTFILE,CUR);
          END;
        IF CUR <> NIL THEN
          CUR:=CUR@.SEQ;
        END;
      END;
    END;
  (* PRINT CODELIST *)
(*-----*)

```

```

{ *-----*
* FIND_MSG: SEARCHES INPUT FILE TO FIND MSG NUMBER. *
*-----*
PROCEDURE FIND_MSG (VAR MESSAGEFILE:TEXT; MSG:INTEGER);
VAR C1:CHAR; DIGIT:INTEGER;
BEGIN
  RESET (MESSAGEFILE);
  C1:=' ';
  DIGIT:=-1;
  REPEAT
    READ (MESSAGEFILE,C1);
    IF C1 = '$' THEN
      READLN (MESSAGEFILE,DIGIT)
    ELSE
      READLN (MESSAGEFILE);
  UNTIL ((C1='$') AND (DIGIT=MSG));
END;
(* FIND_MSG *)
}

```

```

{ *-----*
* INIT_SETS: INITIALIZES IMPORTANT DATA SUCH AS KEY- *
* CCTL LABEL ARRAY, STEP SETS, KEY VARIABLES AND *
* INITIALIZES THE SCRATCH FILE *
*-----*
PROCEDURE INIT_SETS (VAR TEMPFILE:TEXT; VAR STEP_0,STEP_1, *
STEP_2,STEP_3:INSTR SET;
VAR GOOD_SEGMENT:BOCLEAN; VAR MESSAGEFILE:TEXT;
VAR TILBI:TYPELABELS; VAR SBRINVNEST:INTEGER);
VAR C:CHAR;
DIGIT,J,L,K,I:INTEGER;

```

```

{ *-----*
* GET_REGCOUNT: GOES TO SCRATCH FILE AND FINDS THE *
* MESSAGE NUMBER CONTAINING THE REGISTER COUNT *
*-----*
PROCEDURE GET_REGCOUNT (VAR REGCOUNT:INTEGER);
BEGIN
  FIND_MSG (SCRATCH, RGCT);
  READLN (SCRATCH,REGCOUNT);
END;
(* GET_REGCOUNT *)
}

```

```

{ *-----*
*-----*
BEGIN
  SBRINVNEST:=0; (* INITIALIZES THE INVCKE NEST CHECK *)

  RESET (MESSAGEFILE); (* INITIALIZE TILABELS *)
  DIGIT:=-1; (* LABELS IN MESSAGEFILE *)
  L:=1;
  REPEAT
    READ (MESSAGEFILE,C);
    IF C = '$' THEN
      READLN (MESSAGEFILE,DIGIT)
    ELSE
      READLN (MESSAGEFILE);
  UNTIL (C = '$') AND (DIGIT = ALPHALBL);
  L:=0;
  FOR I:=0 TO ALPHALBL DO
    BEGIN
      IF NOT (I IN (.21,26,31,41,46,51,56,82.)) THEN
        BEGIN
          READ (MESSAGEFILE,TILBL (.I.));
          L:=L+1;
          IF L = 4 THEN
            BEGIN
              READLN (MESSAGEFILE);
              L:=0;
            END;
        END;
    END;
END

```

```

ELSE
  TILBL (. I.) := 'BLANK
END;

GET_REGCOUNT (REGCOUNT);

REWRITE (TEMPPFILE);
WRITELN (TEMPPFILE, '39'); (* OPENNING AND MARKING *)
REWRITE (OUTFILE); (* INIT OUTPUTFILE *)
GOOD_SEGMENT := TRUE;

STEP_3 := (. 87, 97.); (* STEP TYPES OF INSTRUCTIONS *)
STEP_2 := (. 61, 67, 77.);
STEP_1 := (. 36, 40, 42, 43, 44, 48, 49, 58, 62, 63, 64, 69, 72, 73, 74,
           . 83, 84, 86.);
STEP_0 := (. 0..99.) - (STEP_3 + STEP_2 + STEP_1);
END; (* INIT_SETS *)
(*-----*)

(*-----*)
(* ADVANCE CODEPTR: MOVES ALONG CODE SKIPPING 1, 2, OR 3*)
(* STEP INSTRUCTIONS AND STOPS ON NEXT COMMAND INSTR. *)
(* TREATS 71 AND 76 AS SINGLE STEPS. *)
(*-----*)
PROCEDURE ADVANCE_CODEPTR (VAR CUR: CODEPTR);
VAR I: INTEGER;
BEGIN
  IF CUR@.KEY IN STEP_3 THEN
    BEGIN
      FOR L:= 1 TO 4 DO
        IF CUR@.SEQ <> NIL THEN
          CUR:=CUR@.SEQ
        END
      END
    END
  ELSE
    IF CUR@.KEY IN STEP_2 THEN
      BEGIN
        FOR L:= 1 TO 3 DO
          IF CUR@.SEQ <> NIL THEN
            CUR:=CUR@.SEQ
          END
        END
      END
    ELSE
      IF CUR@.KEY IN STEP_1 THEN
        BEGIN
          FOR L:= 1 TO 2 DO
            IF CUR@.SEQ <> NIL THEN
              CUR:=CUR@.SEQ
            END
          END
        END
      ELSE
        IF CUR@.SEQ <> NIL THEN
          CUR:=CUR@.SEQ;
        END;
      END;
    END;
  END;
  (* END ADVANCE_CODEPTR *)
(*-----*)

```

```

(*-----*)
(* FRINTLN MSG: PRINTS A SPECIFIC MSG FROM ONE FILE *)
(* TC ANOTHER FILE. THIS ROUTINE WILL TAKE THE *)
(* WHOLE MESSAGE AND PRINT IT. IT EXECUTES A WRITELN *)
(* AT THE END OF THE PRINT *)
(*-----*)
PROCEDURE PRINTLN_MSG (VAR OUTFILE, MESSAGEFILE: TEXT;
                      MSG: INTEGER);
  VAR C1: CHAR;
  BEGIN
    FIND MSG (MESSAGEFILE, MSG);
    READ MESSAGEFILE, C1;
    WHILE C1 <> '$' DO
      BEGIN
        WRITE (OUTFILE, C1);
        WHILE NOT EOLN (MESSAGEFILE) DO
          BEGIN
            READ (MESSAGEFILE, C1);
            WRITE (OUTFILE, C1);
          END;
        READLN (MESSAGEFILE);
        WRITELN (OUTFILE);
        READ (MESSAGEFILE, C1);
      END;
    END;
  END;
  (* PRINTLN_MSG *)
(*-----*)

```

```

(*-----*)
(* PRINT LINE MSG: PRINTS A SPECIFIC ONE-LINE MESSAGE *)
(* TC ANOTHER FILE. DOES NOT WRITELN TO FILE. *)
(* USED FOR LINE LABELS OF GENERATED DATA. *)
(*-----*)
PROCEDURE PRINT_MSGLINE1 (VAR OUTFILE, MESSAGEFILE: TEXT;
                        MSG: INTEGER);
  VAR C1: CHAR;
  BEGIN
    FIND MSG (MESSAGEFILE, MSG);
    READ MESSAGEFILE, C1;
    WHILE C1 <> '$' DO
      BEGIN
        WRITE (OUTFILE, C1);
        READ (MESSAGEFILE, C1);
      END;
    END;
  END;
  (* PRINT_MSGLINE1 *)
(*-----*)

```

```

(*-----*)
(* DET LIMIT: DETERMINES MEMORY LIMITS BASED ON REG CNT. *)
(* ONLY THREE PARTITIONS WERE CONSIDERED. THIS WAS *)
(* BECAUSE ANY OTHER PARTITION SPLITS THE SIDE *)
(* OF A MAG CARD BETWEEN REGISTERS AND PROGRAM. *)
(* THIS WOULD CHANGE REGISTERS DURING REPROGRAMMING *)
(* AND IS THEREFORE UNACCEPTABLE. *)
(*-----*)
PROCEDURE DET_LIMIT (VAR REGCOUNT, LIMIT, NUMBANKS,
                    PART_NUM: INTEGER;
                    VAR PARTITION: REAL);
  BEGIN
    IF REGCOUNT + RTNREGNUM IN (.0..29.) THEN
      BEGIN
        NUMBANKS := 3;
        PARTITION := 719.29;
        PART_NUM := 3;
        LIMIT := 719;
      END
    END
  END

```

```

ELSE
  IF REGCOUNT+RINREGNUM IN (.30..59.) THEN
    BEGIN
      NUMBANKS:=2;
      PARTITION:=479.59;
      PART_NUM:=6;
      LIMIT:=479;
    END
  ELSE
    BEGIN
      NUMBANKS:=1;
      PARTITION:=239.89;
      PART_NUM:=9;
      LIMIT:=239;
    END
  END;
(*-----DET_LIMIT-----*)
(*-----*)
(* CLEAN: REMOVES SAME F JUMPS AND SAME SBRS IN A SEG *)
(* TEL NODE. ALSO GIVES DELETE COUNT FOR F JUMPS *)
(* INCLUDED IN THE CONFINES OF THE SEGMENT. DOUBLE *)
(* DUTY ROUTINE. USED BY COMBINE AND BY SET_LENGTH *)
(*-----*)
PROCEDURE CLEAN(VAR CURTP:TBLPTR; VAR DELETE:INTEGER);
VAR F,S:TBLPTR;
(*-----PRUNE_SAMEF-----*)
(* PRUNE_SAMEF: REMOVES SAME FJUMP ADDRESSTO FROM TEL *)
(*-----*)
PROCEDURE PRUNE_SAMEF(VAR F:TBLPTR);
VAR T,S:TBLPTR;
BEGIN
  WHILE F@.NEXT_FJUMP<>NIL DO
    BEGIN
      S:=F@.NEXT_FJUMP;
      T:=F;
      WHILE (S<>NIL) DO
        IF S@.JUMP_ADDRTO = F@.JUMP_ADDRTO THEN
          BEGIN
            T@.NEXT_FJUMP:=S@.NEXT_FJUMP;
            DISEOSE(S,FWD JUMP);
            S:=T@.NEXT_FJUMP;
          END
        ELSE
          BEGIN
            T:=T@.NEXT_FJUMP;
            S:=S@.NEXT_FJUMP;
          END
        END;
      IF F@.NEXT_FJUMP<>NIL THEN
        F:=F@.NEXT_FJUMP;
      END;
    END;
  END;
(*-----PRUNE_SAMEF-----*)

```

```

(*-----*)
{* PRUNE_GREATOR: REMOVES FJUMPS CONTAINED IN SEGTBL *}
(*-----*)
PROCEDURE PRUNE_GREATOR (VAR F,S:TBLPTR;
                        VAR DELETE:INTEGER);
BEGIN
  WHILE F<>NIL DO
    BEGIN
      IF F@.JUMP_ADDRTO<=CURTP@.STOP_ADDR THEN
        BEGIN
          S@.NEXT_FJUMP:=F@.NEXT_FJUMP;
          DISPOSE(F,FWD_JUMP);
          F:=S@.NEXT_FJUMP;
          DELETE:=DELETE+1;
        END
      ELSE
        BEGIN
          S:=S@.NEXT_FJUMP;
          F:=F@.NEXT_FJUMP;
        END;
      IF CURTP@.F_JUMLIST@.JUMP_ADDRTO<=CURTP@.STOP_ADDR
        THEN
          BEGIN
            F:=CURTP@.F_JUMLIST;
            CURTP@.F_JUMLIST:=F@.NEXT_FJUMP;
            DISPOSE(F,FWD_JUMP);
            DELETE:=DELETE+1;
          END;
        END;
    END;
  END;
  (* PRUNE GREATOR *)
(*-----*)

```

```

(*-----*)
{* PRUNE_SAMES: REMOVES SAME SBR INVOKES FROM SEGTBL *}
(*-----*)
PROCEDURE PRUNE_SAMES (VAR F:TBLPTR);
  VAR S,T,SS,FF:TBLPTR;

  (*-----*)
  {* PASS_BRK: PASSES OVER THE SBR BREAK NODE *}
  (*-----*)
  FUNCTION PASS_BRK (F:TBLPTR):TBLPTR;
  BEGIN
    CASE F@.TAG OF
      SBRBREAK: PASS_BRK:=F@.SBRZ;
      TABLE: PASS_BRK:=F;
    END;
  END;
  (* PASS_BRK *)
  (*-----*)
  BEGIN
    WHILE F<>NIL DO
      BEGIN
        S:=F@.NEXT_SBR;
        IF S<>NIL THEN
          BEGIN
            FF:=PASS_BRK (F@.SBR);
            SS:=PASS_BRK (S@.SBR);
          END;
          T:=F;
          WHILE (S<>NIL) DO
            IF SS = FF THEN
              BEGIN
                T@.NEXT_SBR:=S@.NEXT_SBR;
                DISPOSE(S,SBRPTR);
                S:=T@.NEXT_SBR;
                IF S<>NIL THEN
                  SS:=PASS_BRK (S@.SBR);
                END
              END
            END
          END
        END
      END
    END
  END

```

```

ELSE
  BEGIN
    T:=T@.NEXT_SBR;
    S:=S@.NEXT_SBR;
    IF S<>NIL THEN
      SS:=PASS_BRK(S@.SBR);
    END;
    F:=F@.NEXT_SBR;
  END;
END;
(* PRUNE_SAMESER *)
-----
BEGIN
DELETE:=0;
IF CURTP@.F_JUMLIST<>NIL THEN
  BEGIN
    F:=CURTP@.F_JUMLIST;
    PRUNE_SAMEF(F);
  END;
IF CURTP@.F_JUMLIST<>NIL THEN
  BEGIN
    S:=CURTP@.F_JUMLIST;
    F:=S@.NEXT_FJUMP;
    PRUNE_GREATOR(F,S,DELETE);
  END;
IF CURTP@.SBRLIST<>NIL THEN
  BEGIN
    F:=CURTP@.SERLIST;
    PRUNE_SAMES(F);
  END;
END;
(* CLEAN *)
-----

(*-----*)
(* DIAGS_NEST1SBRBRK: DIAGNOSTIC PRINTOUT IF THERE IS A *)
(* A SER BREAK WITHIN AN ITERATIVE LOOP. NEEDS TO SET *)
(* GOOD_SEGMENT VARIABLE FALSE *)
(*-----*)
PROCEDURE DIAGS_NEST1SBRBRK(VAR TEMPPFILE:TEXT; SEG:TBLPTR;
VAR GOOD_SEGMENT:BOOLEAN);
VAR IS_ERK_BELOW:BOOLEAN;

(*-----*)
(* BELOW_BREAK: SEARCHES OUT BELOW TO SEE IF A BREAK *)
(* IS PRESENT SO DIAGS_NEST1 CAN CHECK FOR A BREAK *)
(* WITHIN A LOOP *)
(*-----*)
PROCEDURE BELOW_BREAK(SEG:TBLPTR; VAR IS_BRK_BELOW:
BOOLEAN);
VAR SER, SBRL:TBLPTR;
BEGIN
  IF NOT IS_BRK_BELOW THEN
    BEGIN
      IF SEG@.SERLIST<>NIL THEN
        BEGIN
          SBRL:=SEG@.SBRLIST;
          WHILE SBRL<>NIL DO
            BEGIN
              SER:=SBRL@.SBR;
              IF SBR@.TAG=SBRBREAK THEN
                IS_BRK_BELOW:=TRUE;
            ELSE
              BELOW_BREAK(SBR,IS_BRK_BELOW);
            SERL:=SBR@.NEXT_SBR;
          END;
        END;
      END;
    END;
  END;
END;
(* BELOW_BREAK *)

```

```

(*-----*)
BEGIN
  IS BRK BELOW:=FALSE;
  BELOW_BREAK(SEG,IS BRK BELOW);
  IF (IS BRK_BELOW) AND (SEG@.NEST=1) THEN
    BEGIN
      GOOD_SEGMENT:=FALSE;
      Writeln(TEMPFILE);
      WRITE(TEMPFILE,');
      Writeln(TEMPFILE,'* SBR BREAK WITHIN A LOOP');
      WRITE(TEMPFILE,');
      Writeln(TEMPFILE,'LOOP BOUNDS',SEG@.START_ADDR:4
        ,' TO ',SEG@.STOP_ADDR:4);
    END;
  END;
(*-----*)
(* DIAGS_NEST1SBRBRK *)

```

```

(*-----*)
(* DIAGS_NEST1LENGTHCHK: PRINTS OUT DIAGNOSTIC IF THERE *)
(* EXISTS AN ITERATIVE LOOP OF TOO GREAT A LENGTH. *)
(* TAKES INTO ACCUNT OUT OF LOOP JUMPS. NEEDS TO *)
(* SET GOOD_SEGMENT FALSE IF ENCOUNTERED *)
(*-----*)
PROCEDURE DIAGS_NEST1LENGTHCHK (VAR TEMPFILE:TEXT;
  CUR:TBLPTR; VAR GOOD_SEGMENT:BOOLEAN);
BEGIN
  IF (CUR@.LENGTH > LIMIT) AND (CUR@.NEST=1) THEN
    BEGIN
      GOOD_SEGMENT:=FALSE;
      Writeln(TEMPFILE);
      WRITE(TEMPFILE,');
      Writeln(TEMPFILE,'* BACK JUMP NEST TOO LONG');
      WRITE(TEMPFILE,');
      Writeln(TEMPFILE,' LOOP BOUNDS',
        CUR@.START_ADDR:4,' TO ',CUR@.STOP_ADDR:4);
    END;
  END;
(*-----*)
(* DIAGS_NEST1SERBRK *)

```

```

{ *-----*
* DIAGS_NEST6SBRINVCHK: CHECKS THAT THE SBR NEST LEVEL *
* DOES NOT EXCEED 6 *
*-----*
PROCEDURE DIAGS_NEST6SBRINVCHK(VAR TEMPFIL:TEXT;
                               CUR:TBLPTR; VAR GOOD_SEGMENT:BOOLEAN;
                               SBRINVEST:INTEGER);
BEGIN
  IF SBRINVEST > 7 THEN
    BEGIN
      GOOD_SEGMENT:=FALSE;
      WRITE(TEMPFIL,' ');
      WRITELN(TEMPFIL,'* SBR INVOKE NEST LEVEL > 6');
      WRITE(TEMPFIL,' ');
      WRITE(TEMPFIL,' CALLES ROUTINE STARTS ');
      WRITELN(TEMPFIL,' AT ABS ADDR ',CUR.START_ADDR:3)
    END;
  END;
(*-----* (* DIAGS_NEST6SBRINVCHK *)

```

```

{ *-----*
* RESET_INCLUDED: SETS ALL INCLUDES TO FALSE. DOES SO *
* FOR ALL ROUTINES ON THE SBRLIST AND BELOW SBRS *
*-----*
PROCEDURE RESET_INCLUDED(VAR SBRL:TBLPTR);
VAR SBRLST,SBR:TBLPTR;
BEGIN
  SBRLST:=SBRL;
  WHILE SBRLST<>NIL DO
    BEGIN
      SER:=SERLIST@.SBR;
      CASE SER@.IAG OF
        TABLE:
          SBR:=SER;
          SBRBREAK;
        SBR:=SER@.SBRZ
      END;
      IF (SBR@.SERLIST<>NIL) AND (SBR@.COALESCED=
          TRUE) THEN
        RESET_INCLUDED(SBR@.SERLIST);
      IF (SBR@.COALESCED=TRUE) THEN
        SBR@.INCLUDED:=FALSE;
        SBRLST:=SBRLST@.NEXT_SBR;
      END;
    END;
  END;
(*-----* (* RSET_INCLUDED *)

```

```

{ *-----*
* INPUT: *
* PURPOSE: TO READ AN INPUT FILE AND FORM SEQ LINKS. *
* THIS FORMS THE INTERNAL CODE STRUCTURE WHICH WILL *
* BE MANIPULATED *
*-----*
PROCEDURE INPUT(VAR SCRATCH:TEXT; VAR BUILT_CODE:CODEPTR;
                VAR BUILT_CODE_COUNT:INTEGER);
VAR ADDRESS:INTEGER;
    TEMP,COUNT:INTEGER;
    CUR,TRAIL:CCDEPTR;
BEGIN
  FIND_MSG(SCRATCH,CODENUM);
  READ(SCRATCH,TEMP);
  IF TEMP > -1 THEN
    BEGIN
      NEW(CUR);
      BUILT_CODE:=CUR;
    END;
  END;

```

```

COUNT:=0;
CUR@.AADDR:=TEMP;
CUR@.RADDR:=COUNT;
READLN(SCRATCH,CUR@.KEY);
TRAIL:=CUR
END; (*IF*)

REPEAT
NEW(CUR);
COUNT:=COUNT+1;
READ(SCRATCH,CUR@.AADDR);
IF CUR@.AADDR<>-1 THEN
BEGIN
CUR@.RADDR:=COUNT;
READLN(SCRATCH,CUR@.KEY);
TRAIL@.SEQ:=CUR;
TRAIL@.JMP:=NIL;
TRAIL:=CUR;
END;
UNTIL (CUR@.AADDR = -1);
BUILT_CODE_COUNT:=COUNT-1;
TRAIL@.JMP:=NIL;
TRAIL@.SEQ:=NIL

END; (* INPUT *)
-----

}
* SETJMP:
* PURPOSE: TO SET THE JUMP POINTER OF THE BUILT CODE
*
}
PROCEDURE SETJMPS (VAR BUILT_CODE:CODEPTR);
VAR CUR:CODEPTR;

}
* SETJMP_PTR: SETS THE JUMPTR OF THE CURRENT NODE
*
}
PROCEDURE SETJMP_PTR (VAR BUILT_CODE:CODEPTR;
CUR:CODEPTR);
VAR MARKER,SEARCH:CODEPTR;
ADDRESS:INTEGER;
BEGIN
MARKER:=CUR@.SEQ;
IF CUR@.KEY IN STEP_3 THEN
MARKER:=MARKER@.SEQ;
ADDRESS:=100*MARKER@.KEY;
ADDRESS:=ADDRESS+MARKER@.SEQ@.KEY;
SEARCH:=BUILT_CODE;
WHILE (SEARCH@.AADDR<>ADDRESS) DO
SEARCH:=SEARCH@.SEQ;
MARKER@.SEQ@.JMP:=SEARCH;
END; (* SETJMPS_PTR *)
}
}
BEGIN
CUR:=BUILT_CODE;
WHILE CUR@.SEQ <> NIL DO
BEGIN
IF CUR@.KEY IN (STEP_2+STEP_3) THEN
SETJMP_PTR(BUILT_CODE,CUR);
IF CUR@.KEY IN (.77,76.) THEN
CUR:=CUR@.SEQ@.SEQ
ELSE
ADVANCE_CODEPTR (CUR);
END;
END;
}
}
END; (* TEST_SETJMP *)
-----

```

```

=====
* BUILD SEGMENT TABLE ROUTINES: ON THIS TABLE ALL *
* OF THE COALESCING IS DONE, AND NOT THE CODE. *
=====
PROCEDURE BUILD_SEG_TBL (BUILT_CODE:CODEPTR;
                        VAR SEG_TBL:TBLPTR; LIMIT:INTEGER;
                        BUILT_CODE_COUNT:INTEGER);

  VAR HDRPTR:TELPTR;

  (*-----*)
  (* BLD_PRIMSEG_TBL: RESULTS IN A TABLE WITH CRITICAL *
  (* POINTS IDENTIFIED. THESE ARE BACK JUMP POINTS *
  (* TO AND FROM LOCATIONS. STOP IS STORED IN STOP *
  (* ADDRESS OF THE FIRST NODE. *
  (*-----*)
  PROCEDURE BLD_PRIMSEG_TBL (BUILT_CODE:CODEPTR;
                          VAR HDRPTR:TBLPTR);

    VAR CURCP:CODEPTR;
        CURTP:TBLPTR;

    (*-----*)
    (* PROCESS_SBRLEL: STORES THE SBRLEL IN THE HEADER *
    (* SBRLIST AND PRODUCES THE FIRST SEGMENT OF THE *
    (* SEGMENT TABLE FOR EACH SBR. *
    (* THIS IS CONFUSING IN THAT THE SAME TYPE OF NODE *
    (* IS USED TO STORE THE LABEL NAME AS IS USED FOR *
    (* THE SEGTABLE. KEY FIELD REDEFINITIONS FOR THIS *
    (* FUNCTION SO THAT THE NAME GOES INTO THE FIELD *
    (* STOP_ADDR. THESE LABEL NODES ARE NEEDED TO BE *
    (* ABLE TO SET THE SBR INVOKE POINTERS LATER ON. *
    (*-----*)
    PROCEDURE PROCESS_SBRLEL (VAR CURCP:CODEPTR;
                            VAR CURTP:TBLPTR);

      VAR TRAILTP:TELPTR;
          BEGIN
            TRAILTP:=CURTP;
            NEW(CURTP, TABLE);
            CURTP@.TAG:=TABLE;
            CURTP@.TABLELIST:=NIL;
            CURTP@.COALESCED:=FALSE;
            CURTP@.INCLUDED:=FALSE;
            CURTP@.SBRLIST:=NIL;
            CURTP@.START_ADDR:=CURCP@.A_ADDR;
            CURTP@.STOP_ADDR:=-1;
            IF CURCP@.KEY = 76 THEN
              BEGIN
                CURTP@.STOP_ADDR:=CURCP@.SEQ@.KEY;
                CURCP:=CURCP@.SEQ@.SEQ;
              END
            ELSE
              CURTP@.STOP_ADDR:=-1;
            IF TRAILTP <> NIL THEN
              TRAILTP@.SBRLIST:=CURTP;
            END;
          (* PROCESS_SBRLEL *)
        (*-----*)

```

```

{*-----*}
{* PROCESS SBRCCDE: PROCESS THE TI-59 SBR CODE FOR *}
{* CRITICAL INFO AND BUILDS THE PRIMITIVE SEGS *}
{*-----*}
PROCEDURE PROCESS_SBRCODE(VAR CURCP:CODEPTR;
                          VAR SBRHDRTP:TBLPTR);
  VAR TOPTP,CURTP:TBLPTR;

