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

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

Mark R. Kindl
James H. W. Inskeep, Jr.

Naval Postgraduate School
Monterey, California