  {*-----*}
  {* IS_BCK_JMP: DETERMINES IF THE JUMP IS BACKWARDS *}
  {*-----*}
  FUNCTION IS_BCK_JMP(CURCP:CODEPTR):BOOLEAN;
    VAR ADDRESS:INTEGER;
    BEGIN
      IF (CURCP@.KEY IN STEP_2) THEN
        BEGIN
          ADDRESS:=CURCP@.SEQ@.SEQ@.JMP@.AADDR;
          IF ADDRESS > CURCP@.AADDR THEN
            IS_BCK_JMP:=FALSE
          ELSE
            IS_BCK_JMP:=TRUE;
          END
        ELSE
          IF (CURCP@.KEY IN STEP_3) THEN
            BEGIN
              ADDRESS:=CURCP@.SEQ@.SEQ@.SEQ@.JMP@.AADDR;
              IF ADDRESS > CURCP@.AADDR THEN
                IS_BCK_JMP:=FALSE
              ELSE
                IS_BCK_JMP:=TRUE;
              END
            ELSE
              IS_BCK_JMP:=FALSE;
            END;
          END;
        END;
      END;
    END;
  END;
  (* IS_BCK_JMP *)
  (*-----*)

```

```

{*-----*}
{* APND JMP_TBL: DETERMINES ALL OUT OF CODE JUMPS *}
{* FROM A "FROM" ADDRESS TO A "TO" ADDRESS. *}
{*-----*}
PROCEDURE APND_JMP_TBL(CURCP:CODEPTR;
                      VAR TOPTP:TBLPTR);
  VAR ADDRESSFR,ADDRESSTO:INTEGER;

  {*-----*}
  {* INSERT CRITS: PLACES CRITICALS IN SEGTBL. *}
  {* CRITICAL IS ADDRESS WHERE A BACK JUMP NEST *}
  {* LEVEL CHANGE TAKES PLACE, IE START OR STOP. *}
  {*-----*}
  PROCEDURE INSERT_CRITS(ADDRESS:INTEGER;
                        VAR TOPTP:TBLPTR);
    VAR CURTP,TRAILTP,INSERTTP:TBLPTR;
    BEGIN
      TRAILTP:=TOPTP;
      CURTP:=TCPTP@.TABLELIST;
      WHILE (CURTP@.START_ADDR < ADDRESS) AND
            (CURTP@.TABLELIST <> NIL) DO
        BEGIN
          TRAILTP:=CURTP;
          CURTP:=CURTP@.TABLELIST;
        END;
      NEW(INSERTTP,TABLE);
      INSERTTP@.TAG:=TABLE;
      INSERTTP@.START_ADDR:=ADDRESS;
      INSERTTP@.STOP_ADDR:=-2;
      IF (CURTP@.TABLELIST=NIL) AND
        (CURTP@.START_ADDR < ADDRESS) THEN

```

```

BEGIN
  CURTP@.TABLELIST:=INSERTTP;
  INSERTTP@.TABLELIST:=NIL
END
ELSE
IF (CURTP@.TABLELIST=NIL) AND
  (CURTP@.START_ADDR > ADDRESS) THEN
  BEGIN
  TRAILTP@.TABLELIST:=INSERTTP;
  INSERTTP@.TABLELIST:=CURTP
  END
ELSE
IF CURTP@.START_ADDR > ADDRESS THEN
  BEGIN
  INSERTTP@.TABLELIST:=CURTP;
  TRAILTP@.TABLELIST:=INSERTTP
  END
ELSE
DISPOSE(INSERTTP, TABLE);
END;
(* INSERT_CRITS *)
-----*)

(*-----*)
(* SET NESTS: SEARCHES THE PRIM SEGTBL AND MARKS *)
(* AS 1 ALL OVERLAPPING BACK JUMPS TO DESIGNATE *)
(* THAT THEY ARE IN A NO BREAK AREA *)
(*-----*)
PROCEDURE SET_NESTS (ADDRESSFR, ADDRESSSTO:INTEGER;
  VAR TOPTP:TBLPTR);
  VAR CURTP:TELPTR;
  BEGIN
  CURTP:=TCPTP;
  WHILE CURTP@.START_ADDR <> ADDRESSSTO DO
  CURTP:=CURTP@.TABLELIST;
  WHILE CURTP@.START_ADDR <> ADDRESSFR DO
  BEGIN
  CURTP@.NEST:= 1;
  CURTP:=CURTP@.TABLELIST
  END;
  CURTP@.NEST:=0;
  IF ((CURTP@.TABLELIST <> NIL) AND (CURTP@.NEST =
  1)) THEN
  CURTP@.NEST:= 1
  END;
  (* SET_NESTS *)
  -----*)
BEGIN
IF CURCP@.KEY IN STEP_2 THEN
  BEGIN
  ADDRESSSTO:=CURCP@.SEQ@.SEQ@.JMP@.AADDR;
  ADDRESSFR:=CURCP@.SEQ@.SEQ@.AADDR
  END
ELSE
  BEGIN
  ADDRESSSTO:=CURCP@.SEQ@.SEQ@.SEQ@.JMP@.AADDR;
  ADDRESSFR:=CURCP@.SEQ@.SEQ@.SEQ@.AADDR
  END;
  INSERT_CRIS(ADDRESSFR, TOPTP);
  INSERT_CRIS(ADDRESSSTO, TOPTP);
  SET_NESTS(ADDRESSFR, ADDRESSSTO, TOPTP)
END;
(* APND_JMP_TEL *)
-----*)
BEGIN
NEW(CURTP, TABLE);
CURTP@.TAG:=TABLE;
CURTP@.START_ADDR:=SBRHDRTP@.START_ADDR;
CURTP@.NEST:=0;
CURTP@.COALESCED:=FALSE;

```

```

CURTP@.INCLUDED:=FALSE;
CURTP@.TABLELIST:=NIL;
CURTP@.SERLIST:=NIL;
SERHDRTP@.TABLELIST:=CURTP;
TCPTP:=SERHDRTP;
WHILE ((CURCF@.KEY <> 76) AND (CURCP@.SEQ <> NIL ))
  DC
  IF IS_BACK_JMP (CURFCP) THEN
    BEGIN
      APND JMP TBL (CURCP, TOPTP);
      ADVANCE_CODEPTR (CURCP)
    END
  ELSE
    ADVANCE_CCDEPTR (CURCP);
  IF CURCP@.SEQ = NIL THEN
    TOPTP@.TABLELIST@.STOP_ADDR:=CURCP@.AADDR
  ELSE
    TOPTP@.TABLELIST@.STOP_ADDR:=CURCP@.AADDR-1;
END;
(*-----*)
BEGIN
CURCP:=BUILT_CODE;
CURTP:=NIL;
PROCESS_SBRLBL (CURCP, CURTP);
HDRPTR:=CURTF;
WHILE (CURCP@.SEQ <> NIL) DO
  BEGIN
    PROCESS_SERCODE (CURCP, CURTP);
    IF CURCP@.KEY = 76 THEN
      PROCESS_SBRLBL (CURCP, CURTP);
  END;
END;
(*-----*)

```

```

(*-----*)
{ * BLD ADVSEGTEL: FILLS IN THE STOPS AND MERGES SAME *
  * NESTED LEVELS INTO ONE SEGMENT. *
  * STOPS ARE STOP_ADDR FIELD *
}
(*-----*)

```

```

PROCEDURE BLD ADVSEGTEL (VAR HDRPTR:TBLPTR);
VAR SBRTP:TBLPTR; STOP:INTEGER;

```

```

(*-----*)
{ * MERGE_ONES: COMBINES SAME NESTED ADJACENT 1 SEGS *
}
(*-----*)
PROCEDURE MERGE_ONES (VAR SBRTP:TBLPTR);
VAR MARK,ZERO,ONE:TBLPTR;

```

```

(*-----*)
{ * MERGE: DOES ACTUAL MERGING OF ADJACENT SEGMENTS *
}
(*-----*)

```

```

PROCEDURE MERGE (VAR ONE,ZERO,MARK:TBLPTR);
VAR DIS:TBLPTR;
BEGIN
  ONE@.STOP_ADDR:=ZERO@.START_ADDR;
  DIS:=ONE@.TABLELIST;
  WHILE ONE@.TABLELIST <> ZERO DO
    BEGIN
      ONE@.TABLELIST:=DIS@.TABLELIST;
      DISPCSE (DIS, TABLE);
      DIS:=ONE@.TABLELIST;
    END;
  IF ZERO@.TABLELIST <> NIL THEN
    ONE@.TABLELIST:=ZERO@.TABLELIST
  ELSE
    ONE@.TABLELIST:=NIL;
  MARK:=ONE;

```

```

DISPOSE (LIS, TABLE);
END; (* MERGE *)
(*-----*)
BEGIN
MARK:=SBRTF@.TABLELIST;
WHILE MARK@.TABLELIST <> NIL DO
BEGIN
IF (MARK@.NEST = 0) AND (MARK@.TABLELIST<>NIL)
THEN MARK:=MARK@.TABLELIST;
ONE:=MARK;
WHILE (MARK@.NEST=1) AND (MARK@.TABLELIST<>NIL)
DO MARK:=MARK@.TABLELIST;
ZERO:=MARK;
IF ONE <> ZERO THEN
BEGIN
MERGE(ONE, ZERO, MARK);
MARK:=ONE;
END;
IF MARK@.TABLELIST <> NIL THEN
MARK:=MARK@.TABLELIST;
END;
END; (* MERGE_ONES *)
(*-----*)

(*-----*)
(* ADD_ZEROS: FILLS IN GAPS IN TABLE WITH 0 SEG *)
(*-----*)
PROCEDURE ADD ZEROS (VAR SBRTF: TBLPTR);
VAR CUR, TRAIL, INSERT: TBLPTR; STOP: INTEGER;
BEGIN
TRAIL:=SBRTF;
CUR:=SERTP@.TABLELIST;
STOP:=CUR@.STOP_ADDR;
WHILE CUR@.TABLELIST <> NIL DO
BEGIN
CUR:=CUR@.TABLELIST;
WHILE TRAIL@.TABLELIST <> CUR DO
TRAIL:=TRAIL@.TABLELIST;
IF TRAIL@.NEST<> CUR@.NEST THEN
TRAIL@.STOP_ADDR:=CUR@.START_ADDR-1
ELSE
BEGIN
NEW (INSERT, TABLE);
INSERT@.NEST:=0;
INSERT@.START_ADDR:=TRAIL@.STOP_ADDR+1;
INSERT@.STOP_ADDR:=CUR@.START_ADDR-1;
INSERT@.TABLELIST:=CUR;
TRAIL@.TABLELIST:=INSERT
END;
END;
IF CUR@.STOP_ADDR <> STOP THEN
BEGIN
NEW (INSERT, TABLE);
INSERT@.NEST:=0;
INSERT@.START_ADDR:=CUR@.STOP_ADDR+1;
INSERT@.STOP_ADDR:=STOP;
INSERT@.TABLELIST:=NIL;
CUR@.TABLELIST:=INSERT
END;
END;
END; (* ADD_ZEROS *)
(*-----*)
BEGIN
SERTP:=HDRPTR;
WHILE SBRTF <> NIL DO
BEGIN
MERGE_ONES (SERTP);
ADD_ZEROS (SBRTF);

```

```

        SBRTP:=SERTP@.SBRLIST;
    END;
END; (* BLD_ADVSEGTEL *)
(*-----*)

(*-----*)
(* BLD_FINSEGTEL: PROCESS CODE FOR SBR INVOKES AND *)
(* FJUMPS. WHEN ENCOUNTERED IT PLACES INTO SEGTEL. *)
(* THESE WILL INCLUDE ONLY ONE INVOKE PER SEGMENT *)
(* AND ONLY ONE FJUMP TO SAME LOCATIONS. REPEATS *)
(* WILL BE IGNORED. LENGTHS OF SEGMENTS WILL ALSO *)
(* BE CALCULATED. LENGTHS DO NOT INCLUDE CODE FOR *)
(* SBR INVOKES/EROMPT CODE. ONLY SEQUENTIAL CONTIN- *)
(* UATION CODE IS INCLUDED IN LENGTH CALCULATION TO *)
(* GETHER WITH FJUMP EROMPT CODE. *)
(*-----*)
PROCEDURE BLD_FINSEGTEL (BUILTCODE:CODEPTR;
                        VAR HDRPTR:TBLPTR; LIMIT:INTEGER);
    VAR CURCP:CODEPTR;
        SERTP:TBLPTR;

    (*-----*)
    (* PROCESS SBRSEGTEL: PLACES SBRS & FJMP INTO SEGTEL *)
    (*-----*)
    PROCEDURE PROCESS_SBRSEGTEL (VAR CURCP:CODEPTR;
                                VAR HDRPTR,SBRTP:TBLPTR);
        VAR CURTP,SBRINVOKE,FJMP:TBLPTR;

        (*-----*)
        (* HANDLE_FWDJMP: INSERTS FWD JUMPS INTO TABLE. *)
        (*-----*)
        PROCEDURE HANDLE_FWDJMP (CURCP:CODEPTR;
                                VAR HDRPTR,CURTP,FJMP:TBLPTR);
            VAR ADDRESSSTO,ADDRESSFR:INTEGER;
                INSERT:TBLPTR;
            BEGIN
                IF CURCP@.KEY IN STEP_3 THEN
                    BEGIN
                        ADDRESSFR:=CURCP@.SEQ@.SEQ@.SEQ@.AADR;
                        ADDRESSSTO:=CURCP@.SEQ@.SEQ@.SEQ@.JMP@.AADR;
                    END
                ELSE
                    BEGIN
                        ADDRESSFR:=CURCP@.SEQ@.SEQ@.AADR;
                        ADDRESSSTO:=CURCP@.SEQ@.SEQ@.JMP@.AADR;
                    END;
                NEW (INSERT,FWD JUMP);
                INSERT@.TAG:=FWD JUMP;
                INSERT@.JUMP_ADDRFR:=ADDRESSFR;
                INSERT@.JUMP_ADDRTO:=ADDRESSSTO;
                INSERT@.JUMP_ADDRTO1:=-1;
                INSERT@.JUMP_ADDRTO2:=-2;
                INSERT@.MEM_ADDR:=-1;
                INSERT@.JUMP_INTADDRTO1:=-1;
                INSERT@.JUMP_INTADDRTO2:=-2;
                INSERT@.NEXT_FJUMP:=NIL;
                IF CURTP@.F_JUMPLIST = NIL THEN
                    BEGIN
                        CURTP@.F_JUMPLIST:=INSERT;
                        FJMP:=INSERT;
                    END
                ELSE
                    BEGIN
                        FJMP@.NEXT_FJUMP:=INSERT;
                        FJMP:=INSERT
                    END;
            END;
        END;
    END;
END; (* HANDLE_FWDJMP *)
(*-----*)

```

```

(*-----*)
{* IS_FWD_JMP: BOOLEAN TRUE IF KEYCODE IS FWD JMP *}
(*-----*)
FUNCTION IS_FWD_JMP (CURCP:CODEPTR) : BOOLEAN;
VAR ADDRESS: INTEGER;
BEGIN
  IF (CURCP@.KEY IN STEP_2) THEN
    BEGIN
      ADDRESS:=CURCP@.SEQ@.SEQ@.JMP@.AADDR;
      IF ADDRESS > CURCP@.AADDR THEN
        IS_FWD_JMP:=TRUE
      ELSE
        IS_FWD_JMP:=FALSE;
      END
    END
  ELSE
    IF (CURCP@.KEY IN STEP_3) THEN
      BEGIN
        ADDRESS:=CURCP@.SEQ@.SEQ@.SEQ@.JMP@.AADDR;
        IF ADDRESS > CURCP@.AADDR THEN
          IS_FWD_JMP:=TRUE
        ELSE
          IS_FWD_JMP:=FALSE;
        END
      END
    ELSE
      IS_FWD_JMP:=FALSE;
    END;
  END;
(* IS_FWD_JMP *)
(*-----*)

```

```

(*-----*)
{* HANDLE_SBRINVOKE: PLACES SBR CALL INTO TABLE *}
(*-----*)
PROCEDURE HANDLE_SBRINVOKE (VAR CURCP: CODEPTR;
  VAR HDRPTR, CURTP, SBRINVOKE: TBLPTR);
VAR TOSBR, INSERT: TBLPTR;
  KEY: INTEGER;
BEGIN
  TOSBR:=HDRPTR;
  KEY:=CURCP@.SEQ@.KEY;
  WHILE TOSBR@.STOP_ADDR<> KEY DO
    TOSBR:=TOSBR@.SERLIST;
  NEW (INSERT, SBRPTR);
  INSERT@.TAG:=SBRPTR;
  INSERT@.FROM:=CURCP@.AADDR+1;
  INSERT@.SER:=TOSBR@.TABLELIST;
  INSERT@.NEXT_SBR:=NIL;
  IF CURTP@.SERLIST = NIL THEN
    BEGIN
      CURTP@.SERLIST:=INSERT;
      SBRINVOKE:=INSERT
    END
  ELSE
    BEGIN
      SBRINVOKE@.NEXT_SBR:=INSERT;
      SBRINVOKE:=INSERT;
    END;
  END;
END;
(* HANDLE_SBRINVOKE *)
(*-----*)

```

```

BEGIN
  CURTP:=SBRTP@.TABLELIST;
  REPEAT
    CURTP@.SERLIST:=NIL;
    CURTP@.F_JUMPLIST:=NIL;
    SBRINVOKE:=NIL;
    FJMP:=NIL;
    WHILE CURCP@.AADDR < CURTP@.STOP_ADDR DO
      BEGIN
        IF CURCP@.KEY = 71 THEN

```

```

        HANDLE_SBRINVOKE(CURCP, HDRPTR, CURTP,
                          SBRINVOKE);
    IF IS_FWD_JMP(CURCP) THEN
        HANDLE_FWDJMP(CURCP, HDRPTR, CURTP, FJMP);
    IF (CURCP@.KEY=76) OR (CURCP@.KEY=71) THEN
        CURCP:=CURCP@.SEQ@.SEQ
    ELSE
        ADVANCE_CODEPTR(CURCP);
    END;
    CURTP:=CURTP@.TABLELIST
UNTIL (CURTF = NIL);
END;
(* PROCESS_SBRSEGTEL *)
(*-----*)

```

```

(*-----*)
(* SET LENGTH: ENSURES LENGTH IS WITHIN MEMORY LIMIT*)
(* IF NOT WILL DIVIDE THE SEGMENT IN HALF AND *)
(* RESET ALL SBRLISTS AND FJUMPLISTS THEN CONTINUE*)
(* NOTE: MAY LEAD TO PROBLEMS IS LIMIT IS *)
(* ARBITRARILY SMALL. *)
(*-----*)

```

```

PROCEDURE SET_LENGTH(BUILT_CODE:CODEPTR;
                    VAR SBRTPT:TBLPTR; LIMIT:INTEGER);
    VAR CURTP:TBLPTR;
        LENGTH,DELETE,L_POSSBR:INTEGER;

```

```

(*-----*)
(* CALCULATE: DETERMINES LENGTH OF A SEGMENT. WILL*)
(* NOT ADD ADDITIONAL STEPS FOR DUPLICATE FJMP *)
(* ADDRTO *)
(*-----*)

```

```

PROCEDURE CALCULATE(CURTP:TBLPTR;
                   VAR LENGTH:INTEGER);
    VAR S,F:TBLPTR;
        ADDITIONS:INTEGER;
    BEGIN
        LENGTH:=CURTP@.STOP_ADDR-CURTP@.START_ADDR
                +FJUMPCONST+1;
        IF CURTP@.F_JUMPLIST<>NIL THEN
            BEGIN
                ADDITIONS:=0;
                F:=CURTP@.F_JUMPLIST;
                IF F@.JUMP_ADDRTO > CURTP@.STOP_ADDR THEN
                    ADDITIONS:=1;
                S:=F;
                F:=F@.NEXT_FJUMP;
                WHILE F<>NIL DO
                    BEGIN
                        IF F@.JUMP_ADDRTO>CURTP@.STOP_ADDR THEN
                            BEGIN
                                ADDITIONS:=ADDITIONS+1;
                                S:=CURTP@.F_JUMPLIST;
                                WHILE ((S<>F) AND (S@.JUMP_ADDRTO<>
                                                                F@.JUMP_ADDRTO)) DO
                                    S:=S@.NEXT_FJUMP;
                                IF S<>F THEN
                                    ADDITIONS:=ADDITIONS-1;
                                END;
                                F:=F@.NEXT_FJUMP;
                            END;
                        S:=S@.NEXT_FJUMP;
                    END;
                LENGTH:=LENGTH+(ADDITIONS)*(FJUMPCONST);
            END;
        END;
    END;
(* CALCULATE *)
(*-----*)

```

```

(*-----*)
(* L_POSSBRK: CALCULATES ANY SBR INVOKES AS A *)
(* POSSIBLE BREAK FOR DIVISION PURPOSES. DOES *)
(* NOT INCLUDE MULTIPLE INVOKES OF SAME SBR *)
(*-----*)
PROCEDURE LENGTH_SBRBRKS (CURTP:TBLPTR;
                          VAR L_POSSBR:INTEGER);
VAR F,T:TBLPTR; COUNT:INTEGER;
BEGIN
  COUNT:=0;
  IF CURTP@.SBRLIST<>NIL THEN
    BEGIN
      F:=CURTP@.SBRLIST;
      WHILE F<>NIL DO
        BEGIN
          IF NOT (F@.SBR@.INCLUDED) THEN
            BEGIN
              COUNT:=COUNT+1;
              F@.SBR@.INCLUDED:=TRUE;
            END;
          F:=F@.NEXT_SBR;
        END;
      F:=CURTP@.SBRLIST;
      WHILE F<>NIL DO
        BEGIN
          F@.SBR@.INCLUDED:=FALSE;
          F:=F@.NEXT_SBR;
        END;
      END;
      L_POSSBR:=COUNT*SBRCONST;
    END;
  (* L_SBRBRKS *)
(*-----*)

```

```

-----*
* DIVIDE: DIVIDES A SEG IN HALF AND RESETS FJMP *
* AND SBR POINTERS. *
-----*

```

```

PROCEDURE DIVIDE(BUILT CODE:CODEPTR;
                 VAR CURTP:TBLPTR);

```

```

  VAR INSERT:TBLPTR;
      S,F:TBLPTR;
      NEW_STOP:INTEGER;

```

```

-----*
* SETT: ENSURES THAT DIVIDE CALCULATED STOP IS *
* NOT SPLITTING A 1,2,3 PART INSTRUCTION. *
-----*

```

```

PROCEDURE SETT(BUILT CODE:CODEPTR;
               VAR NEW_STOP:INTEGER);

```

```

  VAR P,T:CCDEPTR;
  BEGIN

```

```

    P:=BUILT CODE;

```

```

    WHILE (P@.AADDR <= NEW_STOP) DO

```

```

      BEGIN

```

```

        T:=P;

```

```

        IF (P@.KEY = 76) OR (P@.KEY = 71) THEN

```

```

          F:=P@.SEQ@.SEQ

```

```

        ELSE

```

```

          ADVANCE_CODEPTR(P);

```

```

        END;

```

```

        IF (T@.KEY = 76) OR (T@.KEY = 71) THEN

```

```

          T:=T@.SEQ@.SEQ;

```

```

          NEW_STOP:=T@.AADDR-1;

```

```

        END;

```

```

(* SETT *)
-----*

```

```

-----*
* FIND INSSBRLIST: DIVIDES UP THE SBRLIST *
* BETWEEN THE OLD AND NEW SEGMENTS *
-----*

```

```

PROCEDURE FIND_INSSBRLIST(VAR CURTP,S:TBLPTR;
                           NEW_STOP:INTEGER);

```

```

  VAR LIMIT:INTEGER;

```

```

      T:TBLPTR;

```

```

  BEGIN

```

```

    LIMIT:=NEW_STOP+1;

```

```

    IF CURTP@.SBRLIST<>NIL THEN

```

```

      BEGIN

```

```

        S:=CURTP@.SBRLIST;

```

```

        IF (S@.NEXT_SBR<>NIL) AND (S@.FROM
                                     < LIMIT) THEN

```

```

          BEGIN

```

```

            S:=S@.NEXT_SBR;

```

```

            T:=CURTP@.SBRLIST;

```

```

            WHILE (S@.FROM<LIMIT) AND

```

```

                  { S@.NEXT_SBR<>NIL) DO

```

```

              BEGIN

```

```

                S:=S@.NEXT_SBR;

```

```

                T:=T@.NEXT_SBR;

```

```

              END;

```

```

            IF S@.FROM >=LIMIT THEN

```

```

              T@.NEXT_SBR:=NIL

```

```

            ELSE

```

```

              S:=NIL

```

```

          END

```

```

        ELSE

```

```

          BEGIN

```

```

            IF S@.FROM >=LIMIT THEN

```

```

              CURTP@.SBRLIST:=NIL

```

```

            ELSE

```

```

              S:=NIL

```

```

        END;
    ELSE
        S:=NIL;
    END;
    (* FIND_INSSBRLIST *)
    -----*)

    (* FIND_INSFJUMPLIST: DIVIDES FJUMPLIST BETWEEN
    * OLD AND NEW SEGMENTS.
    *)
    PROCEDURE FIND_INSFJUMPLIST (VAR CURTP,F:TBLPTR;
    NEW_STOP:INTEGER);
    VAR LIMIT:INTEGER;
        T:TBLPTR;
    BEGIN
        LIMIT:=NEW_STOP+1;
        IF CURTP@.F_JUMPLIST<> NIL THEN
            BEGIN
                F:=CURTP@.F_JUMPLIST;
                IF (F@.NEXT_FJUMP<>NIL) AND (F@.JUMP_ADDRFR
                < LIMIT) THEN
                    BEGIN
                        F:=F@.NEXT_FJUMP;
                        T:=CURTP@.F_JUMPLIST;
                        WHILE (F@.JUMP_ADDRFR<LIMIT) AND
                        (F@.NEXT_FJUMP<>NIL) DO
                            BEGIN
                                F:=F@.NEXT_FJUMP;
                                T:=T@.NEXT_FJUMP;
                            END;
                        IF F@.JUMP_ADDRFR >=LIMIT THEN
                            T@.NEXT_FJUMP:=NIL
                        ELSE
                            F:=NIL
                        END
                    END
                ELSE
                    BEGIN
                        IF F@.JUMP_ADDRFR >= LIMIT THEN
                            CURTP@.F_JUMPLIST:=NIL
                        ELSE
                            F:=NIL
                        END;
                    END
                END
            END
        ELSE
            F:=NIL;
        END;
    (* FIND_INSFJUMPLIST *)
    -----*)
    BEGIN
        NEW_STOP:=(((CURTP@.STOP_ADDR-CURTP@.START_ADDR
        +1)DIV 2))+CURTP@.START_ADDR;
        SETT (BUILT_CODE,NEW_STOP);
        NEW (INSERT,TABLE);
        INSERT@.NEST:=CURTP@.NEST;
        INSERT@.START_ADDR:=NEW_STOP+1;
        INSERT@.STOP_ADDR:=CURTP@.STOP_ADDR;
        FIND_INSSERLIST (CURTP,S,NEW_STOP);
        FIND_INSFJUMPLIST (CURTP,F,NEW_STOP);
        INSERT@.SERLIST:=S;
        INSERT@.F_JUMPLIST:=F;
        INSERT@.TABLELIST:=CURTP@.TABLELIST;
        CURTP@.TABLELIST:=INSERT;
        CURTP@.STOP_ADDR:=NEW_STOP;
    END;
    (* DIVIDE *)
    -----*)
    BEGIN
        CURTP:=SBRTF@.TABLELIST;
        REPEAT

```

```

CALCULATE (CURTP, LENGTH);
IF (CURTP@.NEST = 0) THEN
  BEGIN
    LENGTH SBRBRKS (CURTP, L_POSSBR);
    IF LENGTH+L_POSSBR > LIMIT THEN
      DIVIDE (BUILT_CODE, CURTP)
    ELSE
      BEGIN
        CLEAN (CURTP, DELETE);
        DELETE:=0; (*CALCULATE HAS THIS COVERED*)
        CURTP@.LENGTH:=LENGTH-DELETE*FJUMPCONST;
        IF CURTP@.TABLELIST=NIL THEN
          CURTP@.LENGTH:=CURTP@.LENGTH-FJUMPCONST
          +SBRCONST;
        CURTP:=CURTP@.TABLELIST;
      END
    END
  ELSE
    BEGIN
      CLEAN (CURTP, DELETE);
      DELETE:=0; (*CALCULATE HAS THIS COVERED *)
      CURTP@.LENGTH:=LENGTH-DELETE*FJUMPCONST;
      CURTP:=CURTP@.TABLELIST
    END;
  UNTIL (CURTP = NIL);
END; (* SET_LENGTH *)
-----
BEGIN
  CURCP:=BUILTCCODE;
  SBRTIP:=HDRPTR;
  WHILE SBRTIP <> NIL DO
    BEGIN
      PROCESS_SERSEGTBL (CURCP, HDRPTR, SBRTIP);
      SET_LENGTH (BUILT_CODE, SBRTIP, LIMIT);
      SBRTIP:=SBRTIP@.SBRLIST
    END;
  END; (* BLD_FINSEGTBL *)
-----
BEGIN
  IF BUILT_CODE_COUNT<=LIMIT THEN
    BEGIN
      NEW (SEGTBL, TABLE);
      SEGTBL@.TAG:=TABLE;
      SEGTBL@.TABLELIST:=NIL;
      SEGTBL@.SBRLIST:=NIL;
      SEGTBL@.COALESCE:=TRUE;
      SEGTBL@.INCLUDED:=FALSE;
      SEGTBL@.START_ADDR:=0;
      SEGTBL@.STOP_ADDR:=BUILT_CODE@.SEQ@.KEY;
      NEW (HDRPTR, TABLE);
      HDRPTR@.START_ADDR:=0;
      HDRPTR@.STOP_ADDR:=BUILT_CODE_COUNT;
      HDRPTR@.NEST:=0;
      HDRPTR@.TABLELIST:=NIL;
      HDRPTR@.SBRLIST:=NIL;
      HDRPTR@.F_JUMLIST:=NIL;
      HDRPTR@.COALESCE:=TRUE;
      HDRPTR@.INCLUDED:=FALSE;
      HDRPTR@.LENGTH:=BUILT_CODE_COUNT+1;
      SEGTBL@.TABLELIST:=HDRPTR;
    END
  ELSE
    BEGIN
      BLD_PRIMSEGTBL (BUILT_CODE, HDRPTR);
      BLD_ADVSEGTBL (HDRPTR);
      BLD_FINSEGTBL (BUILT_CODE, HDRPTR, LIMIT);
      SEGTBL:=HDRPTR;
    END;
  END; (* BLD_SEGTEL *)

```

```

*-----*
* COALESCE: CCALESCE THE SEG TABLE MAKING GOOD BRK. *
* ONLY LOSS OF EFFICIENCY IS WITH CROSS SEGMENT *
* FORWARD JUMPS. THESE MAY PRECLUDE THE COMBINING *
* OF A SEGMENT BECAUSE OF ADDED CODE FOR THE JUMP *
*-----*

```

```

PROCEDURE COALESCE (VAR SBR:TBLPTR; LIMIT:INTEGER;
VAR GOOD SEGMENT:BOOLEAN; VAR SBRINVNEST:INTEGER);
VAR CURSEG:TBLPTR;

```

```

*-----*
* SERSUM: SUMS ALL SBRS ON A SBRLIST. SETS INCLUDED *
* TRUE. ADDS SBRCONST IF SBRBREAK IS ENCOUNTERED. *
*-----*

```

```

PROCEDURE SERSUM (VAR SBRLST:TBLPTR; VAR SUMSBR:INTEGER);
VAR SERL,SER:TBLPTR;

```

```

BEGIN
SERL:=SBRLST;
WHILE SBRL<>NIL DO
BEGIN
SBR:=SBR1@.SER;
CASE SBR@.TAG OF
TABLE:
BEGIN
IF (SER@.COALESCED) THEN
IF NOT (SER@.INCLUDED) THEN
BEGIN
SBR@.INCLUDED:=TRUE;
SUMSBR:=SUMSBR+SBR@.LENGTH
-SBRCONTCONST;
IF SBR@.SBRLIST<>NIL THEN
SERSUM (SBR@.SERLIST,SUMSBR);
END;

```

```

END;
SBRBREAK:
SUMSER:=SUMSBR+SBRCONST;

```

```

END;
SBRL:=SBR1@.NEXT_SBR;

```

```

END; (* SERSUM *)
*-----*

```

```

*-----*
* SBRSUMLINK: ADDS ALL SBRS BELOW AN INVOKE NODE. *
* IT DOES NOT DO THE WHOLE SBRLIST. THAT IS SUMSBR *
*-----*

```

```

PROCEDURE SBRSUMLINK (VAR SBRPTR:TBLPTR; VAR SUMT:INTEGER);
VAR PTR:TBLPTR;

```

```

BEGIN
PTR:=SBRPTR@.SER;
CASE PTR@.TAG OF
TABLE:
IF (PTR@.COALESCED) AND (PTR@.TABLELIST=NIL) THEN
IF (NOT (PTR@.INCLUDED)) THEN
BEGIN
PTR@.INCLUDED:=TRUE;
SBRSUM (PTR@.SERLIST,SUMT);
SUMT:=SUMT+PTR@.LENGTH-SBRCONTCONST;
END

```

```

ELSE
IF (PTR@.COALESCED) AND (NOT (PTR@.INCLUDED)) THEN
SUMT:=SUMT+PTR@.LENGTH-SBRCONTCONST;

```

```

SBRBREAK:
SUMT:=SUMT;

```

```

END; (* SBRSUMLINK *)
*-----*

```

```

*-----*
* CHK SEGSIZE: VERIFIES THAT THE SEGMENT OF NEST 0 *
* TOGETHER WITH ITS REQUIRED SBR INVOKES IS WITHIN *
* MEMORY CONSTRAINTS. IF IT DOES NOT FIT THEN A *
* SBRBREAK IS INSERTED. THIS ROUTINE ASSUMES THAT *
* THE SEGMENT LENGTH IS OK. SET LENGTH ENSURES THIS *
* IF A SBR IS NOT COALESCED THIS ROUTINE MUTUALLY *
* RECURSES WITH COALESCE. DURING ITS CHECK, IT *
* WILL INCLUDE AND RESET INCLUDES. *
*-----*
PROCEDURE CHK_SEGSIZE(VAR CURSEG:TBLPTR; LIMIT:INTEGER);
VAR SEG,SBR,SBFI:TBLPTR;
    CHKLENGTH,SUMSBR:INTEGER;

```

```

*-----*
* INSERT_SBRBRK: INSERTS THE SBRBREAK NODE *
*-----*

```

```

PROCEDURE INSERT_SBRBRK(SERNODE:TBLPTR);
VAR INSERT,CUR,FWD:TBLPTR;
BEGIN
    CUR:=SERNODE;
    IF CUR@.SBR@.TAG <> SBRBREAK THEN
        BEGIN
            FWD:=CUR@.SBR;
            NEW(INSERT,SBRBREAK);
            INSERT@.TAG:=SBRBREAK;
            INSERT@.SBRZ:=CUR@.SBR;
            CUR@.SER:=INSERT;
        END;
    END;
(* INSERT_SBRBRK *)

```

```

*-----*
BEGIN
    SEG:=CURSEG;
    WHILE SEG<>NIL DO
        BEGIN
            CHKLENGTH:=SEG@.LENGTH;
            IF SEG@.TABLELIST=NIL THEN
                CHKLENGTH:=CHKLENGTH-SBRCONTCNST;
            IF SEG@.SERLIST<>NIL THEN
                BEGIN
                    SBRL:=SEG@.SBRLIST;
                    WHILE SBRL<>NIL DO
                        BEGIN
                            SER:=SBRL@.SBR;
                            CASE SBR@.TAG OF
                                TABLE:
                                    BEGIN
                                        IF (SBR@.COALESCED=FALSE) THEN
                                            COALESCE(SBR,LIMIT,
                                                GOOD_SEGMENT,SBRINVNEST);
                                        IF SER@.TABLELIST<>NIL THEN
                                            BEGIN
                                                INSERT_SBRBRK(SBRL);
                                                END;
                                                SUMSBR:=0;
                                                SBRSUMLINK(SBRL,SUMSBR);
                                                CHKLENGTH:=CHKLENGTH+SUMSBR;
                                                IF (CHKLENGTH>LIMIT) THEN
                                                    BEGIN
                                                        INSERT_SBRBRK(SBRL);
                                                        CHKLENGTH:=CHKLENGTH-SUMSER;
                                                        RESET_INCLUDED(SEG@.SERLIST);
                                                    END;
                                                END;
                                            END;
                                        SBRBRK:
                                            END;
                                        SERL:=SBRL@.NEXT_SBR;
                                    END;
                                END;
                            END;
                        END;
                    END;
                END;
            END;
        END;
    END;

```

```

        IF SEG@.SERLIST<>NIL THEN
            RESET INCLUDED(SEG@.SBRLIST);
            DIAGS NEST1SERBRK(TEMPFILE,SEG,GOOD_SEGMENT);
            SEG:=SEG1.TABLELIST;
        END;
    END;                                     (* CHK_SEGSIZE *)
(*-----*)

(*-----*)
(* CCMBINE: TAKES THE CHECKED SEGTABLE AND COMBINES *)
(* IT INTO A MAXIMIZED COMBINATION OF SEGMENTS AND *)
(* SBR. BASICALLY, IT MERGES THE ADJACENT SEGMENTS *)
(* IF THEY CAN BE MERGED. *)
(*-----*)
PROCEDURE COMBINE(VAR CURSEG:TBLPTR; LIMIT:INTEGER);
    VAR SUMSBRF,SUMSBRC,SUMFWD,SUMCUR,SUMTOT:INTEGER;
        CUR,FWD:TBLPTR;
        DELSUMSBRZ,DELSUMFJMP:INTEGER;

    (*-----*)
    (* MERGES: MERGES ADJACENT SEGMENTS TO INCLUDE THEIR *)
    (* RESPECTIVE FJUMPLISTS AND SBRLISTS. IT THEN *)
    (* USES CLEAN TO REMOVE ANY DUPLICATE JUMPS/SERS *)
    (* USES CLEAN'S DELETE FACILITY TO READJUST LENGTH *)
    (* THE LENGTH FOR THE SBRBREAK CODE IS NOT *)
    (* INCLUDED IN SEGMENT LENGTH, ROOM IS LEFT. *)
    (*-----*)
    PROCEDURE MERGES(VAR CUR,FWD:TBLPTR);
        VAR SBRTAIL,JMPTAIL:TBLPTR; DELETE:INTEGER;
        BEGIN
            IF FWD@.F_JUMPLIST<>NIL THEN
                IF CUR@.F_JUMPLIST<>NIL THEN
                    BEGIN
                        JMPTAIL:=CUR@.F_JUMPLIST;
                        WHILE JMPTAIL@.NEXT_FJUMP<>NIL DO
                            JMPTAIL:=JMPTAIL@.NEXT_FJUMP;
                        JMPTAIL@.NEXT_FJMP:=FWD@.F_JUMPLIST;
                    END
                ELSE
                    CUR@.F_JUMPLIST:=FWD@.F_JUMPLIST;
                IF FWD@.SBRLIST<>NIL THEN
                    IF CUR@.SERLIST<>NIL THEN
                        BEGIN
                            SBRTAIL:=CUR@.SERLIST;
                            WHILE SBRTAIL@.NEXT_SBR<>NIL DO
                                SBRTAIL:=SBRTAIL@.NEXT_SBR;
                            SBRTAIL@.NEXT_SBR:=FWD@.SERLIST;
                        END
                    ELSE
                        CUR@.SERLIST:=FWD@.SBRLIST;
                    CUR@.STOP_ADDR:=FWD@.STOP_ADDR;
                    CUR@.TABLELIST:=FWD@.TABLELIST;
                    CUR@.NEST:=0;
                    DELETE:=0;
                    CLEAN(CUR,DELETE);
                    CUR@.LENGTH:=CUR@.LENGTH+FWD@.LENGTH-((DELETE+1)*
                                                                FJUMPCONST);
                    DISPOSE(FWD,DELETE);
                END;
            END;
        END;
    (*-----*)

```

```

{ *-----*
* MOD SUMTOTFJMP: SIMULATES THE COMBINE OF THE FJUMP*
* THIS IS A PREDICTION IF ADJACENT SEGMENTS WERE *
* MERGED. *
*-----*
PROCEDURE MOD_SUMTOTFJMP (CUR, FWD: TBLPTR;
                          VAR DELSUMFJMP: INTEGER);
VAR P, T: TBLPTR; FCOUNT: INTEGER;
BEGIN
  FCOUNT:=0;
  P:=CUR@.F JUMPLIST;
  WHILE P<>NIL DO
    BEGIN
      IF P@.JUMP_ADDRTO<=FWD@.STOP_ADDR THEN
        FCOUNT:=FCOUNT+1;
        T:=FWD@.F JUMPLIST;
        WHILE T<>NIL DO
          BEGIN
            IF P@.JUMP_ADDRTO = T@.JUMP_ADDRTO THEN
              FCOUNT:=FCOUNT+1;
              T:=T@.NEXT_FJUMP;
            END;
            P:=P@.NEXT_FJUMP;
          END;
        DELSUMFJMP:=(FCOUNT+1)*FJUMPCONST;
      END;
    END;
  (*-----*
  (* MOD_SUMTOTFJMP *)
  (*-----*)

```

```

{ *-----*
* MOD SUMTOTSBR: SIMULATES THE CHANGE TO TOTAL LEN *
* BECAUSE OF THE MERGING OF ADJACENT SEGMENTS. *
*-----*
PROCEDURE MOD_SUMTOTSBR (CUR, FWD: TBLPTR;
                          VAR DELSUMSBRZ: INTEGER);
VAR P, T, PP, TT: TBLPTR; SCOUNT: INTEGER;
BEGIN
  SCOUNT:=0;
  P:=CUR@.SBRLIST;
  WHILE P<>NIL DO
    BEGIN
      T:=FWD@.SBRLIST;
      WHILE T<>NIL DO
        BEGIN
          PP:=P@.SBR;
          IF PP@.TAG = SBRBREAK THEN
            BEGIN
              TT:=T@.SER;
              IF TT@.TAG = SBRBREAK THEN
                IF PP@.SBRZ = TT@.SBRZ THEN
                  SCOUNT:=SCOUNT+1;
                END;
              T:=T@.NEXT_SBR;
            END;
          P:=P@.NEXT_SBR;
        END;
      DELSUMSBRZ:=SCOUNT*SBRCONST;
    END;
  (*-----*
  (* MOD_SUMTOTSBR *)
  (*-----*)
  BEGIN
    CUR:=CURSEG;
    DIAGS NEST 1 LENGTHCHK (TEMPFILE, CUR, GOOD_SEGMENT);
    SUMTOT:=0;
    SUMSBRZ:=0;
    IF CUR@.SBRLIST<>NIL THEN
      SERSUM (CUR@.SBRLIST, SUMSBRZ);
    SUMCUR:=CUR@.LENGTH+SUMSBRZ;
    WHILE (CUR@.TABLELIST<>NIL) DO

```

```

BEGIN
  FWD:=CUR@.TABLELIST;
  SUMSBRF:=0;
  IF FWD@.SERLIST<>NIL THEN
    SBRSUM(FWD@.SBRLIST,SUMSBRF);
    SUMFWD:=FWD@.LENGTH+SUMSBRF;
    SUMTOT:=SUMCUR+SUMFWD;
    MOD-SUMTOTFJMP(CUR,FWD,DELSUMFJMP);
    MOD-SUMTCTISBR(CUR,FWD,DELSUMSBRZ);
    SUMTCT:=SUMTOT-DELSUMFJMP-DELSUMSBRZ;
    IF SUMTOT<=LIMIT THEN
      BEGIN
        MERGES(CUR,FWD);
        CUR@.LENGTH:=CUR@.LENGTH-DELSUMSBRZ;
        SUMCUR:=SUMTOT;
      END
    ELSE
      BEGIN
        RESET INCLUDED(FWD@.SBRLIST);
        RESET INCLUDED(CUR@.SBRLIST);
        CUR:=FWD;
        DIAGS_NEST1LENGTHCHK(TEMPFILE,CUR,GOOD_SEGMENT);
        SUMCUR:=CUR@.LENGTH;
        SUMSERC:=0;
        SBRSUM(CUR@.SBRLIST,SUMSERC);
        SUMCUR:=SUMCUR+SUMSERC;
      END;
    SUMTOT:=0;
  END;
  IF CUR@.SBRLIST<>NIL THEN
    RESET INCLUDED(CUR@.SBRLIST); (*ALL INCLUDES RST*)
    CURSEG@.COALESCED:=TRUE;
  END;
  (*-----(* COMBINE *)-----*)
  BEGIN
    CURSEG:=SBR;
    SBRINVNEST:=SBRINVNEST+1;
    DIAGS_NEST6SBRINVCHK(TEMPFILE,CURSEG,GOOD_SEGMENT,
      SBRINVNEST);
    CHK_SEGSIZE(CURSEG,LIMIT);
    CCMEINE(CURSEG,LIMIT);
    SBRINVNEST:=SBRINVNEST-1;
  END;
  (*-----(* COALESCE *)-----*)

```

```

*=====  

* INSTRUCTIONS: PRINTS OUT THE SEGMENTED CODE TOGETHER *  

* WITH OTHER INFORMATION TO USE THE SEGMENTED CODE. *  

* DCES THIS BY FIRST CHECKING THE SEGMENT TABLE AND *  

* ASSIGNING A MEMORY MODULE NUMBER TO SPECIFIC LOCA- *  

* TICNS IN THE TABLE WHERE SBR BREAKS OCCUR. AN *  

* IMPLIED ASSIGNMENT IS MADE TO THE FIRST SBR AS A *  

* START POINT. OTHERS ARE INCLUDED IF THERE IS A BRK *  

* LEADING TO IT. ONCE MODULES ARE ASSIGNED THEN CODE *  

* WITH PROMPTS ADDED ARE COPIED FROM THE ORIGINALS. *  

* THESE COPIES ARE THEN PRINTED OUT. *  

*=====  

PROCEDURE INSTRUCTIONS (VAR OUTFILE, MESSAGEFILE,  

TEMPFILE:TEXT; BUILT_CODE:CODEPTR;  

SEGTL:TBLPTR; PART_NUM:INTEGER;  

PARTITION:REAL; GOOD_SEGMENT:BOOLEAN);  

VAR HEAD_MODULE:TBLPTR;  

-----  

* BLD_MODULENODES: BUILDS THE MODULES BASED ON THE *  

* BREAKS ENCOUNTERED IN THE SEGTL *  

-----  

PROCEDURE BLD_MODULENODES (SEGTL:TBLPTR;  

VAR HEAD_MODULE:TBLPTR);  

VAR TAIL_MODULE, SEG:TBLPTR;  

MEMCOUNT:INTEGER;  

-----  

* INSERT_MODULENODES: INSERTS A MODULENODE INTO *  

* THE MODULELIST. IT MUST FIRST CHECK TO SEE *  

* THAT IT IS NOT ALREADY ACCOUNTED FOR AS THERE *  

* MAY BE MULTIPLE INVOKES OF THE SAME BREAK. *  

-----  

PROCEDURE INSERT_MODULENODES (SEGTL:TBLPTR;  

VAR HEAD_MODULE, TAIL_MODULE:TBLPTR;  

VAR MEMCOUNT:INTEGER);  

VAR INSERT:TBLPTR;  

-----  

* NOT_IN_MODULELIST: CHECKS TO SEE IF IN LIST *  

-----  

FUNCTION NOT_IN_MODULELIST (SEGTL, HEAD_MODULE  

:TBLPTR):  

BOOLEAN;  

VAR S:TBLPTR;  

BEGIN  

S:=HEAD_MODULE;  

NOT_IN_MODULELIST:=TRUE;  

IF S<>NIL THEN  

BEGIN  

WHILE S<>NIL DO  

BEGIN  

IF S@.SEGTL = SEGTL THEN  

NOT_IN_MODULELIST:=FALSE;  

S:=S@.NEXT;  

END;  

END;  

END;  

END; (* NOT_IN_MODULELIST *)  

-----  

BEGIN  

IF NOT_IN_MODULELIST (SEGTL, HEAD_MODULE) THEN  

BEGIN  

NEW (INSERT, MODULE);  

INSERT@.TAG:=MODULE;  

INSERT@.MEMNUM:=MEMCOUNT;  

MEMCOUNT:=MEMCOUNT+1;  

INSERT@.OFFSET:=-SEGTL@.START_ADDR;  

INSERT@.HIGH_OFFSET:=- (SEGTL@.START_ADDR DIV  

100);

```



```

*-----*
* BLD MEMODULECODE: THIS ROUTINE BUILDS THE CODE FOR *
* THE MEMORY MODULE CODELISTS. IT WILL JUSTIFY ALL *
* THE ADDRESSES AND ADD BREAK CODE. *
*-----*
PROCEDURE BLD_MEMODULECODE (BUILT_CODE:CODEPTR;
                           VAR HEAD_MEMODULE:TBLPTR);
  VAR CURMEM:TBLPTR;

  *-----*
  * BLD A MEMORY: THIS ROUTINE INITIALIZES THE *
  * RECURSIVE PROCESS THAT WILL BE DONE IN THE *
  * FORM_MEMORY ROUTINE. *
  *-----*
  PROCEDURE BLD_A_MEMORY (BUILT_CODE:CODEPTR;
                        VAR HEAD_MEMODULE:TBLPTR;
                        CURMEM:TBLPTR);

    VAR ADDRESS:INTEGER;
        CODE_H, CODE_T:TBLPTR;
        SEG:TBLPTR;

    *-----*
    * FORM MEMORY: BUILDS A COMPLETE MEMORY MODULE *
    * CODE COMPLETE WITH BREAK CODE AND JUSTIFY. *
    * ROUTINE RECURSES ON EACH SBR ON THE SBRLISTS *
    *-----*
    PROCEDURE FORM_MEMORY (BUILT_CODE:CODEPTR;
                        VAR HEAD_MEMODULE, CURMEM, SEG, CODE_H,
                        CODE_T:TBLPTR;
                        VAR ADDRESS:INTEGER);

      VAR CODE_HH, CODE_TT, SER, SBRL:TBLPTR;

      *-----*
      * PROCESS SEG: TAKES CARE OF ONE SEGMENT IN THE *
      * SEGMENT TABLE'S WORTH OF CODE. *
      *-----*
      PROCEDURE PROCESS_SEG (BUILT_CODE:CODEPTR;
                          VAR HEAD_MEMODULE,
                          CURMEM:TBLPTR; SEG:TBLPTR;
                          VAR CODE_HH, CODE_TT:TBLPTR;
                          VAR ADDRESS:INTEGER);

        VAR START, STOP:INTEGER;

        *-----*
        * COPY CODE: COPIES CODE FROM A START TO A *
        * STOP POINT OF THE BUILT_CODE *
        *-----*
        PROCEDURE COPYCODE (BUILT_CODE:CODEPTR;
                          VAR CODE_HH, CODE_TT:TBLPTR;
                          VAR ADDRESS:INTEGER;
                          START, STOP:INTEGER);

          VAR INSERT, CURTP:TBLPTR;
              CURCP:CODEPTR;
          BEGIN
            CURCP:=BUILT_CODE;
            WHILE CURCP@.AADDR<>START DO
              CURCP:=CURCP@.SEQ;
              NEW (INSERT, CODE);
              INSERT@.TAG:=CODE;
              INSERT@.ABS_ADDR:=CURCP@.AADDR;
              INSERT@.ADDRESS:=ADDRESS;
              ADDRESS:=ADDRESS+1;
              INSERT@.KEYCODE:=CURCP@.KEY;
              INSERT@.SEQUENTIAL:=NIL;
              CODE_HH:=INSERT;
              CODE_TT:=INSERT;
            REPEAT
              NEW (INSERT, CODE);
              CURCP:=CURCP@.SEQ;
              INSERT@.TAG:=CODE;

```

```

INSERT@.ABS ADDR:=CURCP@.AADDR;
INSERT@.ADDRESS:=ADDRESSSS;
ADDRESSSS:=ADDRESSSS+1;
INSERT@.KEYCODE:=CURCP@.KEY;
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;
UNTIL TCURCP@.AADDR = STOP);
CODE_TT@.SEQUENTIAL:=NIL;
END: (* COPYCODE *)
(*-----*)

```

```

(*-----*)
(* ADD RETURN CODE: ADDS SBR RETURN CODE TO THE *)
(* TAIL SEGMENT OF THE INVOKED SBR *)
(*-----*)

```

```

PROCEDURE ADD_RETURN CODE (VAR CODE TT: TELPTR;
VAR ADDRESSSS: INTEGER);

```

```

VAR INSERT: TBLPTR;
BEGIN

```

```

CODE_TT@.KEYCODE:=STO; (*INVSBR CHG 2 STC*)
NEW (INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESSSS;
ADDRESSSS:=ADDRESSSS+1;
INSERT@.KEYCODE:=DISPLAYREGSTORE; (*DISPLAY*)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

```

```

NEW (INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESSSS;
ADDRESSSS:=ADDRESSSS+1;
INSERT@.KEYCODE:=RCLIND; (* RCL IND *)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

```

```

NEW (INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESSSS;
ADDRESSSS:=ADDRESSSS+1;
INSERT@.KEYCODE:=MANRTNREG; (* MAN RTN REG *)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

```

```

NEW (INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESSSS;
ADDRESSSS:=ADDRESSSS+1;
INSERT@.KEYCODE:=OP; (* OP *)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

```

```

NEW (INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESSSS;
ADDRESSSS:=ADDRESSSS+1;
INSERT@.KEYCODE:=30+MANRTNREG; (*+MANRTNREG*)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

```

```

NEW (INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESSSS;
ADDRESSSS:=ADDRESSSS+1;
INSERT@.KEYCODE:=RS; (* RUN/STCP *)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

```

```

CODE_TT@.SEQUENTIAL:=NIL;
END; (* ADD_RETURNCODE *)
(*-----*)

```

```

(*-----*)
(* ADDCODE SBRBRK MARKINVOKED: ADDS CODE FOR A *
* SBRERK AND WILL MARK THE INVOKED RTN PCR *
* MANUAL RETURN CODE *)
(*-----*)
PROCEDURE ADDCODE SBRBRK MARKINVOKED
(VAR HEAD_MEMODULE, CURMEM: TBLPTR;
SEG: TBLPTR; VAR CODE TT: TBLPTR;
VAR ADDRESS: INTEGER);
VAR INSERT, SBR, SBRL: TBLPTR;
LBIADDR: INTEGER;

```

```

(*-----*)
(* FIND LBL: FINDS THE LABEL FROM A GIVEN *
* ADDRESS USING THE BUILT_CODE LIST *)
(*-----*)
FUNCTION FIND_LBL (BUILT_CODE: CODEPTR;
ADDRESS: INTEGER): INTEGER;
VAR C: CODEPTR;
BEGIN
C:=BUILT_CODE;
WHILE C@.AADDR<>ADDRESS DO
C:=C@.SEQ;
FIND_LBL:=C@.SEQ@.KEY;
END; (* FIND_LBL *)
(*-----*)

```

```

(*-----*)
(* GENCCDESBR: ADIS CODE FOR A ERK SBR INVOK*
* *)
(*-----*)
PROCEDURE GENCODESBR (VAR CODE TT: TBLPTR;
HEAD_MEMODULE, CURMEM, SER: TBLPTR;
VAR ADDRESS: INTEGER);
VAR RELADDR, HUNDREDS, TENS, UNITS: INTEGER;
MEMPTR: TBLPTR;
BEGIN
MEMPTR:=HEAD_MEMODULE;
WHILE (MEMPTR@.SEGTBLS<>SBR@.SBRZ) DO
MEMPTR:=MEMPTR@.NEXT;
RELADDR:=SBR@.SBRZ@.START_ADDR+MEMPTR@.
OFFSET;
HUNDREDS:=RELADDR DIV 100;
TENS:=(RELADDR-(HUNDREDS*100)) DIV 10;
UNITS:=RELADDR-(HUNDREDS*100+TENS*10);
NEW (INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESS;
ADDRESS:=ADDRESS+1;
INSERT@.KEYCODE:=LBL; (* LBL *)
IF CODE TT <> NIL THEN
CODE TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;
NEW (INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESS;
ADDRESS:=ADDRESS+1;
INSERT@.KEYCODE:=FIND_LBL (* KEY_LBL *)
(BUILT_CODE, SBR@.SBRZ@.START_ADDR);
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

```

```

NEW (INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESS;
ADDRESS:=ADDRESS+1;
INSERT@.KEYCODE:=STO; (* STORE *)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

```

```

NEW (INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESS;
ADDRESS:=ADDRESS+1;
INSERT@.KEYCODE:=DISPLAY REGSTORE; (* DISP *)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

```

```

NEW (INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESS;
ADDRESS:=ADDRESS+1;
INSERT@.KEYCODE:=OP; (* OP *)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

```

```

NEW (INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESS;
ADDRESS:=ADDRESS+1;
INSERT@.KEYCODE:=20+MANRTNREG; (* INCR *)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

```

```

NEW (INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESS;
ADDRESS:=ADDRESS+1;
INSERT@.KEYCODE:=CURMEM@.MEMNUM; (* MEM# *)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

```

```

NEW (INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESS;
ADDRESS:=ADDRESS+1;
INSERT@.KEYCODE:=STOIND; (* IND STO *)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

```

```

NEW (INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESS;
ADDRESS:=ADDRESS+1;
INSERT@.KEYCODE:=MANRTNREG; (* MAN RTN *)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

```

```

NEW (INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESS;
ADDRESS:=ADDRESS+1;
INSERT@.KEYCODE:=MEMPTR@.MEMNUM; (* MEM **)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

```

```

NEW (INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESS;
ADDRESS:=ADDRESS+1;

```

```
INSERT@.KEYCODE:=DECIMAL; (* . *)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;
```

```
NEW(INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESSSS;
ADDRESSSS:=ADDRESSSS+1;
INSERT@.KEYCODE:=HUNDREDS; (* HUNDREDS *)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;
```

```
NEW(INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESSSS;
ADDRESSSS:=ADDRESSSS+1;
INSERT@.KEYCODE:=TENS; (* TENS *)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;
```

```
NEW(INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESSSS;
ADDRESSSS:=ADDRESSSS+1;
INSERT@.KEYCODE:=UNITS; (* UNITS *)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;
```

```
NEW(INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESSSS;
ADDRESSSS:=ADDRESSSS+1;
INSERT@.KEYCODE:=RS; (* R/S *)
INSERT@.SEQUENTIAL:=NIL;
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;
```

```
END; (* GENCODE_SER *)
(*-----*)
```

```
BEGIN
IF SEG@.SBRLIST<>NIL THEN
BEGIN
SERL:=SEG@.SBRLIST;
WHILE SBRL<>NIL DO
BEGIN
SBR:=SBRL@.SBR;
IF SBR@.TAG=SBRBREAK THEN
GENCODESBR(CODE_TT, HEAD_MEMODULE,
CURMEM, SBR, ADDRESSSS);
SBRL:=SBRL@.NEXT_SER;
END;
END;
END;
```

```
END; (* ADDCODE_SBRBRK_MARKINVOKED *)
(*-----*)
```

```

(*-----*)
(* JUSTIFY_CODE: SETS ALL THE JUMPS AND ADDRS *)
(*-----*)
PROCEDURE JUSTIFY_CODE(SEG,CURMEM:TBLPTR;
                      VAR CODE_HH:TBLPTR);
  VAR T,F,JMPTR:TBLPTR;
      DELADDR,RELADDR,ABSADDR:INTEGER;

  (*-----*)
  (* ADVANCE_CODE_TBL: ADVANCES THE CODEPTRS *)
  (* OF THE 1, 2, AND 3-STEP INSTRUCTIONS. *)
  (*-----*)
  PROCEDURE ADVANCE_CODE_TBL(VAR F:TBLPTR);
    BEGIN
      IF F@.KEYCODE IN STEP 3 THEN
        F:=F@.SEQUENTIAL@.SEQUENTIAL@.SEQUENTIAL
          @.SEQUENTIAL
      ELSE
        IF F@.KEYCODE IN STEP 2 THEN
          F:=F@.SEQUENTIAL@.SEQUENTIAL@.
            SEQUENTIAL
        ELSE
          IF F@.KEYCODE IN (STEP 1+(.71,76.))
            THEN F:=F@.SEQUENTIAL@.SEQUENTIAL
          ELSE
            F:=F@.SEQUENTIAL;
        (* ADVANCE_CODE_TBL *)
      END;
    (*-----*)
  BEGIN
    F:=CODE_HH;
    WHILE F<>NIL DO
      BEGIN
        IF F@.KEYCODE IN (STEP_3+STEP_2) THEN
          BEGIN
            IF F@.KEYCODE IN STEP 3 THEN
              T:=F@.SEQUENTIAL@.SEQUENTIAL
            ELSE
              T:=F@.SEQUENTIAL;
            ABSADDR:=T@.KEYCODE*100;
            ABSADDR:=ABSADDR
              +T@.SEQUENTIAL@.KEYCODE;
            IF SEG@.STOP_ADDR >= ABSADDR THEN
              BEGIN
                DELADDR:=ABSADDR-T@.ABS_ADDR;
                RELADDR:=T@.ADDRESS+DELADDR;
                T@.KEYCODE:=RELADDR DIV 100;
                T@.SEQUENTIAL@.KEYCODE:=
                  RELADDR-(100*T@.KEYCODE);
              END
            ELSE
              BEGIN
                JMPTR:=SEG@.F_JUMPLIST;
                WHILE JMPTR@.JUMP_ADDRTO<>
                  ABSADDR DO
                  JMPTR:=JMPTR@.NEXT_FJUMP;
                  T@.KEYCODE:=JMPTR@.
                    JUMP_INTADDRTC1;
                  T@.SEQUENTIAL@.KEYCODE:=JMPTR@.
                    JUMP_INTADDRTC2;
                END;
              END;
            ADVANCE_CODE_TBL(F);
          END;
        END;
      END;
    (* JUSTIFY *)
  (*-----*)

```

```

-----*
* AEDCODE_FJMP: ADDS CODE FOR JUMP BREAK *
-----*
PROCEDURE ADDCODE_FJMP(VAR HEAD MEMODULE,CURMEM,
                      SEG, CODE TT:TBLPTR;
                      VAR ADDRESS:INTEGER);
VAR CUR,INSERT,JUMPTR,MEMPTR:TBLPTR;
    ADDCRT01,ADDRTO2, MEM_ADDRTO:INTEGER;
    DELTA_ADDRESS:INTEGER;

-----*
* GEN JUMPCODE SETINTADDRS: GENERATES THE *
* JUMP CODE AND SETS THE INTADDR FIELDS *
* OF THE SEGMENT TABLE F JUMPLIST. *
* INTADDR ARE THE ADDRESSES LOCAL TO THAT *
* SPECIFIC PIECE OF CODE (THE PROMPT). *
-----*
PROCEDURE GEN JUMPCODE SETINTADDRS
(VAR CODE_TT:TBLPTR; VAR ADDRESS:INTEGER;
 VAR JUMPTR:TBLPTR);
VAR INSERT:TBLPTR;
    HUNDREDS,TENS,UNITS,RELADDRESS:INTEGER;
BEGIN
HUNDREDS:=JUMPTR@.JUMP_ADDRTO1;
TENS:=JUMPTR@.JUMP_ADDRTO2 DIV 10;
UNITS:=JUMPTR@.JUMP_ADDRTO2-(10*TENS);

NEW(INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESS;
RELADDRESS:=ADDRESS;
ADDRESS:=ADDRESS+1;
INSERT@.KEYCODE:=STO; (* STO *)
IF CODE_TT<>NIL THEN
    CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

NEW(INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESS;
ADDRESS:=ADDRESS+1;
INSERT@.KEYCODE:= DISPLAYREGSTORE; (* DISP*)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

NEW(INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESS;
ADDRESS:=ADDRESS+1;
INSERT@.KEYCODE:=CE; (* CE *)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

NEW(INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESS;
ADDRESS:=ADDRESS+1;
INSERT@.KEYCODE:= JUMPTR@.MEM_ADDR;
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

NEW(INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESS;
ADDRESS:=ADDRESS+1;
INSERT@.KEYCODE:= DECIMAL; (* .*)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

```

```

NEW (INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESSSS;
ADDRESSSS:=ADDRESSSS+1;
INSERT@.KEYCODE:=HUNDREDS; (* 100S *)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

NEW (INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESSSS;
ADDRESSSS:=ADDRESSSS+1;
INSERT@.KEYCODE:=TENS; (* 10S *)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

NEW (INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESSSS;
ADDRESSSS:=ADDRESSSS+1;
INSERT@.KEYCODE:=UNITS; (* 1S *)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;

NEW (INSERT, CODE);
INSERT@.TAG:=CODE;
INSERT@.ADDRESS:=ADDRESSSS;
ADDRESSSS:=ADDRESSSS+1;
INSERT@.KEYCODE:=RS; (* R/S *)
CODE_TT@.SEQUENTIAL:=INSERT;
CODE_TT:=INSERT;
JUMPTR@.JUMP_INTADDRTO1:=RELADDRESS
DIV 100;
JUMPTR@.JUMP_INTADDRTO2:=RELADDRESS-100*
JUMPTR@.JUMP_INTADDRTO1;
END; (* GEN_JUMPCODE_SETINTADDR *)
(*-----*)
BEGIN
IF SEG@.TABLELIST<>NIL THEN
BEGIN
MEMPTR:=HEAD MEMODULE;
WHILE NOT (SEG@.STOP_ADDR+1 = MEMPTR@.
SEGTBLS@.START_ADDR) DO
MEMPTR:=MEMPTR@.NEXT;
ADDRTO1:=MEMPTR@.SEGTBLS@.START_ADDR
DIV 100;
ADDRTO2:=MEMPTR@.SEGTBLS@.START_ADDR
- (100*ADDRTO1);
ADDRTO1:=ADDRTO1+MEMPTR@.HIGHOFFSET;
ADDRTO2:=ADDRTO2+MEMPTR@.LOWOFFSET;
MEM_ADDRTO:=MEMPTR@.MEMNUM;
NEW (JUMPTR, FWD_JUMP);
WITH JUMPTR DO
BEGIN
JUMP_ADDRTO1:=ADDRTO1;
JUMP_ADDRTO2:=ADDRTO2;
MEM_ADDR:=MEM_ADDRTO;
END;
GEN_JUMPCODE_SETINTADDR
(CODE_TT, ADDRESSSS, JUMPTR);
DISPOSE (JUMPTR, FWD_JUMP);
END;
IF SEG@.F_JUMPLIST<>NIL THEN
BEGIN
JUMPTR:=SEG@.F_JUMPLIST;
WHILE JUMPTR<>NIL DO
BEGIN
MEMPTR:=HEAD_MEMODULE;

```

```

WHILE NOT ((JUMPTR@.JUMP_ADDRTO >=
MEMPTR@.SEGTBLS@.START_ADDR) AND
(JUMPTR@.JUMP_ADDRTO <=
MEMPTR@.SEGTBLS@.STOP_ADDR)) DO
MEMPTR:=MEMPTR@.NEXT;

DELTA_ADDRESS:=JUMPTR@.JUMP_ADDRTO-
MEMPTR@.SEGTBLS@.START_ADDR;
ADDRTO1:=DELTA_ADDRESS DIV 100;
ADDRTO2:=DELTA_ADDRESS-100*
ADDRTO1;
MEM_ADDRTO:=MEMPTR@.MEMNUM;
JUMPTR@.JUMP_ADDRTO1:=ADDRTO1;
JUMPTR@.JUMP_ADDRTO2:=ADDRTO2;
JUMPTR@.MEM_ADDR:=MEM_ADDRTO;
GEN_JUMPCODE SETINTADDRS
(CODE TT,ADDRESS, JUMPTR);
JUMPTR:=JUMPTR@.NEXT_FJUMP;
END;
END;
(* ADDCODE_FJMP *)
(*-----*)
BEGIN
START:=SEG@.START_ADDR;
STOP:=SEG@.STOP_ADDR;
COPYCODE (BUILT_CODE, CODE_HH, CODE TT,
-ADDRESS, START, STOP);
IF (CURMEM@.RETURNCODE_NEEDED) AND
(SEG=CURMEM@.SEGTBLS) THEN
ADD_RETURNCODE (CODE TT, ADDRESS);
ADDCODE_FJMP (HEAD_MEMODULE, CURMEM, SEG, CODE TT,
ADDRESS);
ADDCODE_SBRBRK_MARKINVOKED (HEAD_MEMODULE,
CURMEM, SEG, CODE TT, ADDRESS);
JUSTIFY_CODE (SEG, CURMEM, CODE_HH);
END;
(* PROCESS_SEG *)
(*-----*)

```

```

(*-----*)
* MARKINVCKED: MARKS MEMODULE OF SBR WHICH WAS *
* INVOLVED IN A BREAK FOR A MANUAL RETURN. *
* ADD_RETRUNCODE USES THIS MARK TO ADD CCDE. *
(*-----*)
PROCEDURE MARKINVOKED(VAR HEAD_MEMODULE:TBLPTR;
                      SBR:TBLPTR);
VAR MEMPTR, SBR_BRKINCLUDE:TBLPTR;
BEGIN
  SBR_BRKINCLUDE:=SBR@.SBRZ;
  MEMPTR:=HEAD_MEMODULE;
  WHILE SER_BRKINCLUDE@.TABLELIST<>NIL DO
    SBR_BRKINCLUDE:=SBR_BRKINCLUDE@.TABLELIST;
  WHILE MEMPTR@.SEGTBLS<>SBR_BRKINCLUDE DO
    MEMPTR:=MEMPTR@.NEXT;
  MEMPTR@.RETURNCODE_NEEDED:=TRUE;
END; (* MARKINVOKED *)
(*-----*)
BEGIN
  IF (SEG@.INCLUDED = FALSE) THEN
    BEGIN
      PROCESS_SEG (BUILT_CODE, HEAD_MEMODULE, CURMEM,
                  SEG, CODE_HH, CODE_TT, ADDRESS$);
      IF CCDE H=NIL THEN
        CODE H:=CODE_HH;
      IF CCDE T<>NIL THEN
        CODE T@.SEQUENTIAL:=CODE_HH;
      SEG@.INCLUDED:=TRUE;
      CODE T:=CODE TT;
      IF (SEG@.SBRLIST<>NIL) THEN
        BEGIN
          SERL:=SEG@.SBRLIST;
          WHILE SBRL<>NIL DO
            BEGIN
              SBR:=SBRL@.SBR;
              CASE SER@.TAG OF
                SERBREAK:
                  MARKINVOKED (HEAD_MEMODULE, SER);
                TABLE:
                  FORM_MEMORY (BUILT_CODE,
                              HEAD_MEMODULE, CURMEM, SBR, CODE H,
                              CODE_T, ADDRESS$);
              END; (* CASE *)
              SBR:=SBRL@.NEXT_SBR;
            END;
          END;
        END;
      END;
    END; (* FORM_MEMORY *)
  (*-----*)
  BEGIN
    SEG:=CURMEM@.SEGTBLS;
    ADDRESS$:=0; CODE H:=NIL; CODE T:=NIL;
    FCRM_MEMORY (BUILT_CODE, HEAD_MEMODULE, CURMEM, SEG,
                 CODE_H, CODE_T, ADDRESS$);
    CURMEM@.CODELIST:=CODE_H;
  END; (* BLD_A_MEMORY *)
  (*-----*)
  BEGIN
    CURMEM:=HEAD_MEMODULE;
    WHILE CURMEM<>NIL DO
      BEGIN
        BLD_A_MEMORY (BUILT_CODE, HEAD_MEMODULE, CURMEM);
        CURMEM@.SEGTBLS@.INCLUDED:=FALSE;
        IF CURMEM@.SEGTBLS@.SBRLIST<>NIL THEN
          RESET_INCLUDED (CURMEM@.SEGTBLS@.SBRLIST);
        CURMEM:=CURMEM@.NEXT;
      END;
    END;
  END; (* BLD_MEMODULECODE *)
  (*-----*)

```

```
(*-----*)
PROCEDURE OUTPUT_INSTR(VAR OUTFILE,MESSAGEFILE,
                      TEMPFILE:TEXT; BUILT_CODE:CODEPTR;
                      HEAD_MEMODULE:TBLPTR; PART_NUM:INTEGER;
                      PARTITION:REAL;
                      GOOD_SEGMENT:BOOLEAN);
```

```
(*-----*)
{* CUTPUT_GOODSEGS: PRINTS OUT GOOD SEGMENT INSTRUC *}
{*-----*}
PRCCEDURE CUTPUT_GOODSEG(VAR OUTFILE,MESSAGEFILE:
                        TEXT; HEAD_MEMODULE:TBLPTR; PART_NUM:INTEGER;
                        PARTITION:REAL);
```

```
(*-----*)
{* CUTPUT MSGF1: PRINTS OUT GOOD SEGMENT GENERAL *}
{* INSTRUCTIONS AND THE PROGRAM LISTING BY MEM *}
{* MODULES. *}
{*-----*}
PROCEDURE OUTPUT_MSGF1(VAR OUTFILE,MESSAGEFILE:TEXT;
                      HEAD_MEMODULE:TBLPTR; PARTNUM:INTEGER;
                      PARTITION:REAL);
```

```
VAR P,T:TBLPTR;
    STEP_TYPE:INTEGER;
```

```
(*-----*)
{* SET_STEP: SETS THE KEY CODE COUNTER FOR THE *}
{* PRINTLINE PROCEDURE. USED FOR CORRECT OUT- *}
{* PUT OF THE TI-59 CODE (3,2,1,0 STEP INSTR) *}
{*-----*}
* 3 STEP      2 STEP      71676      1      0
* AAA          AAA          AAA          AAA      AAA
* X            XXX          AAA          XX      (.1.)
* XXX          XXX          (.5,4.)/(.3,2.)      .
* XXX          XXX          (.8,7,6.)              & (.0.)
* (.12,11,10,9.)
```

```
PROCEDURE SET_STEP(T:TBLPTR;
                  VAR STEP_TYPE:INTEGER);
BEGIN
  IF STEP_TYPE IN (.0,1,3,5,8.) THEN
    IF T@.KEYCODE IN STEP_3 THEN
      STEP_TYPE:=12
    ELSE
      IF T@.KEYCODE IN STEP_2 THEN
        STEP_TYPE:=8
      ELSE
        IF T@.KEYCODE IN STEP_1 THEN
          STEP_TYPE:=3
        ELSE
          IF T@.KEYCODE IN (.71,76.) THEN
            STEP_TYPE:=5
          ELSE
            STEP_TYPE:=1;
          END;
        END;
      END;
    END;
  END;
  (* SET_STEP *)
  (*-----*)
```

```

(*-----*)
{* PRINT_LINE: PRINTS OUT ONE LINE OF CODE AT A *}
{* TIME. *}
(*-----*)
PROCEDURE FRINT_LINE (VAR OUTFILE:TEXT; VAR T:
                    TBLPTR; VAR STEP_TYPE:INTEGER);

(*-----*)
{* WRITELELS: WRITES TBLPTR LBLS IN WHOLE KEY *}
{* CODE FORMAT IE. NNN NN AAA *}
(*-----*)
PROCEDURE WRITELELS (VAR OUTFILE:TEXT; T:TBLPTR);
BEGIN
  WRITE (OUTFILE, '
WRITE LEADZERO (OUTFILE, T@.ADDRESS, 3);
WRITE (OUTFILE, '
WRITE LEADZERO (OUTFILE, T@.KEYCODE, 2);
WRITE (OUTFILE, '
WRITELELS (OUTFILE, T@.KEYCODE);
END; (* WRITELELS *)
(*-----*)

(*-----*)
{* WRITENUMS: WRITES OUT A LINE OF DIGITS OF *}
{* TI-59 CODE (KEYCODES ARE DIGITS NOT LBLS) *}
(*-----*)
PROCEDURE WRITENUMS (VAR OUTFILE:TEXT; T:TBLPTR);
BEGIN
  WRITE (OUTFILE, '
WRITE LEADZERO (OUTFILE, T@.ADDRESS, 3);
WRITE (OUTFILE, '
WRITE LEADZERO (OUTFILE, T@.KEYCODE, 2);
WRITE (OUTFILE, '
WRITE LEADZERO (OUTFILE, T@.KEYCODE, 2);
WRITELN (OUTFILE);
END; (* WRITENUMS *)
(*-----*)
BEGIN
IF STEP TYPE IN (.1,3,4,5,8,12.) THEN
  WRITELELS (OUTFILE, T)
ELSE
  WRITENUMS (OUTFILE, T);
  T:=T@.SEQUENTIAL;
  STEP_TYPE:=STEP_TYPE-1;
END; (* PRINT_LINE *)
(*-----*)
BEGIN
PRINTLN MSG (OUTFILE, MESSAGEFILE, BAXINSTR);
PRINT MSGLINE1 (OUTFILE, MESSAGEFILE, RTNRGTOP);
WRITELN (OUTFILE, REGCOUNT:3);
PRINT MSGLINE1 (OUTFILE, MESSAGEFILE, STOINRG);
WRITELN (OUTFILE, MANRTNREG:3);
WRITELN (OUTFILE);
PRINT MSGLINE1 (OUTFILE, MESSAGEFILE, PGMPARTIS);
WRITELN (OUTFILE, PARTITION:4:2);
WRITELN (OUTFILE);
PRINT MSGLINE1 (OUTFILE, MESSAGEFILE, PARTNUMIS);
WRITELN (OUTFILE, PARTNUM:1);
WRITELN (OUTFILE); WRITELN (OUTFILE);
P:=HEAD MEMODULE;
WHILE P<>NIL DO
  BEGIN
    T:=P@.CODELIST;
    STEP_TYPE:=0;
    WRITELN (OUTFILE); WRITELN (OUTFILE);
    WRITELN (OUTFILE);
    PRINT_MSGLINE1 (OUTFILE, MESSAGEFILE, MODN);
  END;
END;

```

```

WRITEIN (OUTFILE, P@.MEMNUM: 1);
PRINTIN_MSG (OUTFILE, MESSAGEFILE, CARD1);
PRINTLN_MSG (OUTFILE, MESSAGEFILE, SIDE1);
WHILE T<>NIL DO
  BEGIN
    IF T@.ADDRESS=240 THEN
      BEGIN
        WRITELN (OUTFILE); WRITELN (OUTFILE);
        PRINTLN_MSG (OUTFILE, MESSAGEFILE,
                     SIDE2);
      END;
    IF T@.ADDRESS = 480 THEN
      BEGIN
        WRITELN (OUTFILE); WRITELN (OUTFILE);
        PRINTLN_MSG (OUTFILE, MESSAGEFILE,
                     CARD2);
        PRINTLN_MSG (OUTFILE, MESSAGEFILE,
                     SIDE1);
      END;
    SET_STEP (T, STEP_TYPE);
    PRINT_LINE (OUTFILE, T, STEP_TYPE);
  END;
  P:=P@.NEXT;
END;
END;
(* OUTPUT_MSGF1 *)
(*-----*)

(*-----*)
(* OUTPUT_MSGF2: OUTPUTS SPECIFIC PROMPTS AND *)
(* SPECIAL PROGRAM INSTRUCTIONS *)
(*-----*)
PROCEDURE OUTPUT_MSGF2 (VAR OUTFILE, MESSAGEFILE: TEXT;
                       HEAD_MODULE: TBLPTR);
  VAR SBRL, SEP, SM, P, F, SEG: TBLPTR;
      IS_SBRERK: BOOLEAN;
  BEGIN
    P:=HEAD_MODULE;
    PRINTLN_MSG (OUTFILE, MESSAGEFILE, SPECIFICS);
    WHILE P<>NIL DO
      BEGIN
        SEG:=P@.SEGIBLS;
        F:=SEG@.F_JUMPLIST;
        SBRL:=SEG@.SBRLIST;
        WRITEIN (OUTFILE);
        PRINT_MSGLINE1 (OUTFILE, MESSAGEFILE,
                       MODPROMPTS);
        WRITEIN (OUTFILE, P@.MEMNUM: 1);
        PRINTIN_MSG (OUTFILE, MESSAGEFILE, PFWDJ);
        IF F= NIL THEN
          PRINTLN_MSG (OUTFILE, MESSAGEFILE, NONE)
        ELSE
          BEGIN
            WHILE F<>NIL DO
              BEGIN
                PRINT_MSGLINE1 (OUTFILE, MESSAGEFILE,
                                ASTER);
                WRITE (OUTFILE, F@.MEM_ADDR: 1, '.');
                WRITE_LEADZERO (OUTFILE, (F@.
                                           JUMP_ADDRTO1*100
                                           F@.JUMP_ADDRTO2), 3);
                WRITELN (OUTFILE);
                F:=F@.NEXT_PJUMP;
              END;
            END;
          END;
        PRINTLN_MSG (OUTFILE, MESSAGEFILE, PSBRINV);
        IF SERI= NIL THEN
          PRINTLN_MSG (OUTFILE, MESSAGEFILE, NONE)
        END;
      END;
    END;
  END;

```

```

ELSE
  BEGIN
    IS SBRBRK:=FALSE;
    WHILE SBRL<>NIL DO
      BEGIN
        SBR:=SFRL@.SBR;
        IF SBR@.TAG = SBRBREAK THEN
          BEGIN
            SM:=HEAD MEMODULE;
            SBR:=SBR@.SBRZ;
            WHILE SM@.SEGTBLS<>SBR DO
              SM:=SM@.NEXT;
            IS SBRBRK:=TRUE;
            PRINT_MSGLINE1(OUTFILE,
                          MESSAGEFILE,ASTER);
            WRITELN(OUTFILE,SM@.MEMNUM:1,
                  '.000');
          END;
          SBRL:=SBRL@.NEXT_SBR;
        END;
        IF NOT IS SBRBRK THEN
          PRINTLN_MSG(OUTFILE,MESSAGEFILE,NCNE)
        END;
        PRINTLN_MSG(OUTFILE,MESSAGEFILE,PANRTN);
        IF P@.RETURNCODE NEEDED THEN
          PRINTLN_MSG(OUTFILE,MESSAGEFILE,YES)
        ELSE
          PRINTLN_MSG(OUTFILE,MESSAGEFILE,NONE);
        PRINTLN_MSG(OUTFILE,MESSAGEFILE,PSEQ);
        IF SEG@.TABLELIST <> NIL THEN
          BEGIN
            PRINT_MSGLINE1(OUTFILE,MESSAGEFILE,
                          ASTER);
            WRITELN(OUTFILE,P@.NEXT@.
                  MEMNUM:1, '.000')
          END
        ELSE
          PRINTLN_MSG(OUTFILE,MESSAGEFILE,NONE);
        P:=P@.NEXT;
      END;
      WRITELN(OUTFILE);WRITELN(OUTFILE);
      PRINTLN_MSG(OUTFILE,SCRATCH,DATAREAD);
      WRITELN(OUTFILE);WRITELN(OUTFILE);
      PRINTLN_MSG(OUTFILE,SCRATCH,REGMAP);
      WRITELN(OUTFILE);WRITELN(OUTFILE);
    END;
  END;
  (*-----(* OUTPUT_MSG2 *)
  BEGIN
    OUTPUT_MSGF1(OUTFILE,MESSAGEFILE,HEAD MEMODULE,
                PART_NUM,PARTITION);
    OUTPUT_MSGF2(OUTFILE,MESSAGEFILE,HEAD_MEMODULE);
    WRITELN(OUTFILE);
    PRINTLN_MSG(OUTFILE,MESSAGEFILE,ENDLBL);
  END;
  (*-----(* OUTPUT_GOODSEG *)

```

```

{ *-----* }
{ * OUTPUT_EADSEG: HANDLES BAD SEG INSTRUCTIONS * }
{ *-----* }
PROCEDURE OUTPUT_BADSEG (VAR OUTFILE, MESSAGEFILE,
                        TEMPFILE:TEXT; BUILT_CODE:CODEPTR);
BEGIN
  WRITELN(OUTFILE); WRITELN(OUTFILE);
  PRINTLN MSG(OUTFILE, MESSAGEFILE, FAILINSTR);
  RESET(TEMPFILE);
  PRINTLN MSG(OUTFILE, TEMPFILE, 9);
  WRITELN(OUTFILE); WRITELN(OUTFILE);
  PRINTLN MSG(OUTFILE, MESSAGEFILE, UNSEGCCODLBL);
  WRITELN(OUTFILE); WRITELN(OUTFILE);
  PRINT CODELIST(OUTFILE, BUILT_CODE);
  WRITELN(OUTFILE);
  PRINTLN MSG(OUTFILE, MESSAGEFILE, ENDLBL);
END; (* OUTPUT_BADSEG *)
{ *-----* }
BEGIN
  IF NOT GOOD_SEGMENT THEN
    OUTPUT_BADSEG(OUTFILE, MESSAGEFILE, TEMPFILE,
                 BUILT_CODE)
  ELSE
    OUTPUT_GOODSEG(OUTFILE, MESSAGEFILE, HEAD_MEMODULE,
                  PART_NUM, PARTITION);
END; (* OUTPUT_INSTR *)
{ *-----* }
BEGIN
  BLD_MEMODULENODES(SEGTBL, HEAD_MEMODULE);
  BLD_MEMODULECODE(BUILT_CODE, HEAD_MEMODULE);
  OUTPUT_INSTR(OUTFILE, MESSAGEFILE, TEMPFILE, BUILT_CODE,
              HEAD_MEMODULE, PART_NUM, PARTITION,
              GOOD_SEGMENT);
END; (* INSTRUCTIONS *)
{ *=====* }

```

```
(*****  
(* MAIN DRIVER *)  
*****)
```

```
BEGIN  
  INIT_SETS(TEMPFILE,STEP 0,STEP 1,STEP 2,STEP 3,  
            GOOD_SEGMENT,MESSAGEFILE,TILBL,SBRINVNEST);  
  DET_LIMIT(REGCOUNT,LIMIT,NUMBANKS,PART_NUM,PARTITION);  
  INPUT (SCRATCH,BUILT_CODE,BUILT_CODE_COUNT);  
  SETJMPS (BUILT_CODE);  
  BUILD_SEGTBL (BUILT_CODE,SEGTBL,LIMIT,BUILT_CODE_COUNT);  
  COALESCE (SEGTBL@.TABLELIST,LIMIT,GOOD_SEGMENT  
            SBRINVNEST);  
  WRITEIN (TEMPFILE,'$9'); (* CLOSES TEMPFILE DIAG FILE *)  
  INSTRUCTIONS (OUTFILE,MESSAGEFILE,TEMPFILE,  
                BUILT_CODE,SEGTBL@.TABLELIST,PART_NUM,PARTITION,  
                GOOD_SEGMENT);  
  REWRITE (TEMPFILE); (* ERASES TEMPFILE DIAG FILE *)  
END.
```

```
(*****  
(* END OF PROGRAM *)  
*****)
```

APPENDIX J

MESSAGEFILE FILE--LINKER MESSAGES

\$81

=====

SAX59 PROGRAM INSTRUCTIONS: VERSION 1.0

=====

* CONGRATULATIONS, YOU HAVE JUST COMPILED A BASIC PROGRAM INTO A TI-59 PROGRAM. IN SO DOING IT IS VERY POSSIBLE THAT YOUR PROGRAM IS LARGER THAN THE MEMORY OF THE CALCULATOR. IF THIS IS THE CASE THEN THE PROGRAM HAS BEEN SEGMENTED AND PROMPTING CODE INSERTED TO GUIDE YOUR CALCULATOR PROGRAM DURING ITS EXECUTION. THE REMAINDER OF THIS OUTPUT CONSISTS OF TI-59 CODE LISTINGS AND OTHER INFORMATION TO AID YOU IN YOUR PROGRAM EXECUTION.

* THE FOLLOWING DEFINITIONS ARE PROVIDED AS AN AID TO READING THE PROGRAM LISTING FILE.

* DEFINITIONS:

* MODULE: A MODULE IS DEFINED TO BE ALL THE MEMORY DEDICATED TO PROGRAM STEPS. THE SIZE IS VARIABLE AND IS DEPENDENT ON THE REGISTER REQUIREMENT. VALUES RANGE FROM 0 TO 239, 479 OR 719 DEPENDING ON THE AMOUNT OF REGISTERS USED BY THE PROGRAM.

* CARD: A CARD IS DEFINED TO BE ONE MAGNETIC CARD. A CARD HOLDS 480 PROGRAM STEPS. THESE STEPS ARE NOT CONTIGUOUS BUT ARE ARRANGED ON THE TWO SIDES OF THE CARD.

* SIDE: A SIDE IS ONE HALF OF A CARD. IT CONTAINS UP TO 240 STEPS. WHEN ONE SIDE OF A CARD IS READ BY THE CALCULATOR 240 PROGRAM STEPS ARE FILLED IN MEMORY. THESE BLOCKS OF 240 STEPS ARE REFERED TO AS "BANKS" IN THE MANUFACTURER LITERATURE. WHEN LOADING A CARD YOU WILL LOAD ONLY BANK NUMBER 1 AND/OR 2 FOR PROGRAM STEPS.

* PARTITION: THIS IS DEFINED TO BE THE CURRENT SETTING OF CALCULATOR MEMORY AS APPLIED TO THE AMOUNT OF MEMORY DEDICATED TO STORAGE REGISTERS AND THE AMOUNT DEDICATED TO PROGRAM STEPS. WE WILL BE DEALING WITH 3 PARTITIONS. THESE ARE:

3	719.29
4	479.59
5	239.69

FORMAT: X YYY.ZZ

WHERE X STANDS FOR PARTITION NUMBER
YYY STANDS FOR PROGRAM STEPS (0-YYY)
ZZ STANDS FOR REGISTERS (0-ZZ).

=====

TI-59 PROGRAM LISTING BY MODULE/CARD/SIDE:

=====

- * THE FOLLOWING IS YOUR PROGRAM LISTING. THE PROGRAM IS LISTED ACCORDING TO MODULE NUMBER AND ITS ASSOCIATED CARDS AND CARD SIDES.
- * REFER TO THE TI-59 PROGRAMMER'S GUIDE ON HOW TO INPUT A PROGRAM AND WRITE IT TO MAGNETIC CARDS.
- * CAUTION: ENSURE THAT THE CORRECT CALCULATOR PARTITION IS SET BEFORE INPUTTING A PROGRAM AND WRITING TO MAGNETIC CARDS.
- * CAUTION: ENSURE THAT YOU DO NOT CONFUSE BANK NUMBERS WITH CARD/MODULE OR SIDE NUMBERS. THE NUMBERS WHICH REFER TO THE LISTING ARE AKIN TO A VIRTUAL ADDRESS AND DO NOT REPRESENT THE ACTUAL BANK NUMBER. IF IN DOUBT, REMEMBER TO USE THE TABLE BELOW TO TRANSLATE VIRTUAL TO ACTUAL BANK NUMBERS.

VIRTUAL BANK	ACTUAL BANK
MODULE #	
CARD1	
SIDE1 -----	BANK1
MODULE #	
CARD1	
SIDE2 -----	BANK2
MODULE #	
CARD2	
SIDE1 -----	BANK3

=====

TI-59 LISTING

=====

\$81

=====

TI-59 PROGRAM SPECIFIC INSTRUCTIONS:

=====

- * THE FOLLOWING INFORMATION WILL TELL YOU HOW TO RUN YOUR PROGRAM.
- * YOU MUST ENTER YOUR PROGRAM MANUALLY INTO THE CALCULATOR AND WRITE THE PROGRAM TO MAGNETIC CARDS. THIS STEP ONLY NEEDS TO BE ACCOMPLISHED ONCE. AFTER THAT, THE PROGRAM IS ENTERED USING THE MAGNETIC CARD FACILITY OF THE CALCULATOR. SEE THE MANUFACTURER'S LITERATURE ON ENTERING A PROGRAM AND WRITING IT TO MAGNETIC CARDS. YOU WILL NEED TO PARTITION MEMORY.

- * HOW TO PARTITION THE MEMORY
 - * KEY SEQUENCE:

X
2ND
OP
17

- * X IS THE PARTITION NUMBER GIVEN IN THE LISTING OF YOUR PROGRAM.
- * WHEN TO PARTITION THE MEMORY
 - * ONCE BEFORE READING IN CARDS.
 - * ONCE BEFORE MANUALLY ENTERING PROGRAM IN ORDER TO WRITE TO CARDS.

- * HOW TO START AND RUN YOUR PROGRAM
 - * TURN ON CALCULATOR
 - * PARTITION CALCULATOR
 - * LOAD ALL MODULE 1 CARDS
 - * OPTIONAL STEP: IF YOU SELECTED THE MANUAL DATA INPUT DENOTED IN YOUR BASIC PROGRAM BY USING THE "DATA" AND "READ" STATEMENTS THEN YOU MUST MANUALLY ENTER YOUR DATA INTO THE CALCULATOR MEMORY. THIS IS DONE BY REFERRING TO "INPUT DATA TO READ" TABLE PROVIDED AT THE END OF THIS LISTING. MANUALLY ENTER THE GIVEN DATA INTO THE REGISTERS USING THE FOLLOWING KEYSTROKES:

DATA
STO
XX

- * WHERE XX IS THE DESIRED REGISTER NUMBER.
- * INITIALIZE THE MANUAL SBR RETURN CONTROL STACK WITH THE FOLLOWING KEYSTROKES:

XX
STO
08

- * WHERE XX IS THE MANUAL RETURN REGISTER STACK TOP. (THIS IS GIVEN WITH THE PROGRAM LISTING NEAR THE PARTITION INFORMATION.)
- * PRESS "A" TO START.
- * FOLLOW DISPLAY PROMPTS.
- * DEFINITIONS:
 - * RUN-TIME PROMPTS: ARE DEFINED TO BE CALCULATOR PROMPTS DISPLAYED IN THE CALCULATOR WINDOW IN THE FORM OF A 4 DIGIT DECIMAL, 2 DIGIT INTEGER OR A 1 DIGIT INTEGER. EACH PROMPT

IS OUTLINED BELOW:

- * 4 DIGIT DECIMAL
 - * FORMAT: X.YYY
 - * X STANDS FOR MODULE NUMBER (1-9)
 - * YYY STANDS FOR STARTING ADDRESS
 - * ACTIONS:
 - * LOAD ALL MODULE X CARDS.
 - * PRESS FOLLOWING KEY SEQUENCE TO INITIALIZE:
RCL
OO
GTO
Y
Y
Y
 - * PRESS R/S TO CONTINUE IN NEW MOD.
- * 2 DIGIT INTEGER
 - * FORMAT: XX WHERE XX STANDS FOR A REGISTER NUMBER.
 - * ACTIONS:
 - * LOOK UP IN REGISTER MAP PROVIDED THE BASIC NAME THAT CORRESPONDS TO THE XX NUMBER.
 - * ENTER THE BASIC VARIABLE VALUE.
 - * PRESS R/S TO CONTINUE WITH THE ENTERED VALUE.
- * 1 DIGIT INTEGER
 - * FORMAT: X WHERE IS A MODULE NUMBER.
 - * ACTIONS:
 - * LOAD ALL MODULE X CARDS.
 - * PRESS FOLLOWING SEQUENCE TO INITIALIZE:
RCL
OO
INV
SBR
 - * PRESS R/S TO CONTINUE IN NEW MCD.
- * PAUSE IN DISPLAY
 - * AN UNFORMATTED DIGIT FLASHES IN THE DISPLAY BEFORE BEING DISPLAYED. THIS IS AN ANSWER THAT CORRESPONDS TO A REQUESTED ANSWER IN THE BASIC PROGRAM USING THE BASIC PRINT STATEMENT. THESE ANSWERS OCCUR IN THE SAME ORDER AS THEY WERE REQUESTED IN THE BASIC PROGRAM.
 - * ACTIONS: NOTE ANSWER AND PRESS R/S.
- * 888 IN DISPLAY
 - * SPECIFIC PROMPT THAT INDICATES THAT THE PROGRAM HAS STOPPED EXECUTION.
 - * ACTIONS: IF DESIRED FIND ANSWERS IN THE CALCULATOR MEMORY USING THE "TI-59 REGISTER TO NAME MAPPING" AT THE END OF THE INSTRUCTIONS.

* EXPECTED CONTROL FLOW PROMPTS BY MODULE FOLLOW:

\$82
\$83

=====
END BAX59 SEGMENTATION/INSTRUCTION: VERSION 1.0
=====

\$83
\$84

=====
BAX59 PROGRAM INSTRUCTIONS: VERSION 1.0
**** SEGMENTER FAILURE ***** PROGRAM FAILURE****
BAX59 DIAGNOSTICS FOLLOW:
=====

* SEGMENTER FAILURES:

- * THE SEGMENTOR COULD NOT SEGMENT THE COMPILED PROGRAM IN A SATISFACTORY MANNER. POSSIBLE REASONS FOR THE FAILURE ARE GIVEN BELOW.
- * THERE ARE TWO TYPES OF SEGMENT BREAKS:
 - * A JUMP BREAK OCCURS THROUGH AN ABSOLUTE JUMP TO SOME PORTION OF CODE IN ANOTHER MODULE.
 - * A SER BREAK OCCURS THROUGH A SBR INVOKE TO A SBR WHOSE DEFINITION RESIDES IN ANOTHER MODULE.
- * SEGMENT FAILURE OCCURS WHEN ONE OF THE ABOVE BREAKS OCCURS INSIDE A BACKWARD JUMPING LOOP THAT COVERS MORE PROGRAM STEPS THAN IS AVAILABLE IN THE CALCULATOR MEMORY. SEGMENTATION IS NOT ALLOWED IN A LOOP AS IT IS IMPRACTICAL TO KEEP READING IN CARDS EVERY TIME THE PROGRAM LOOPS BACK OVER A BREAK (IMAGINE A 1 TO 1000 LOOP OVER SUCH A BREAK). TO AVOID SUCH A PROBLEM YOU MUST STRUCTURE YOUR BASIC PROGRAM TO AVOID LARGE BACKWARD-JUMPING LOOPS.
- * PROGRAM FAILURES: POSSIBLE PROGRAM FAILURE OCCURS WHEN SUBROUTINE CALLS ARE NESTED GREATER THAN SIX DEEP. THE CALCULATOR ONLY HAS SIX SUBROUTINE RETURN REGISTERS.
- * BEICW ARE DIAGNOSTICS INDICATING THE SIZES OF THE LOOPS IN TI-59 PROGRAM STEPS AND THE TYPES OF BREAKS OCCURRING WITHIN THESE LOOPS. DIAGNOSTICS ARE GIVEN IN ABSOLUTE CODE. SER NESTING LEVEL DIAGNOSTICS ARE GIVEN FOR INVOKED ROUTINE DEFINITION.

\$84
\$5

=====

EAX59 VERSION 1.0
UNSEGMENTED ABSOLUTE COMPILED TI59 CODE FOLLOWS

=====

\$5
\$6

* SEQUENTIAL CONTINUATION: 4 DIGIT REAL CODE.

\$6
\$7

* MANUAL RETURN FROM A SUBROUTINE: 1 DIGIT CODE.

\$7
\$8

* FORWARD JUMP CONTINUATION: 4 DIGIT REAL CODE.

\$8
\$9

* SUBROUTINE INVOKE: 4 DIGIT REAL CODE.

\$9
\$99

0		1		2		3
4		5		6		7
8		9		2ND E'		A
E		C		D		E
2ND A'		2ND B'		2ND C'		2ND D'
2ND CIR		INV		LNK		CE
CLR		2ND INV		2ND LOG		2ND CP
2ND TAN		X<=>T		X**2		SORT(X)
1/X		2ND PGM		2ND P=>R		2ND SIN
2ND CCS		2ND INC		STO		RCL
SUM		Y**X		2ND CMS		2ND EXC
2ND PRD		X		EE		(
2ND INT		2ND DEG		2ND ENG		2ND FIX
2ND EXC	2ND IND	2ND PRG	2ND IND	GTO		2ND PGM 2ND IND
2ND X=T		2ND NCF		X		2ND PAUSE
SBR		STO 2ND IND		2ND OP		2ND RAD
-		2ND LEL		RCL 2ND IND		SUM 2ND IND
X-BAR		2ND GRAD		2ND X>=T		2ND SUMMATION
2ND OP 2ND IND		+		RST		GTO 2ND IND
2ND D.MS		2ND PI		2ND STFLG		2ND IFFLG
INV SBR		2ND DSZ		2ND LIST		R/S
2ND WRITE				+/-		=
				2ND ADV		2ND PRT

\$99
 \$100 * EXPECTED PROMPTS FOR MODULE # \$
 \$101
 * NONE
 \$101
 \$102 * \$
 \$102
 \$103 * YES
 \$103
 \$104
 * MANUAL RETURN REGISTER TOP IS \$
 \$104
 \$105
 STORE IN REGISTER: \$
 \$105
 \$106
 * PROGRAM PARTITION IS \$
 \$106
 \$107
 * PARTITION NUMBER IS \$
 \$107
 \$108
 *MODULE # \$
 \$108
 \$109 CARD #1
 \$109
 \$110 CARD #2
 \$110
 \$111 SIDE #1
 \$111
 \$112 SIDE #2
 \$112

APPENDIX K
ARTILLERY TEST PROGRAM SOURCE CODE

```

00005 OPTION 0 5
00010 REM *****
00011 REM
00012 REM
00014 REM
00015 REM *THIS TEST PROGRAM IS AN ADAPTATION OF THE PROGRAM
00016 REM USED BY THE FIELD ARTILLERY IN THE COMPUTATION
00017 REM OF FIRING DATA FOR THEIR GUNS. THE ORIGINAL
00018 REM PROGRAM WAS WRITTEN FOR THE TI-59 CALCULATOR.
00019 REM THIS TEST WAS CHOSEN NOT ONLY TO EVALUATE THE
00020 REM THE COMPILER AND SEGMENTOR BUT TO COMPARE THE
00022 REM THE RELATIVE EFFICIENCY OF THE TRANSLATED WEASIC
00023 REM PROGRAM WITH THAT OF A HUMAN CODED PROGRAM. BOTH
00024 REM PROGRAMS ACCOMPLISH THE SAME TASK.
00025 REM
00026 REM *****
00027 REM
00050 REM ***** DATA SECTION M109 *****
00055 REM
00100 REM *CHARGE CONSTANTS M109A1 SELF PROPELLED
00120 REM *CHARGE 4
00130 DATA -.0133670, 21.2691, -105.7
00140 DATA -.00001499, .06630, -.41
00150 DATA .77, .01314, .00001720
00160 REM *CHARGE 5
00170 DATA -.0149331, 24.3439, 64.7
00180 DATA -.00001420, .07069, .06
00190 DATA 1.26, .01508, .00001678
00200 REM *CHARGE 7
00210 DATA -.0173835, 29.8741, 2255.2
00220 DATA -.00001668, .08487, 3.29
00230 DATA 1.3, .02713, .00001306
00240 REM *CHARGE 8
00250 DATA -.0182137, 32.3731, 4107.4
00260 DATA -.00001668, .09272, 5.74
00270 DATA 1.36, .02891, .00001410
00280 REM
00290 REM *M109 MAX RANGE OF CURVE FIT BY CHARGE
00300 DATA 5700, 7000, 10800, 17600
00310 REM
00320 REM *M109 HIGH ANGLE CROSS OVER POINT MILS
00330 DATA 715
00340 REM
00350 REM *BATTERY DATA / BTRYE, BTRYN, BTRYA, BTRYL
00360 DATA 0, 0, 0, 800
00370 REM
00380 REM *REGISTRATION DATA / RNGK, DFCOR
00390 DATA 1.0, 0
00400 REM
00410 REM *TARGET DATA / OBSERVOR LOCATION (DUAL MEANING)
00420 DATA 4000, 4000, 10
00430 REM
00440 REM *OBSERVOR DATA
00450 DATA 4000, -400, 10
00460 REM
00470 REM *SPECIFIC CORRECTION FACTORS DATA
00471 DATA 1018.5924, 1600, 3200
00472 REM
00473 REM

```

```

00480 REM ***** VARIABLE READ INITIALIZATION *****
00490 REM
00495 REM *M109 FALLISTIC CONSTANTS BY CHARGE
00500 REM *CHARGE 4
00510 READ A24,A14,A04
00520 READ C24,C14,C04
00530 READ B04,B14,B24
00540 REM *CHARGE 5
00550 READ A25,A15,A05
00560 READ C25,C15,C05
00570 READ B05,B15,B25
00580 REM *CHARGE 7
00590 READ A27,A17,A07
00600 READ C27,C17,C07
00610 READ B07,B17,B27
00620 REM *CHARGE 8
00630 READ A28,A18,A08
00640 READ C28,C18,C08
00650 READ B08,B18,B28
00660 REM
00670 REM *M109 MAX RANGE OF CURVE FIT VARIABLES
00680 READ CHG4MAX,CHG5MAX,CHG7MAX,CHG8MAX
00690 REM
00700 REM *M109 HIGH ANGLE CROSS OVER VARIABLE
00710 READ HACRCSS
00720 REM
00730 REM *BATTERY VARIABLES
00740 READ ETRYE,ETRYN,BTRYA,BTRYL
00750 REM
00760 REM *REGISTRATION VARIABLES
00770 READ RGK,DFCOR
00780 REM
00790 REM *TARGET VARIABLES OR OBSERVOR INIT LOCATION
00800 READ GRIDE,GRIDN,GRIDA
00810 REM
00820 REM *OBSERVOR VARIABLES
00830 READ OT,LATDEV,RGDEV
00840 REM
00845 REM *SPECIFIC CORRECTION FACTORS VARIABLES
00860 READ MILRAD,ROTCOR,REFDEF
00870 REM
00875 REM *****
00880 REM ***** MAIN PROGRAM BEGINS *****
00900 REM START
00910 REM *COMPUTE TARGET GRID
00920 REM GOSUE 1050
00940 REM *COMPUTE GUN RANGE, AZIMUTH
00950 REM GOSUE 1130
00970 REM *COMPUTE FIRING DATA
00980 REM GOSUE 1240
00990 REM
01000 REM STOP
01010 REM ***** MAIN STOP *****
01020 REM
01022 REM
01030 REM ***** SUBROUTINES *****
01035 REM
01040 REM *****
01050 REM *** COMPUTE NEW TARGET GRID FROM SHIFTS ***
01051 REM *****
01055 REM START
01060 REM GRIDN = GRIDN+(RGDEV*SIN((ROTCOR-OT)/MILRAD) &
01070 REM & MILRAD)-LATDEV*COS((ROTCOR-OT)/MILRAD) &
01080 REM GRIDE = GRIDE+(RGDEV*CCS((ROTCOR-OT)/MILRAD) &
01090 REM & MILRAD)+LATDEV*SIN((ROTCOR-OT)/MILRAD) &
01100 REM RETURN
01110 REM *****
01115 REM *****
01120 REM *****

```

```

01130 REM **** COMPUTE GUN RANGE, AZIMUTH *****
01131 REM *****
01135 REM START
01140 TGTRG = SQRT((GRIDE-BTRYE)**2+
01145 & (GRIDN-BTRYN)**2)
01150 & TGTAZ = ASIN((GRIDN-BTRYN)/TGTRG)*MILRAD
01160 IF GRIDE >= BTRYE
01170 TGTAZ = ROTCOR - TGTAZ
01180 ELSE
01190 TGTAZ = 3*ROTCOR + TGTAZ
01200 ENDIF
01210 RETURN
01220 REM *****
01223 REM *****
01230 REM *****
01240 REM **** FIRING DATA COMPUTATION ROUTINE *****
01245 REM *****
01250 REM START
01260 IF TGTRG <= CHG4MAX
01270 INVOKE= FN_FD(A24,A14,A04,C24,C14,C04, &
01275 & B24,E14,B04)
01280 ELSEIF TGTRG <= CHG5MAX
01290 INVOKE= FN_FD(A25,A15,A05,C25,C15,C05, &
01295 & B25,E15,B05)
01300 ELSEIF TGTRG <= CHG7MAX
01310 INVOKE= FN_FD(A27,A17,A07,C27,C17,C07, &
01315 & B27,E17,B07)
01320 ELSEIF TGTRG <= CHG8MAX
01330 INVOKE= FN_FD(A28,A18,A08,C28,C18,C08, &
01335 & B28,E18,B08)
01340 ELSE
01350 PRINT TGTRG
01360 ENDIF
01370 RETURN
01380 REM *****
01390 REM *****
01395 REM *****
01400 REM *** FIRING DATA COMPUTATION FUNCTION *****
01405 REM *****
01410 REM START
01420 DEF FN_FD(A2,A1,A0,C2,C1,C0,B2,B1,B0)
01430 EL = (-A1+SQRT(A1**2-(4*A2*(A0-TGTRG*RGK) &
01435 & )))/(2*A2)
01440 IF EL > HACROSS
01450 PRINT TGTAZ,TGTRG
01460 ELSE
01470 PRINT C0+C1*EL+C2*EL**2
01480 PRINT REFDEF+DFCOR+(BTRYL-TGTAZ)+ &
01481 & (B0+B1*EL+B2*EL**2)
01490 PRINT EL+((GRIDA-BTRYA+20)/TGTRG*1000)
01500 ENDIF
01510 FNEND
01515 REM *****
01516 REM END FAX59 TEST PROGRAM NUMBER ONE
01518 REM *****

```

APPENDIX L
 TEST PROGRAM LISTING FILE (LISTF)

=====

 WBASIC PROGRAM LISTING

 =====

```

00005 OPTICN 0 5
00010 REM *****
00011 REM
00012 REM
00014 REM
00015 REM *THIS TEST PROGRAM IS AN ADAPTATION OF THE PROGRAM
00016 REM USED BY THE FIELD ARTILLERY IN THE COMPUTATION
00017 REM OF FIRING DATA FOR THEIR GUNS. THE ORIGINAL
00018 REM PROGRAM WAS WRITTEN FOR THE TI-59 CALCULATOR.
00019 REM THIS TEST WAS CHOSEN TO NOT ONLY EVALUATE THE
00020 REM THE COMPILER AND SEGMENTOR BUT TO COMPARE THE
00022 REM THE RELATIVE EFFICIENCY OF THE TRANSLATED WEASIC
00023 REM PROGRAM WITH THAT OF A HUMAN CODED PROGRAM. BOTH
00024 REM PROGRAMS ACCOMPLISH THE SAME TASK.
00025 REM
00026 REM *****
00027 REM
00050 REM ***** DATA SECTION M109 *****
00055 REM
00100 REM *CHARGE CONSTANTS M109A1 SELF PROPELLED
00120 REM *CHARGE 4
00130 DATA -.0133670, 21.2691, -105.7
00140 DATA -.00001499, .06630, -.41
00150 DATA .77, .01314, .00001720
00160 REM *CHARGE 5
00170 DATA -.0149331, 24.3439, 64.7
00180 DATA -.00001420, .07069, .06
00190 DATA 1.26, .01508, .00001678
00200 REM *CHARGE 7
00210 DATA -.0173835, 29.8741, 2255.2
00220 DATA -.00001668, .08487, 3.29
00230 DATA 1.3, .02713, .00001306
00240 REM *CHARGE 8
00250 DATA -.0182137, 32.3731, 4107.4
00260 DATA -.00001668, .09272, 5.74
00270 DATA 1.36, .02891, .00001410
00280 REM
00290 REM *M109 MAX RANGE OF CURVE FIT BY CHARGE
00300 DATA 5700, 7000, 10800, 17600
00310 REM
00320 REM *M109 HIGH ANGLE CROSS OVER POINT MILS
00330 DATA 715
00340 REM
00350 REM *BATTERY DATA/ BTRYE, BTRYN, BTRYA, BTRYL
00360 DATA 0, 0, 0, 800
00370 REM
00380 REM *REGISTRATION DATA/RNGK, DFCOR
00390 DATA 1.0, 0
00400 REM
00410 REM *TARGET DATA/CBSERVOR LOCATION (EUAL MEANING)
00420 DATA 4000, 4000, 10
00430 REM
00440 REM *OESERVOR DATA
00450 DATA 4000, -400, 10
00460 REM
  
```

```

00470 REM      *SPECIFIC CORRECTION FACTORS DATA
00471 DATA      1018.5924,1600,3200
00472 REM
00473 REM
00480 REM      ***** VARIABLE READ INITIALIZATION *****
00490 REM
00495 REM      *M109 EALLISTIC CONSTANTS BY CHARGE
00500 REM      *CHARGE 4
00510 READ      A24,A14,A04
00520 READ      C24,C14,C04
00530 READ      B04,B14,B24
00540 REM      *CHARGE 5
00550 READ      A25,A15,A05
00560 READ      C25,C15,C05
00570 READ      B05,B15,B25
00580 REM      *CHARGE 7
00590 READ      A27,A17,A07
00600 READ      C27,C17,C07
00610 READ      B07,B17,B27
00620 REM      *CHARGE 8
00630 READ      A28,A18,A08
00640 READ      C28,C18,C08
00650 READ      B08,B18,B28
00660 REM
00670 REM      *M109 MAX RANGE OF CURVE FIT VARIABLES
00680 READ      CHG4MAX,CHG5MAX,CHG7MAX,CHG8MAX
00690 REM
00700 REM      *M109 HIGH ANGLE CROSS OVER VARIABLE
00710 READ      HACRCSS
00720 REM
00730 REM      *BATTERY VARIABLES
00740 READ      ETRYE,ETRYN,BTRYA,BTRYL
00750 REM
00760 REM      *REGISTRATION VARIABLES
00770 READ      RGK,DFCOR
00780 REM
00790 REM      *TARGET VARIABLES OR OBSERVOR INIT LOCATION
00800 READ      GRIDE,GRIDN,GRIDA
00810 REM
00820 REM      *OBSERVOR VARIABLES
00830 READ      OT,LATDEV,RGDEV
00840 REM
00845 REM      *SPECIFIC CORRECTION FACTORS VARIABLES
00860 READ      MILRAD,ROTCOR,REFDEF
00870 REM
00875 REM      *****
00880 REM      ***** MAIN PROGRAM BEGINS *****
00900 REM      START
00910 REM      *COMPUTE TARGET GRID
00920 REM      GOSUE 1050
00940 REM      *COMPUTE GUN RANGE,AZIMUTH
00950 REM      GOSUE 1130
00970 REM      *COMPUTE FIRING DATA
00980 REM      GOSUE 1240
00990 REM
01000 REM      STOP
01010 REM      ***** MAIN STOP *****
01020 REM
01022 REM
01030 REM      ***** SUBROUTINES *****
01035 REM
01040 REM      *****
01050 REM      *** COMPUTE NEW TARGET GRID FROM SHIFTS ***
01051 REM      *****
01055 REM      START
01060 REM      GRIDN = GRIDN+(RGDEV*SIN((ROTCOR-OT)/
01070 &          MILRAD)-LATDEV*COS((ROTCOR-OT)/MILRAD)
01080 REM      GRIDE = GRIDE+(RGDEV*COS((ROTCOR-OT)/
01090 &          MILRAD)+LATDEV*SIN((ROTCOR-OT)/MILRAD)

```

```

01100      RETURN
01110 REM *****
01115 REM *****
01120 REM *****
01130 REM ***** COMPUTE GUN RANGE, AZIMUTH *****
01131 REM *****
01135 REM      START
01140      TGTRG = SQR((GRIDE-BTRYE)**2+
01145 &      (GRIDN-BTRYN)**2) &
01150      TGTAZ = ASIN((GRIDN-BTRYN)/TGTRG)*MILRAD
01160      IF GRIDE >= BTRYE
01170          TGTAZ = ROTCOR - TGTAZ
01180      ELSE
01190          TGTAZ = 3*ROTCOR + TGTAZ
01200      ENDIF
01210      RETURN
01220 REM *****
01223 REM *****
01230 REM *****
01240 REM ***** FIRING DATA COMPUTATION ROUTINE *****
01245 REM *****
01250 REM      START
01260      IF TGTRG <= CHG4MAX
01270          INVCKE= FN_FD(A24,A14,A04,C24,C14,C04, &
01275 &          B24,E14,B04) &
01290      ELSEIF TGTRG <= CHG5MAX
01295 &          INVCKE= FN_FD(A25,A15,A05,C25,C15,C05, &
01300 &          B25,E15,B05) &
01310      ELSEIF TGTRG <= CHG7MAX
01315 &          INVCKE= FN_FD(A27,A17,A07,C27,C17,C07, &
01320 &          B27,E17,B07) &
01330      ELSEIF TGTRG <= CHG8MAX
01335 &          INVCKE= FN_FD(A28,A18,A08,C28,C18,C08, &
01340 &          B28,E18,B08) &
01350      ELSE
01355          PRINT TGTRG
01360      ENDIF
01370      RETURN
01380 REM *****
01390 REM *****
01395 REM *****
01400 REM ***** FIRING DATA COMPUTATION FUNCTION *****
01405 REM *****
01410 REM      START
01420      DEF FN_FD(A2,A1,A0,C2,C1,C0,B2,B1,B0)
01430      EI = (-A1+SQR(A1**2-(4*A2*(A0-TGTRG*RGK) &
01435 &      )))/(2*A2) &
01440      IF EI > HACROSS
01450          PRINT TGTAZ,TGTRG
01460      ELSE
01470          PRINT C0+C1*EI+C2*EI**2
01480          PRINT REFDEF+DFCOR+ (BTRYL-TGTAZ) + &
01481 &          (B0+B1*EI+B2*EI**2)) &
01490          PRINT EI+ ((GRIDA-BTRYA+20)/TGTRG*1000)
01500      ENDEF
01510      FNEND
01515 REM *****
01516 REM ***** END FAX59 TEST PROGRAM NUMBER ONE *****
01518 REM *****

```

=====
 COMPILATION SUMMARY
 =====

0 FATAL ERRORS.
 0 WARNING MSGS.

81 IS NEXT AVAILABLE REGISTER
 TOTAL REGISTERS RESERVED = 11

TOTAL REGISTERS USED = 70
 TOTAL LABELS USED = 5

----- COMPILATION TERMINATES -----

=====
 TI-59 CODE TRANSLATED FROM WBASIC
 (UNSEGMENTED)
 =====

\$0	ADDR	CODE	
	000	76	2ND LBL
	001	11	A
	002	71	SBR
	003	12	B
	004	68	2ND NCP
	005	71	SBR
	006	13	C
	007	68	2ND NCP
	008	71	SBR
	009	14	D
	010	68	2ND NCP
	011	24	CE
	012	08	8
	013	08	8
	014	08	8
	015	91	R/S
	016	76	2ND LBL
	017	12	B
	018	53	(
	019	43	RCL
	020	59	+
	021	59	+
	022	33	(
	023	43	RCL
	024	63	*
	025	63	*
	026	33	(
	027	53	RCL
	028	43	RCL
	029	65	-
	030	75	-
	031	43	RCL
	032	61)
	033	54)
	034	55	/
	035	43	RCL
	036	64)
	037	54)
	038	58	2ND SIN
	039	75	-
	040	43	RCL
	041	62	*
	042	65	*
	043	53	(
	044	53	(
	045	43	RCL
	046	65	-
	047	75	-
	048	43	RCL
	049	61)
	050	54)
	051	55	/
	052	43	RCL
	053	64)
	054	54)
	055	39	2ND CCS

056	554	}	
057	544	}	
058	422	STO	
059	593	59	
060	533	(
061	433	RCL	
062	588	58	
063	853	+	
064	533	(
065	433	RCL	
066	633	63	
067	653	*	
068	533	(
069	533		
070	433	RCL	
071	655	65	
072	533	-	
073	433	RCL	
074	611	61	
075	544)	
076	555	/	
077	433	RCL	
078	644	64	
079	544)	
080	299	2ND	CCS
081	855	+	
082	433	RCL	
083	622	62	
084	655	*	
085	533	(
086	533		
087	433	RCL	
088	655	65	
089	755	-	
090	433	RCL	
091	611	61	
092	554)	
093	555	/	
094	433	RCL	
095	644	64	
096	544)	
097	388	2ND	SIN
098	544	}	
099	544	}	
100	422	STO	
101	588	58	
102	922	INV	SBR
103	766	2ND	IEL
104	133	C	
105	533	(
106	533		
107	533		
108	433	RCL	
109	588	58	
110	755	-	
111	433	RCL	
112	522	52	
113	544)	
114	455	Y**X	
115	022	2	
116	555	+	
117	533	(
118	433	RCL	
119	599	59	
120	755	-	
121	433	RCL	
122	533	53	
123	544)	
124	455	Y**X	

125	0	2
126	4)
127	4	SQRT (X)
128	4)
129	4	STO
130	6	67
131	3	{
132	3	{
133	3	{
134	4	RCL
135	5	59
136	7	-
137	4	RCL
138	5	53
139	5)
140	5	/
141	4	RCL
142	6	67
143	5)
144	2	2ND INV
145	3	2ND SIN
146	6	*
147	4	RCL
148	6	64
149	5)
150	4	STO
151	6	68
152	4	RCL
153	5	58
154	4	X<=>T
155	4	RCL
156	5	52
157	3	X<=>T
158	2	INV
159	7	2ND X>=T
160	0	01
161	7	74
162	5	(
163	4	RCL
164	6	65
165	7	-
166	4	RCL
167	6	68
168	5)
169	4	STO
170	6	68
171	6	GT0
172	0	01
173	8	85
174	3	(
175	3	3
176	6	*
177	4	RCL
178	6	65
179	8	+
180	4	RCL
181	6	68
182	5)
183	4	STO
184	6	68
185	9	INV SER
186	7	2ND LBL
187	1	D
188	4	RCL
189	6	67
190	3	X<=>T
191	4	RCL
192	4	47
193	2	INV

194	77	2ND X>=T
195	02	02
196	44	44
197	53	(
198	43	RCL
199	11	11
200	42	STO
201	71	71
202	43	RCL
203	12	12
204	42	STO
205	72	72
206	43	RCL
207	13	13
208	42	STO
209	73	73
210	43	RCL
211	14	14
212	42	STO
213	74	74
214	43	RCL
215	15	15
216	42	STO
217	75	75
218	43	RCL
219	16	16
220	42	STO
221	76	76
222	43	RCL
223	19	19
224	42	STO
225	77	77
226	43	RCL
227	18	18
228	42	STO
229	78	78
230	43	RCL
231	17	17
232	42	STO
233	79	79
234	53	(
235	71	SBR
236	15	E
237	54)
238	54)
239	42	STO
240	69	69
241	61	GTO
242	04	04
243	16	16
244	43	RCL
245	67	67
246	32	X<=>I
247	43	RCL
248	48	48
249	22	INV
250	77	2ND X>=T
251	03	03
252	00	00
253	53	(
254	43	RCL
255	20	20
256	42	STO
257	71	71
258	43	RCL
259	21	21
260	42	STO
261	72	72
262	43	RCL

263	22	22
264	42	STO
265	73	73
266	43	RCL
267	23	23
268	42	STO
269	74	74
270	43	RCL
271	24	24
272	42	STO
273	75	75
274	43	RCL
275	25	25
276	42	STO
277	76	76
278	43	RCL
279	28	28
280	42	STO
281	77	77
282	43	RCL
283	27	27
284	42	STO
285	78	78
286	43	RCL
287	26	26
288	42	STO
289	79	79
290	53	(
291	71	SBR
292	15	F
293	54)
294	54)
295	42	STO
296	69	69
297	61	GTO
298	04	04
299	16	16
300	43	RCL
301	67	67
302	32	X<=>T
303	43	RCL
304	49	49
305	22	INV
306	77	2ND X>=T
307	03	03
308	56	56
309	53	(
310	43	RCL
311	29	29
312	42	STO
313	71	71
314	43	RCL
315	30	30
316	42	STO
317	72	72
318	43	RCL
319	31	31
320	42	STO
321	73	73
322	43	RCL
323	32	32
324	42	STO
325	74	74
326	43	RCL
327	33	33
328	42	STO
329	75	75
330	43	RCL
331	34	34

332	42	STO	
333	76	76	
334	43	RCL	
335	37	37	
336	42	STO	
337	77	77	
338	43	RCL	
339	36	36	
340	42	STO	
341	78	78	
342	43	RCL	
343	35	35	
344	42	STO	
345	79	79	
346	53	(
347	71	SBR	
348	15	E	
349	54	}	
350	54	STO	
351	42	69	
352	69	GT O	
353	61	04	
354	04	16	
355	16	RCL	
356	43	67	
357	67	X<=>T	
358	32	RCL	
359	43	50	
360	50	INV	
361	22	2ND	X>=T
362	77	04	
363	04	12	
364	12	(
365	53	RCL	
366	43	38	
367	38	STO	
368	42	71	
369	71	RCL	
370	43	39	
371	39	STO	
372	42	72	
373	72	RCL	
374	43	40	
375	40	STO	
376	42	73	
377	73	RCL	
378	43	41	
379	41	STO	
380	42	74	
381	74	RCL	
382	43	42	
383	42	STO	
384	42	75	
385	75	RCL	
386	43	43	
387	43	STO	
388	42	76	
389	76	RCL	
390	43	46	
391	46	STO	
392	42	77	
393	77	RCL	
394	43	45	
395	45	STO	
396	42	78	
397	78	RCL	
398	43	44	
399	44	STO	
400	42		

401	79	79	
402	53	(
403	71	SBR	
404	15	E	
405	54)	
406	54)	
407	42	STO	
408	69	69	
409	61	GTO	
410	04	04	
411	16	16	
412	43	RCL	
413	67	67	
414	99	2ND	PRT
415	98	2ND	ACV
416	92	INV	SBR
417	76	2ND	LEL
418	15	E	
419	00	0	
420	42	STO	
421	70	70	
422	53	(
423	53	(
424	43	RCL	
425	72	72	
426	94	+/-	
427	85	+	
428	53	(
429	43	RCL	
430	72	72	
431	45	Y**X	
432	02	2	
433	75	-	
434	53	(
435	04	4	
436	65	*	
437	43	RCL	
438	71	71	
439	65	*	
440	53	(
441	43	RCL	
442	73	73	
443	75	-	
444	43	RCL	
445	67	67	
446	65	*	
447	43	RCL	
448	56	56	
449	54)	
450	54)	
451	54)	
452	24	SQRT (X)	
453	54)	
454	55	/	
455	53	(
456	02	2	
457	65	*	
458	43	RCL	
459	71	71	
460	54)	
461	54)	
462	42	STO	
463	80	80	
464	43	RCL	
465	80	80	
466	32	X<=>T	
467	43	RCL	
468	51	51	
469	77	2ND	X>=T

470	04	04
471	82	82
472	43	RCL
473	68	68
474	99	2ND PRT
475	43	RCL
476	67	67
477	99	2ND FRT
478	98	2ND ADV
479	61	GTO
480	05	05
481	65	65
482	53	(
483	43	RCL
484	76	76
485	85	+
486	43	RCL
487	75	75
488	65	*
489	43	RCL
490	80	80
491	85	+
492	43	RCL
493	74	74
494	65	*
495	43	RCL
496	80	80
497	45	Y**X
498	02	2
499	54)
500	99	2ND PRT
501	98	2ND ADV
502	53	(
503	43	RCL
504	66	66
505	85	+
506	43	RCL
507	57	57
508	85	+
509	53	(
510	43	RCL
511	55	55
512	75	-
513	43	RCL
514	68	68
515	54)
516	85	+
517	53	(
518	43	RCL
519	79	79
520	85	+
521	43	RCL
522	78	78
523	65	*
524	43	RCL
525	80	80
526	85	+
527	43	RCL
528	77	77
529	65	*
530	43	RCL
531	80	80
532	45	Y**X
533	02	2
534	54)
535	54)
536	99	2ND PRT
537	98	2ND ADV
538	53	(

```

539 43 RCL
540 80 80
541 85 +
542 53 {
543 53 }
544 43 RCL
545 60 60
546 75 -
547 43 RCL
548 54 54
549 85 +
550 02 2
551 00 0
552 54 )
553 53 /
554 43 RCL
555 67 67
556 65 *
557 01 1
558 00 0
559 00 0
560 00 0
561 54 )
562 54 }
563 99 2ND PRT
564 98 2ND ADV
565 43 RCL
566 70 70
567 92 INV SER

```

```

=====
BAX59 SYMBOL TABLE DUMP
=====

```

BUCKET	CONTENTS	REG	TYP
03	REFDEF	66	GLOBAL VAR
05	LOG10	..	QUICK FN
06	GRIDA	60	GLOBAL VAR
08	EL	80	GLOBAL VAR
10	GRIDE	58	GLOBAL VAR
13	FP	..	QUICK FN
	ABS	..	QUICK FN
14	FN_FD	70	PARAMETER FN
16	PI	..	CONSTANT
	IP	..	QUICK FN
	CSC	..	QUICK FN
18	SEC	..	QUICK FN
20	DFCOR	57	GLOBAL VAR
24	LOG	..	QUICK FN
26	GRIDN	59	GLOBAL VAR
	RGK	56	GLOBAL VAR
	RND	10	LONG FN
28	ACOS	..	QUICK FN
30	MILRAD	64	GLOBAL VAR

33	TAN	..	QUICK FN
	ATN	::	QUICK FN
	ASIN	::	QUICK FN
35			
36	COS	..	QUICK FN
38	COT	..	QUICK FN
40	RGDEV	63	GLOBAL VAR
41	SIN	..	QUICK FN
43	OT	61	GLOBAL VAR
53	EXP	..	QUICK FN
59	LATDEV	62	GLOBAL VAR
63	SQR	..	QUICK FN
64	BTRYA	54	GLOBAL VAR
67	INVOKE	69	GLOBAL VAR
69	BTRYE	52	GLOBAL VAR
71	TGTRG	67	GLOBAL VAR
74	HACROSS	51	GLOBAL VAR
75	CHG 4 MAX	47	GLOBAL VAR
77	CHG 5 MAX	48	GLOBAL VAR
	CHG 7 MAX	49	GLOBAL VAR
	A04	13	GLOBAL VAR
78			
	CHG 8 MAX	50	GLOBAL VAR
	A05	22	GLOBAL VAR
	E04	17	GLOBAL VAR
	A 14	12	GLOBAL VAR
79			
	TGTA Z	68	GLOBAL VAR
	E05	26	GLOBAL VAR
	A 15	21	GLOBAL VAR
	E 14	18	GLOBAL VAR
	C04	16	GLOBAL VAR
	A24	11	GLOBAL VAR
80			
	A07	31	GLOBAL VAR
	B 15	27	GLOBAL VAR
	C05	25	GLOBAL VAR
	A25	20	GLOBAL VAR
	B24	19	GLOBAL VAR
	C 14	15	GLOBAL VAR
81			
	BTRYL	55	GLOBAL VAR
	A08	40	GLOBAL VAR
	E07	35	GLOBAL VAR
	A 17	30	GLOBAL VAR
	B25	28	GLOBAL VAR
	C 15	24	GLOBAL VAR
	C24	14	GLOBAL VAR
82			
	E08	44	GLOBAL VAR
	A 18	39	GLOBAL VAR
	B 17	36	GLOBAL VAR
	C07	34	GLOBAL VAR

	A27	29	GLOBAL VAR
	C25	23	GLOBAL VAR
83	-----		
	BTRY N	53	GLOBAL VAR
	B18	45	GLOBAL VAR
	C08	43	GLOBAL VAR
	A28	38	GLOBAL VAR
	E27	37	GLOBAL VAR
	C17	33	GLOBAL VAR
84	-----		
	ROTCOR	65	GLOBAL VAR
	B28	46	GLOBAL VAR
	C18	42	GLOBAL VAR
	C27	32	GLOBAL VAR
85	-----		
	C28	41	GLOBAL VAR

APPENDIX 3

TEST PROGRAM NAME MAPPING FILE (NAMEF)

=====

TI-59 REGISTER TO NAME MAPPING

=====

REG#	BASIC NAME
11	A24
12	A14
13	A04
14	C24
15	C14
16	C04
17	B04
18	B14
19	B24
20	A25
21	A15
22	A05
23	C25
24	C15
25	C05
26	B05
27	B15
28	B25
29	A27
30	A17
31	A07
32	C27
33	C17
34	C07
35	B07
36	B17
37	B27
38	A28
39	A18
40	A08
41	C28
42	C18
43	C08
44	B08
45	B18
46	B28
47	CHG4MAX
48	CHG5MAX
49	CHG7MAX
50	CHG8MAX
51	HACROSS
52	BTRYE
53	BTRYN
54	BTRYA
55	BTRYL
56	RGK
57	UFOR
58	GRIDE
59	GRIDN
60	GRIDA
61	OT
62	LATDEV
63	RGDEV

64 MILRAD
65 ROTCOR
66 REFDEF
67 TGTRG
68 TGTAZ
69 INVOKE
70 FN FD
71 (FN PARAMETER)
72 (FN PARAMETER)
73 (FN PARAMETER)
74 (FN PARAMETER)
75 (FN PARAMETER)
76 (FN PARAMETER)
77 (FN PARAMETER)
78 (FN PARAMETER)
79 (FN PARAMETER)
80 EL

APPENDIX N
 TEST PROGRAM DATA/READ MAPPING FILE (READF)

=====

INPUT DATA TO READ MAPPING

=====

DATA	REG	NAME
- .0133670	11	A24
21.2691	12	A14
-105.7	13	A04
- .00001499	14	C24
.06630	15	C14
-.41	16	C04
.77	17	B04
.01314	18	B14
.00001720	19	B24
- .0149331	20	A25
24.3439	21	A15
64.7	22	A05
- .00001420	23	C25
.07069	24	C15
.06	25	C05
1.26	26	B05
.01508	27	B15
.00001678	28	B25
- .0173835	29	A27
29.8741	30	A17
2255.2	31	A07
- .00001668	32	C27
.08487	33	C17
3.29	34	C07
1.3	35	B07
.02713	36	B17
.00001306	37	B27
- .0182137	38	A28
32.3731	39	A18
4107.4	40	A08
- .00001668	41	C28
.09272	42	C18
5.74	43	C08
1.36	44	B08
.02891	45	B18
.00001410	46	B28
5700	47	CHG4 MAX
7000	48	CHG5 MAX
10800	49	CHG7 MAX
17600	50	CHG8 MAX

715	51	HACROSS
0	52	BTRYE
0	53	BTRYN
0	54	BTRYA
800	55	BTRYL
1.0	56	RGK
0	57	DFCOR
4000	58	GRIDE
4000	59	GRIDN
10	60	GRIDA
4000	61	OT
-400	62	LATDEV
10	63	RGDEV
1018.5924	64	MILRAD
1600	65	ROTCOR
3200	66	REFDEF

APPENDIX Q
 TEST PROGRAM LINK INTERFACE FILE (SCRATCH)

\$1
 81 IS NEXT AVAILABLE REG.
 \$1

\$2

=====

TI-59 CCDE TRANSLATED FROM WBASIC
 (UNSEGMENTED)

=====

\$0	ADDR	CODE	-----	BEGIN TI-59 CODE.
	000	76	2ND	LEL
	001	11	A	
	002	71	SBR	
	003	12	B	
	004	68	2ND	NOP
	005	71	SBR	
	006	13	C	
	007	68	2ND	NOP
	008	71	SBR	
	009	14	D	
	010	68	2ND	NCP
	011	24	CE	
	012	08	8	
	013	08	8	
	014	08	8	
	015	91	R/S	
	016	76	2ND	LEL
	017	12	B	
	018	53	(
	019	43	RCL	
	020	59	59	
	021	55	+	
	022	55	(
	023	43	RCL	
	024	63	63	
	025	55	*	
	026	55	(
	027	55	{	
	028	43	RCL	
	029	65	65	
	030	75	-	
	031	43	RCL	
	032	61	61	
	033	54)	
	034	55	/	
	035	43	RCL	
	036	64	64	
	037	54	}	
	038	38	2ND	SIN
	039	75	-	
	040	43	RCL	
	041	62	62	
	042	55	*	
	043	55	(
	044	53	{	
	045	43	RCL	

046		65
047		-
048		RCL
049		61
050)
051		/
052		RCL
053		64
054)
055		2ND CCS
056)
058		STO
059		59
060		(
061		RCL
062		58
063		+
064		(
065		RCL
066		63
067		*
068		{
069		{
070		RCL
071		65
072		-
073		RCL
074		61
075)
076		/
077		RCL
078		64
079)
080		2ND COS
081		+
082		RCL
083		62
084		*
085		{
086		{
087		RCL
088		65
089		-
090		RCL
091		61
092)
093		/
094		RCL
095		64
096)
097		2ND SIN
098)
099)
100		STO
101		58
102		INV SER
103		2ND LBL
104		C
105		{
106		{
107		{
108		RCL
109		58
110		-
111		RCL
112		52
113)
114		Y**X

115	00	2
116	85	+
117	53	(
118	43	RCL
119	59	59
120	75	-
121	43	RCL
122	53	53
123	54)
124	45	Y**X
125	02	2
126	54)
127	34	SQRT (X)
128	54)
129	42	STO
130	67	67
131	53	(
132	53	{
133	53	{
134	43	RCL
135	59	59
136	75	-
137	43	RCL
138	53	53
139	54)
140	55	/
141	43	RCL
142	67	67
143	54)
144	27	2ND INV
145	38	2ND SIN
146	65	*
147	43	RCL
148	64	64
149	54)
150	42	STO
151	68	68
152	43	RCL
153	58	58
154	32	X<=>T
155	43	RCL
156	52	52
157	32	X<=>T
158	22	INV
159	77	2ND X>=T
160	01	01
161	74	74
162	53	(
163	43	RCL
164	65	65
165	75	-
166	43	RCL
167	68	68
168	54)
169	42	STO
170	68	68
171	61	GTO
172	01	01
173	85	85
174	53	(
175	03	3
176	65	*
177	43	RCL
178	65	65
179	85	+
180	43	RCL
181	68	68
182	54)
183	42	STO

184	68	58	
185	92	INV	SBR
186	76	2ND	LEL
187	14	D	
188	43	RCL	
189	67	67	
190	32	X<=>T	
191	43	RCL	
192	47	47	
193	22	INV	
194	77	2ND	X>=T
195	02	02	
196	44	44	
197	53	(
198	43	RCL	
199	11	11	
200	42	STO	
201	71	71	
202	43	RCL	
203	12	12	
204	42	STO	
205	72	72	
206	43	RCL	
207	13	13	
208	42	STO	
209	73	73	
210	43	RCL	
211	14	14	
212	42	STO	
213	74	74	
214	43	RCL	
215	15	15	
216	42	STO	
217	75	75	
218	43	RCL	
219	16	16	
220	42	STO	
221	76	76	
222	43	RCL	
223	19	19	
224	42	STO	
225	77	77	
226	43	RCL	
227	18	18	
228	42	STO	
229	78	78	
230	43	RCL	
231	17	17	
232	42	STO	
233	79	79	
234	53	(
235	71	SBR	
236	15	F	
237	54)	
238	54)	
239	42	STO	
240	69	69	
241	61	GTO	
242	04	04	
243	16	16	
244	43	RCL	
245	67	67	
246	32	X<=>T	
247	43	RCL	
248	48	48	
249	22	INV	
250	77	2ND	X>=T
251	03	03	
252	00	00	

2253	53	(RCL
2255	43	20	STO
2256	42	71	RCL
2257	43	21	STO
2258	42	72	RCL
2259	42	22	STO
2260	43	73	RCL
2261	22	23	STO
2262	44	74	RCL
2263	73	24	STO
2264	43	75	RCL
2265	23	25	STO
2266	42	76	RCL
2267	42	28	STO
2268	74	77	RCL
2269	43	27	STO
2270	24	78	RCL
2271	42	26	STO
2272	42	79	RCL
2273	75	(SB R
2274	43	15)
2275	25	54	STO
2276	42	69	GTO
2277	76	04	16
2278	43	04	RCL
2279	28	16	67
2280	42	32	X<=>T
2281	77	43	RCL
2282	43	49	INV
2283	27	2ND	X>=T
2284	43	03	56
2285	43	(RCL
2286	29	29	STO
2287	42	71	RCL
2288	71	30	STO
2289	43	72	RCL
2290	30	31	STO
2291	42	73	RCL
2292	72	31	STO
2293	43	73	RCL
2294	33	73	STO
2295	12	73	RCL

322	43	RCL
323	32	32
324	42	STO
325	74	74
326	43	RCL
327	33	33
328	42	STO
329	75	75
330	43	RCL
331	34	34
332	42	STO
333	76	76
334	43	RCL
335	37	37
336	42	STO
337	77	77
338	43	RCL
339	36	36
340	42	STO
341	78	78
342	43	RCL
343	35	35
344	42	STO
345	79	79
346	53	(
347	71	SBR
348	15	E
349	54)
350	54)
351	42	STO
352	69	69
353	61	GTO
354	04	04
355	16	16
356	43	RCL
357	67	67
358	32	X<=>T
359	43	RCL
360	50	50
361	22	INV
362	77	2ND X>=T
363	04	04
364	12	12
365	53	(
366	43	RCL
367	38	38
368	42	STO
369	71	71
370	43	RCL
371	39	39
372	42	STO
373	72	72
374	43	RCL
375	40	40
376	42	STO
377	73	73
378	43	RCL
379	41	41
380	42	STO
381	74	74
382	43	RCL
383	42	42
384	42	STO
385	75	75
386	43	RCL
387	43	43
388	42	STO
389	76	76
390	43	RCL

391	46	46	STO
392	42	77	STO
393	77	43	RCL
394	43	45	45
395	45	42	STO
396	42	78	78
397	78	43	RCL
398	43	44	44
399	44	42	STO
400	42	79	79
401	79	53	(
402	53	11	SBR
403	11	15	E
404	15	54)
405	54	54)
406	54	42	STO
407	42	69	69
408	69	61	GTO
409	61	04	04
410	04	16	16
411	16	43	RCL
412	43	67	67
413	67	2ND	FRT
414	99	2ND	ADV
415	98	2ND	SER
416	92	INV	SER
417	76	2ND	LBL
418	15	E	E
419	00	0	0
420	42	STO	STO
421	70	70	70
422	53	53	53
423	53	53	RCL
424	53	72	72
425	72	2	+/-
426	99	94	+/-
427	94	55	+
428	55	53	(
429	53	72	RCL
430	72	72	72
431	45	55	Y**X
432	00	2	2
433	55	-	-
434	55	4	4
435	04	*	*
436	55	55	RCL
437	43	71	71
438	71	*	*
439	65	*	*
440	53	((
441	43	RCL	RCL
442	73	73	73
443	55	-	-
444	43	RCL	RCL
445	67	67	67
446	55	*	*
447	43	RCL	RCL
448	56	56	56
449	55))
450	44))
451	44	SQRT (X)	SQRT (X)
452	44))
453	44	/	/
454	53	2	2
455	02	*	*
456	55	RCL	RCL
457	71	71	71
458	71		
459			

460	54)
461	54	}
462	42	STO
463	80	80
464	43	RCL
465	80	80
466	32	X<=>T
467	43	RCL
468	51	51
469	77	2ND X>=T
470	04	04
471	82	82
472	43	RCL
473	68	68
474	99	2ND PRT
475	43	RCL
476	67	67
477	99	2ND PRT
478	98	2ND ADV
479	61	GTO
480	05	05
481	65	65
482	53	(
483	43	RCL
484	76	76
485	85	+
486	43	RCL
487	75	75
488	65	*
489	43	RCL
490	80	80
491	85	+
492	43	RCL
493	74	74
494	65	*
495	43	RCL
496	80	80
497	45	Y**X
498	02	2
499	54)
500	99	2ND PRT
501	98	2ND ADV
502	53	(
503	43	RCL
504	66	66
505	85	+
506	43	RCL
507	57	57
508	85	+
509	53	(
510	43	RCL
511	55	55
512	75	-
513	43	RCL
514	68	68
515	54)
516	85	+
517	53	(
518	43	RCL
519	79	79
520	85	+
521	43	RCL
522	78	78
523	65	*
524	43	RCL
525	80	80
526	85	+
527	43	RCL
528	77	77

567	92	INV	SER
566	70	RCL	70
565	43	2ND	ADV
564	98	2ND	PRT
563	54)	
562	54)	
561	00	0	
560	00	0	
559	00	0	
558	00	1	*
557	65	67	RCL
556	65	67	/
555	54)	
554	54)	
553	00	0	
552	00	2	+
551	85	54	RCL
550	85	54	+
549	00	80	RCL
548	00	80	(
547	58	2ND	ADV
546	99	2ND	PRT
545	54)	
544	54)	
543	02	2	Y**X
542	55	80	RCL
541	55	80	*
540	00	80	RCL
539	43	54	54
538	58	2ND	ADV
537	58	2ND	PRT
536	54)	
535	54)	
534	33	2	Y**X
533	33	2	Y**X
532	33	2	Y**X
531	33	2	Y**X
530	00	80	RCL
529	65	65	*

-1 ----- END TI-59 CODE.

\$2

\$3

=====

TI-59 REGISTER TO NAME MAPPING

=====

REG#	BASIC NAME
11	A24
12	A14
13	A04
14	C24
15	C14
16	C04
17	B04
18	B14
19	B24
20	A25
21	A15
22	A05
23	C25
24	C15
25	C05
26	B05
27	B15
28	B25
29	A27
30	A17
31	A07
32	C27
33	C17
34	C07
35	B07
36	B17
37	B27
38	A28
39	A18
40	A08
41	C28
42	C18
43	C08
44	B08
45	B18
46	B28
47	CHG4MAX
48	CHG5MAX
49	CHG7MAX
50	CHG8MAX
51	HACROSS
52	BTRYE
53	BTRYN
54	BTRYA
55	BTRYL
56	RGK
57	DFCOR
58	GRIDE
59	GRIDN
60	GRIDA
61	OT
62	LATDEV
63	RGDEV
64	MILRAD
65	ROTCOR
66	REFDEF
67	TGTRG
68	TGTAZ
69	INVOKE
70	FN
71	FD
72	(FN PARAMETER)
	(FN PARAMETER)

73 (FN PARAMETER)
74 (FN PARAMETER)
75 (FN PARAMETER)
76 (FN PARAMETER)
77 (FN PARAMETER)
78 (FN PARAMETER)
79 (FN PARAMETER)
80 (FN PARAMETER)

EL

\$3

\$4

=====

INPUT DATA TO READ MAPPING

=====

DATA	REG	NAME
- .0133670	11	A24
21.2691	12	A14
-105.7	13	A04
- .00001499	14	C24
.06630	15	C14
-.41	16	C04
.77	17	B04
.01314	18	B14
.00001720	19	B24
- .0149331	20	A25
24.3439	21	A15
64.7	22	A05
- .00001420	23	C25
.07069	24	C15
.06	25	C05
1.26	26	B05
.01508	27	B15
.00001678	28	B25
- .0173835	29	A27
29.8741	30	A17
2255.2	31	A07
- .00001668	32	C27
.08487	33	C17
3.29	34	C07
1.3	35	B07
.02713	36	B17
.00001306	37	B27
- .0182137	38	A28
32.3731	39	A18
4107.4	40	A08
- .00001668	41	C28
.09272	42	C18
5.74	43	C08
1.36	44	B08
.02891	45	B18
.00001410	46	B28
5700	47	CHG4 MAX
7000	48	CHG5 MAX
10800	49	CHG7 MAX
17600	50	CHG8 MAX
715	51	HACROSS
0	52	BTRYE
0	53	BTRYN
0	54	BTRYA
800	55	BTRYL
1.0	56	RGK
0	57	DFCOR

4000
4000
10

400C
-400
10

1018.5924
1600
3200

\$4

58 GRID E
59 GRID N
60 GRID A

61 OT
62 LAT DEV
63 RG DEV

64 MIL RAD
65 ROT COR
66 REF DEF

APPENDIX P

TEST PROGRAM LINKER OUTPUT

=====

EAX59 PROGRAM INSTRUCTIONS: VERSION 1.0

=====

* CONGRATULATIONS, YOU HAVE JUST COMPILED A BASIC PROGRAM INTO A TI-59 PROGRAM. IN SO DOING IT IS VERY POSSIBLE THAT YOUR PROGRAM IS LARGER THAN THE MEMORY OF THE CALCULATOR. IF THIS IS THE CASE THEN THE PROGRAM HAS BEEN SEGMENTED AND PROMPTING CODE INSERTED TO GUIDE YOUR CALCULATOR PROGRAM DURING ITS EXECUTION. THE REMAINDER OF THIS OUTPUT CONSISTS OF TI-59 CODE LISTINGS AND OTHER INFORMATION TO AID YOU IN YOUR PROGRAM EXECUTION.

* THE FOLLOWING DEFINITIONS ARE PROVIDED AS AN AID TO READING THE PROGRAM LISTING FILE.

* DEFINITIONS:

* MODULE: A MODULE IS DEFINED TO BE ALL THE MEMORY DEDICATED TO PROGRAM STEPS. THE SIZE IS VARIABLE AND IS DEPENDENT ON THE REGISTER REQUIREMENT. VALUES RANGE FROM 0 TO 239, 479 OR 719 DEPENDING ON THE AMOUNT OF REGISTERS USED BY THE PROGRAM.

* CARD: A CARD IS DEFINED TO BE ONE MAGNETIC CARD. A CARD HOLDS 480 PROGRAM STEPS. THESE STEPS ARE NOT CONTIGUOUS BUT ARE ARRANGED ON THE TWO SIDES OF THE CARD.

* SIDE: A SIDE IS ONE HALF OF A CARD. IT CONTAINS UP TO 240 STEPS. WHEN ONE SIDE OF A CARD IS READ BY THE CALCULATOR 240 PROGRAM STEPS ARE FILLED IN MEMORY. THESE BLOCKS OF 240 STEPS ARE REFERED TO AS "BANKS" IN THE MANUFACTURER LITERATURE. WHEN LOADING A CARD YOU WILL LOAD ONLY BANK NUMBER 1 AND/OR 2 FOR PROGRAM STEPS.

* PARTITION: THIS IS DEFINED TO BE THE CURRENT SETTING OF CALCULATOR MEMORY AS APPLIED TO THE AMOUNT OF MEMORY DEDICATED TO STORAGE REGISTERS AND THE AMOUNT DEDICATED TO PROGRAM STEPS. WE WILL BE DEALING WITH 3 PARTITIONS. THESE ARE:

3	719.29
4	479.59
5	239.69

FORMAT: X YYY.ZZ

WHERE X STANDS FOR PARTITION NUMBER
YYY STANDS FOR PROGRAM STEPS (0-YYY)
ZZ STANDS FOR REGISTERS (0-ZZ).

=====

TI 59 PROGRAM LISTING BY MODULE/CARD/SIDE:

=====

- * THE FOLLOWING IS YOUR PROGRAM LISTING. THE PROGRAM IS LISTED ACCORDING TO MODULE NUMBER AND ITS ASSOCIATED CARDS AND CARD SIDES.
- * REFER TO THE TI-59 PROGRAMMER'S GUIDE ON HOW TO INPUT A PROGRAM AND WRITE IT TO MAGNETIC CARDS.
- * CAUTION: ENSURE THAT THE CORRECT CALCULATOR PARTITION IS SET BEFORE INPUTTING A PROGRAM AND WRITING TO MAGNETIC CARDS.
- * CAUTION: ENSURE THAT YOU DO NOT CONFUSE BANK NUMBERS WITH CARD/MODULE OR SIDE NUMBERS. THE NUMBERS WHICH REFER TO THE LISTING ARE AKIN TO A VIRTUAL ADDRESS AND DO NOT REPRESENT THE ACTUAL BANK NUMBER. IF IN DOUBT, REMEMBER TO USE THE TABLE BELOW TO TRANSLATE VIRTUAL TO ACTUAL BANK NUMBERS.

VIRTUAL BANK	ACTUAL BANK
MODULE #	
CARD1	
SIDE1 -----	BANK1
MODULE #	
CARD1	
SIDE2 -----	BANK2
MODULE #	
CARD2	
SIDE1 -----	BANK3

=====

TI-59 LISTING

=====

- * MANUAL RETURN REGISTER TOP IS 81

STORE IN REGISTER: 8
- * PROGRAM PARTITION IS 239.89
- * PARTITION NUMBER IS 9

*MODULE # 1

CARD #1

SIDE #1

000	76	2ND LBL
001	11	A
002	71	SBR
003	12	B
004	68	2ND NOP
005	71	SBR
006	13	C
007	68	2ND NOP
008	71	SBR
009	14	D
010	68	2ND NOP
011	24	CE
012	08	8
013	08	8

0014	0918
0015	0917
0016	0916
0017	0915
0018	0914
0019	0913
0020	0912
0021	0911
0022	0910
0023	0909
0024	0908
0025	0907
0026	0906
0027	0905
0028	0904
0029	0903
0030	0902
0031	0901
0032	0900
0033	0899
0034	0898
0035	0897
0036	0896
0037	0895
0038	0894
0039	0893
0040	0892
0041	0891
0042	0890
0043	0889
0044	0888
0045	0887
0046	0886
0047	0885
0048	0884
0049	0883
0050	0882
0051	0881
0052	0880
0053	0879
0054	0878
0055	0877
0056	0876
0057	0875
0058	0874
0059	0873
0060	0872
0061	0871
0062	0870
0063	0869
0064	0868
0065	0867
0066	0866
0067	0865
0068	0864
0069	0863
0070	0862
0071	0861
0072	0860
0073	0859
0074	0858
0075	0857
0076	0856
0077	0855
0078	0854
0079	0853
0080	0852
0081	0851
0082	0850

8	
R/S	
2ND	LBL
STO	
00	
2ND	OP
28	
1	
STO	2ND IND
08	
2	
0	
0	
0	
R/S	
2ND	LBL
B	
(
CL	
59	
+	
(
RCL	
63	
X	
)	
RCL	
65	
-	
RCL	
61	
)	
/	
RCL	
64	
)	
2ND	SIN
-	
RCL	
62	
X	
(
RCL	
65	
-	
RCL	
61	
)	
/	
RCL	
64	
)	
2ND	COS
)	
STO	
59	
(
RCL	
58	
+	
(
RCL	
63	
X	

152 43
 153 53
 154 54
 155 55
 156 43
 157 67
 158 54
 159 27
 160 38
 161 65
 162 43
 163 44
 164 54
 165 42
 166 88
 167 43
 168 58
 169 32
 170 33
 171 52
 172 32
 173 22
 174 77
 175 01
 176 89
 177 53
 178 43
 179 65
 180 75
 181 43
 182 68
 183 54
 184 42
 185 68
 186 61
 187 02
 188 00
 189 53
 190 03
 191 65
 192 43
 193 55
 194 85
 195 43
 196 68
 197 54
 198 42
 199 68
 200 92

RCL
 53
)
 RCL
 67
)
 2ND INV
 2ND SIN
 X
 RCL
 64
)
 STO
 68
 RCL
 58
 X<=>T
 RCL
 52
 X<=>T
 INV
 2ND X>=I
 01
 89
 (
 RCL
 65
 -
 RCL
 68
)
 STO
 68
 GTO
 02
 00
 (
 3
 X
 RCL
 65
 +
 RCL
 68
)
 STO
 68
 INV SBR

*MODULE # 2
CARD # 1
SIDE # 1

000	76	2ND LBL
001	14	D
002	43	RCL
003	67	67
004	32	X<=>T
005	43	RCL
006	47	47
007	22	INV
008	77	2ND X>=T
009	00	00
010	58	58
011	53	(
012	43	RCL
013	11	11
014	42	STO
015	71	71
016	43	RCL
017	12	12
018	42	STO
019	72	72
020	43	RCL
021	13	13
022	42	STO
023	73	73
024	43	RCL
025	14	14
026	42	STO
027	74	74
028	43	RCL
029	15	15
030	42	STO
031	75	75
032	43	RCL
033	16	16
034	42	STO
035	76	76
036	43	RCL
037	19	19
038	42	STO
039	77	77
040	43	RCL
041	18	18
042	42	STO
043	78	78
044	43	RCL
045	17	17
046	42	STO
047	79	79
048	53	(
049	71	SBR
050	15	E
051	54)
052	54)
053	42	STO
054	69	69
055	61	GTO
056	01	01
057	23	23
058	43	RCL
059	67	67
060	32	Y<=>T
061	43	RCL
062	48	48
063	22	INV
064	77	2ND X>=T

AD-A132 172

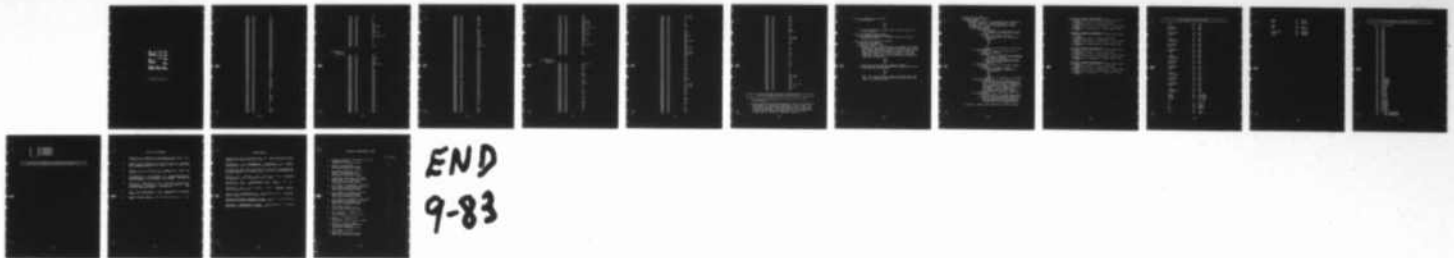
DDSIGN AND IMPLEMENTATION OF A BASIC CROSS-COMPILER AND
VIRTUAL MEMORY MA. (U) NAVAL POSTGRADUATE SCHOOL
MONTEREY CA M R KINDL ET AL. JUN 83

4/4

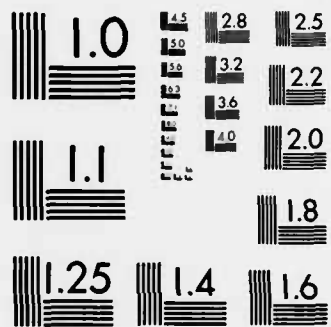
UNCLASSIFIED

F/G 9/2

NL



END
9-83



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

065
 066
 067
 068
 069
 070
 071
 072
 073
 074
 075
 076
 077
 078
 079
 080
 081
 082
 083
 084
 085
 086
 087
 088
 089
 090
 091
 092
 093
 094
 095
 096
 097
 098
 099
 100
 101
 102
 103
 104
 105
 106
 107
 108
 109
 110
 111
 112
 113
 114
 115
 116
 117
 118
 119
 120
 121
 122
 123
 124
 125
 126
 127
 128
 129
 130
 131
 132
 133

532
 533
 200
 422
 711
 43
 21
 42
 72
 43
 22
 42
 22
 42
 73
 43
 23
 42
 74
 43
 24
 42
 75
 43
 25
 42
 76
 43
 28
 42
 77
 43
 27
 42
 78
 43
 26
 42
 79
 53
 71
 54
 54
 42
 69
 51
 23
 42
 00
 24
 03
 93
 00
 00
 91
 42
 00
 24
 03
 00
 01
 91
 42
 00

01
 32
 (RCL
 20
 STO
 71
 RCL
 21
 STO
 72
 RCL
 22
 STO
 73
 RCL
 23
 STO
 74
 RCL
 24
 STO
 75
 RCL
 25
 STO
 76
 RCL
 28
 STO
 77
 RCL
 27
 STO
 78
 RCL
 26
 STO
 79
 (SBR
 E)
 STO
 69
 STO
 01
 23
 STO
 00
 CE
 S
 00
 00
 00
 R/S
 STO
 00
 CE
 S
 3
 1
 R/S
 STO
 00

134 24
 135 03
 136 93
 137 00
 138 00
 139 00
 140 91
 141 76
 142 15
 143 42
 144 00
 145 69
 146 28
 147 02
 148 72
 149 08
 150 04
 151 93
 152 00
 153 00
 154 00
 155 91

CE
 3
 0
 0
 0
 0
 R/S
 2ND LBL
 E
 STO
 0
 2ND OP
 28
 2
 STO 2ND IND
 08
 4
 0
 0
 0
 R/S

*MODULE # 3
 CARD #1
 SIDE #1

000 43
 001 67
 002 32
 003 43
 004 49
 005 22
 006 77
 007 00
 008 56
 009 53
 010 43
 011 29
 012 42
 013 71
 014 43
 015 30
 016 42
 017 72
 018 43
 019 31
 020 42
 021 73
 022 43
 023 32
 024 42
 025 74
 026 43
 027 33
 028 42
 029 75
 030 43
 031 34
 032 42
 033 76
 034 43
 035 37
 036 42
 037 77
 038 43
 039 36
 040 42

RCL
 67
 X<=>T
 RCL
 49
 INV
 2ND X>=T
 00
 56
 1
 RCL
 29
 STO
 71
 RCL
 30
 STO
 72
 RCL
 31
 STO
 73
 RCL
 32
 STO
 74
 RCL
 33
 STO
 75
 RCL
 34
 STO
 76
 RCL
 37
 STO
 77
 RCL
 36
 STO

04 1
 04 2
 04 3
 04 4
 04 5
 04 6
 04 7
 04 8
 04 9
 05 0
 05 1
 05 2
 05 3
 05 4
 05 5
 05 6
 05 7
 05 8
 05 9
 06 0
 06 1
 06 2
 06 3
 06 4
 06 5
 06 6
 06 7
 06 8
 06 9
 07 0
 07 1
 07 2
 07 3
 07 4
 07 5
 07 6
 07 7
 07 8
 07 9
 08 0
 08 1
 08 2
 08 3
 08 4
 08 5
 08 6
 08 7
 08 8
 08 9
 09 0
 09 1
 09 2
 09 3
 09 4
 09 5
 09 6
 09 7
 09 8
 09 9
 10 0
 10 1
 10 2
 10 3
 10 4
 10 5
 10 6
 10 7
 10 8
 10 9

78
 43
 35
 42
 29
 79
 53
 71
 15
 54
 54
 42
 69
 61
 01
 16
 43
 67
 32
 43
 50
 22
 77
 01
 12
 53
 43
 38
 42
 71
 43
 39
 42
 72
 43
 40
 42
 73
 43
 41
 42
 74
 43
 42
 42
 75
 43
 43
 42
 76
 43
 46
 42
 77
 43
 45
 42
 78
 43
 44
 42
 79
 53
 71
 15
 54
 54
 42
 69
 61

78
 RCL
 STO
 STO
)
 SBR
)
 STO
 STO
 01
 16
 RCL
 67
 X<=>T
 STO
 INV
 22
 ND X>=T
 01
 12
)
 RCL
 38
 STO
 71
 RCL
 39
 STO
 72
 RCL
 40
 STO
 73
 RCL
 41
 STO
 74
 RCL
 42
 STO
 75
 RCL
 43
 STO
 76
 RCL
 46
 STO
 77
 RCL
 45
 STO
 78
 RCL
 44
 STO
 79
)
 SBR
)
 STO
 STO
 61

110	01	01
111	16	16
112	43	RCL
113	67	67
114	99	2ND PRT
115	98	2ND ADV
116	42	STO
117	00	00
118	73	RCL 2ND IND
119	08	08
120	69	2ND OP
121	38	38
122	91	R/S
123	76	2ND LBL
124	15	FE
125	42	STO
126	00	00
127	69	2ND OP
128	28	28
129	03	3
130	72	STO 2ND IND
131	08	08
132	04	4
133	93	.
134	00	0
135	00	0
136	00	0
137	91	R/S

*MODULE # 4
 CARD #1
 SIDE #1

000	76	2ND LBL
001	15	FE
002	00	0
003	42	STO
004	70	70
005	53	{
006	53	{
007	43	RCL
008	72	72
009	94	+/-
010	85	+
011	53	{
012	43	RCL
013	72	72
014	45	Y**X
015	02	2
016	75	-
017	53	{
018	04	4
019	65	X
020	43	RCL
021	71	71
022	65	X
023	53	{
024	43	RCL
025	73	73
026	75	-
027	43	RCL
028	67	67
029	65	X
030	43	RCL
031	56	56
032	54	}
033	54	}
034	54	}

035
 036
 037
 038
 039
 040
 041
 042
 043
 044
 045
 046
 047
 048
 049
 050
 051
 052
 053
 054
 055
 056
 057
 058
 059
 060
 061
 062
 063
 064
 065
 066
 067
 068
 069
 070
 071
 072
 073
 074
 075
 076
 077
 078
 079
 080
 081
 082
 083
 084
 085
 086
 087
 088
 089
 090
 091
 092
 093
 094
 095
 096
 097
 098
 099
 100
 101
 102
 103

534
 554
 553
 553
 653
 711
 544
 422
 800
 433
 800
 323
 511
 777
 000
 655
 433
 688
 999
 433
 677
 999
 888
 611
 001
 488
 533
 433
 766
 855
 433
 755
 655
 433
 800
 855
 433
 744
 655
 433
 800
 455
 022
 544
 999
 988
 533
 433
 666
 855
 433
 577
 885
 533
 442
 555
 755
 433
 688
 554
 855
 533
 433
 799
 85

S Q R T (X)
)
 /
 {
 2
 X
 R C L
 7 1
)
 S T O
 8 0 0
 R C L
 8 0 0
 X < = > T
 R C L
 5 1 1
 2 N D X > = T
 0 0 0
 6 5 5
 R C L
 6 8 8
 2 N D P R T
 R C L
 6 7 7
 2 N D P R T
 2 N D A D V
 G T O
 0 1 1
 4 8 8
 /
 R C L
 7 6 6
 +
 R C L
 7 5 5
 X
 R C L
 8 0 0
 +
 R C L
 7 4 4
 X
 R C L
 8 0 0
 Y * X
 2
)
 2 N D P R T
 2 N D A D V
 (R C L
 6 6 6
 +
 R C L
 5 7 7
 +
 (R C L
 5 5 5
 -
 R C L
 6 8 8
)
 +
 (R C L
 7 9 9
 +

104	4	RCL
105	78	78
106	65	X
107	43	RCL
108	80	80
109	85	+
110	43	RCL
111	77	77
112	65	X
113	43	RCL
114	80	80
115	45	Y**X
116	00	2
117	54)
118	99)
119	99	2ND PRT
120	99	2ND ADV
121	53	{
122	53	RCL
123	80	80
124	85	+
125	53	{
126	53	RCL
127	43	60
128	60	-
129	75	RCL
130	43	54
131	54	+
132	85	2
133	00	0
134	00)
135	54)
136	55	/
137	43	RCL
138	67	67
139	65	X
140	00	1
141	00	0
142	00	0
143	00	0
144	54)
145	54)
146	99	2ND PRT
147	98	2ND ADV
148	43	RCL
149	70	70
150	42	STO
151	00	00
152	73	RCL 2ND IND
153	08	08
154	69	2ND OP
155	38	38
156	91	R/S

=====

TI-59 PROGRAM SPECIFIC INSTRUCTIONS:

=====

- * THE FOLLOWING INFORMATION WILL TELL YOU HOW TO RUN YOUR PROGRAM.
- * YOU MUST ENTER YOUR PROGRAM MANUALLY INTO THE CALCULATOR AND WRITE THE PROGRAM TO MAGNETIC CARDS. THIS STEP ONLY NEEDS TO BE ACCOMPLISHED ONCE. AFTER THAT, THE PROGRAM IS ENTERED USING THE MAGNETIC CARD FACILITY OF THE CALCULATOR. SEE THE MANUFACTURER'S LITERATURE ON ENTERING A PROGRAM AND WRITING IT TO MAGNETIC CARDS. YOU WILL NEED TO PARTITION MEMORY.

- * HOW TO PARTITION THE MEMORY
- * KEY SEQUENCE:

X
2ND
OP
17

* X IS THE PARTITION NUMBER GIVEN IN THE LISTING OF YOUR PROGRAM.

- * WHEN TO PARTITION THE MEMORY
 - * ONCE BEFORE FEADING IN CARDS.
 - * ONCE BEFORE MANUALLY ENTERING PROGRAM IN ORDER TO WRITE TO CARDS.

- * HOW TO START AND RUN YOUR PROGRAM

- * TURN ON CALCULATOR
- * PARTITION CALCULATOR
- * LOAD ALL MODULE 1 CARDS
- * OPTIONAL STEP: IF YOU SELECTED THE MANUAL DATA INPUT DENOTED IN YOUR BASIC PROGRAM BY USING THE "DATA" AND "READ" STATEMENTS THEN YOU MUST MANUALLY ENTER YOUR DATA INTO THE CALCULATOR MEMORY. THIS IS DONE BY REFERRING TO "INPUT DATA TO READ" TABLE PROVIDED AT THE END OF THIS LISTING. MANUALLY ENTER THE GIVEN DATA INTO THE REGISTERS USING THE FOLLOWING KEYSTROKES:

DATA
STO
XX

- * WHERE XX IS THE DESIRED REGISTER NUMBER.
- * INITIALIZE THE MANUAL SBR RETURN CONTROL STACK WITH THE FOLLOWING KEYSTROKES:

XX
STO
08

WHERE XX IS THE MANUAL RETURN REGISTER STACK TOP.
(THIS IS GIVEN WITH THE PROGRAM LISTING NEAR THE PARTITION INFORMATION.)

```

* PRESS "A" TO START.
* FOLLOW DISPLAY PROMPTS.
* DEFINITIONS:
  * RUN-TIME PROMPTS: ARE DEFINED TO BE CALCULATOR
    PROMPTS DISPLAYED IN THE CALCULATOR WINDOW
    IN THE FORM OF A 4 DIGIT DECIMAL, 2 DIGIT
    INTEGER OR A 1 DIGIT INTEGER. EACH PROMPT
    IS OUTLINED BELOW:
    * 4 DIGIT DECIMAL
      * FORMAT: X.YYY
      * X STANDS FOR MODULE NUMBER (1-9)
      * YYY STANDS FOR STARTING ADDRESS
    * ACTIONS:
      * LOAD ALL MODULE X CARDS.
      * PRESS FOLLOWING KEY SEQUENCE TO
        INITIALIZE:
          RCL
          00
          GTO
          Y
          Y
          Y
      * PRESS R/S TO CONTINUE IN NEW MOD.
    * 2 DIGIT INTEGER
      * FORMAT: XX WHERE XX STANDS FOR A
        REGISTER NUMBER.
    * ACTIONS:
      * LOOK UP IN REGISTER MAP PROVIDED
        THE BASIC NAME THAT CORRESPONDS
        TO THE XX NUMBER.
      * ENTER THE BASIC VARIABLE VALUE.
      * PRESS R/S TO CONTINUE WITH THE
        ENTERED VALUE.
    * 1 DIGIT INTEGER
      * FORMAT: X WHERE X IS A MODULE NUMBER.
    * ACTIONS:
      * LOAD ALL MODULE X CARDS.
      * PRESS FOLLOWING SEQUENCE TO
        INITIALIZE:
          RCL
          00
          INV
          SBR
      * PRESS R/S TO CONTINUE IN NEW MOD.
    * PAUSE IN DISPLAY
      * AN UNFORMATTED DIGIT FLASHES IN THE
        DISPLAY BEFORE BEING DISPLAYED.
        THIS IS AN ANSWER THAT CORRESPONDS
        TO A REQUESTED ANSWER IN THE BASIC
        PROGRAM USING THE BASIC PRINT
        STATEMENT. THESE ANSWERS OCCUR IN
        THE SAME ORDER AS THEY WERE
        REQUESTED IN THE BASIC PROGRAM.
      * ACTIONS: NOTE ANSWER AND PRESS R/S.
    * 888 IN DISPLAY
      * SPECIFIC PROMPT THAT INDICATES THAT
        THE PROGRAM HAS STOPPED EXECUTION.
      * ACTIONS: IF DESIRED FIND ANSWERS IN
        THE CALCULATOR MEMORY USING THE
        "TI-59 REGISTER TO NAME MAPPING"
        AT THE END OF THE INSTRUCTIONS.

```

* EXPECTED CONTROL FLOW PROMPTS BY MODULE FOLLOW:

```

* EXPECTED PROMPTS FOR MODULE # 1
* FORWARD JUMP CONTINUATION: 4 DIGIT REAL CCDE.
* NONE
* SUBROUTINE INVOKE: 4 DIGIT REAL CODE.
* 2.000
* MANUAL RETURN FROM A SUBROUTINE: 1 DIGIT CODE.
* NONE
* SEQUENTIAL CONTINUATION: 4 DIGIT REAL CODE.
* NONE

* EXPECTED PROMPTS FOR MODULE # 2
* FORWARD JUMP CONTINUATION: 4 DIGIT REAL CCDE.
* 3.116
* 3.000
* SUBROUTINE INVOKE: 4 DIGIT REAL CODE.
* 4.000
* MANUAL RETURN FROM A SUBROUTINE: 1 DIGIT CODE.
* NONE
* SEQUENTIAL CONTINUATION: 4 DIGIT REAL CODE.
* 3.000

* EXPECTED PROMPTS FOR MODULE # 3
* FORWARD JUMP CONTINUATION: 4 DIGIT REAL CCDE.
* NONE
* SUBROUTINE INVOKE: 4 DIGIT REAL CODE.
* 4.000
* MANUAL RETURN FROM A SUBROUTINE: 1 DIGIT CODE.
* YES
* SEQUENTIAL CONTINUATION: 4 DIGIT REAL CODE.
* NONE

* EXPECTED PROMPTS FOR MODULE # 4
* FORWARD JUMP CONTINUATION: 4 DIGIT REAL CODE.
* NONE
* SUBROUTINE INVOKE: 4 DIGIT REAL CODE.
* NONE
* MANUAL RETURN FROM A SUBROUTINE: 1 DIGIT CODE.
* YES
* SEQUENTIAL CONTINUATION: 4 DIGIT REAL CODE.
* NONE

```

=====

INPUT DATA TO READ MAPPING

=====

DATA	REG	NAME
-.0133670	11	A24
21.2691	12	A14
-105.7	13	A04
-.00001499	14	C24
.06630	15	C14
-.41	16	C04
.77	17	B04
.01314	18	B14
.00001720	19	B24
-.0149331	20	A25
24.3439	21	A15
64.7	22	A05
-.00001420	23	C25
.07069	24	C15
.06	25	C05
1.26	26	B05
.01508	27	B15
.00001678	28	B25
-.0173835	29	A27
29.8741	30	A17
2255.2	31	A07
-.00001668	32	C27
.08487	33	C17
3.29	34	C07
1.3	35	B07
.02713	36	B17
.00001306	37	B27
-.0182137	38	A28
32.3731	39	A18
4107.4	40	A08
-.00001668	41	C28
.09272	42	C18
5.74	43	C08
1.36	44	B08
.02891	45	B18
.00001410	46	B28
5700	47	CHG4 MAX
7000	48	CHG5 MAX
10800	49	CHG7 MAX
17600	50	CHG8 MAX
715	51	HACROSS
0	52	BTRYE
0	53	BTRYN
0	54	BTRYA
800	55	BTRYL
1.0	56	RGK
0	57	DFCOR

4000
4000
10

4000
-400
10

1018.5924
1600
3200

58 GRIDE
59 GRIDN
60 GRIDA

61 OT
62 LATDEV
63 RGDEV

64 MILRAD
65 ROTCOR
66 REFDEF

=====

 TI-59 REGISTER TO NAME MAPPING

 =====

REG #	BASIC NAME
11	A24
12	A14
13	A04
14	C24
15	C14
16	C04
17	B04
18	B14
19	B24
20	A25
21	A15
22	A05
23	C25
24	C15
25	C05
26	B05
27	B15
28	B25
29	A27
30	A17
31	A07
32	C27
33	C17
34	C07
35	B07
36	B17
37	B27
38	A28
39	A18
40	A08
41	C28
42	C18
43	C08
44	B08
45	B18
46	B28
47	CHG4MAX
48	CHG5MAX
49	CHG7MAX
50	CHG8MAX
51	HACROSS
52	BTRYE
53	BTRYN
54	BTRYA
55	BTRYL
56	RGK
57	DFCOR
58	GRIDE
59	GRIDN
60	GRIDA
61	OT
62	LATDEV
63	RGDEV
64	MILRAD
65	ROTCOR
66	REFDEF
67	TGTRG
68	TGTAZ
69	INVOKE
70	FN FD
71	{ FN PARAMETER }
72	{ FN PARAMETER }
73	{ FN PARAMETER }

74 (FN PARAMETER)
75 (FN PARAMETER)
76 (FN PARAMETER)
77 (FN PARAMETER)
78 (FN PARAMETER)
79 (FN PARAMETER)
80 EL

=====
END BAX59 SEGMENTATION/INSTRUCTION: VERSION 1.0
=====

LIST OF REFERENCES

1. Hamming, R.W., The Art of Programming the TI-59, Naval Postgraduate School, pp. 6-7, April 1980.
2. Graham, J.W., McPhee, K.I., and Welch, J.W., Waterloo Basic Primer and Reference Manual, WATPAC Publications Ltd., October 1980.
3. Jensen, K. and Wirth, N., Pascal User Manual and Report, 2d ed., Springer-Verlag, 1978.
4. Constantine, L. and Yourdon, E., Structured Design: Fundamentals of a Discipline of Computer Program and System Design, Prentice-Hall, Inc., 1979.
5. Basili, V. and Turner, A.J., "Iterative Enhancement: A Practical Technique for Software Development," IEEE Transactions on Software Engineering, v. 1, pp. 390-396, December 1975.
6. Aho, A.V. and Ullman, J.D., Principles of Compiler Design, Addison-Wesley, p. 340, 1977.
7. Master Library Manual, Texas Instruments Inc., p. 54, 1979.

BIBLIOGRAPHY

Augenstein, M.J., and Tenenbaum, A.M., Data Structures Using Pascal, Prentice-Hall, Inc., 1981.

Calingaert, P., Assemblers, Compilers, and Program Translation, Computer Science Press, Inc., 1979.

Computer Systems Group, University of Waterloo, Introduction to Waterloo Basic for VM/370 CMS, University of Waterloo, 31 October 1980.

Farish, R.F., O'Grady, C.D., and Oliva, R.A., Personal Programming, Texas Instruments Inc., 1979.

Gottfried, B.S., Programming with Basic, 2d ed., McGraw-Hill, Inc., 1982.

Kernighan, B.W. and Plauger, P.J., Software Tools, Addison-Wesley, 1976.

Lewis, P.M., Rosenkrantz, D.J., and Stearns, R.E., Compiler Design Theory, Addison-Wesley, 1976.

PASCAL/VS Language Reference Manual, 2d ed., International Business Machines Corporation, April 1981.

PASCAL/VS Programmer's Guide, International Business Machines Corporation, 31 December 1981.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2. Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3. Curricular Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93940	1
4. Department Chairman, Code 52HQ Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5. Dr. Bruce J. MacLennan, Code 52ML Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
6. Dr. Gordon E. Latta, Code 53LZ Department of Mathematics Naval Postgraduate School Monterey, California 93940	1
7. Dr. Richard W. Hamming, Code 52HG Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
8. CPT Mark R. Kindl 413 E. Washington Street Villa Park, Illinois 60181	2
9. CPT James H.W. Inskeep, Jr P.O. Box 444 Locust Grove, Virginia 22101	2
10. CAPT I. F. Rogers, USN Box 327 Lumberport, West Virginia 26386	1
11. CAPT J. C. White, USMC Computer Science Department U.S. Naval Academy Annapolis, Maryland 21401	1
12. CPT Thomas A. Gandy SMC 1597 Naval Postgraduate School Monterey, California 93940	1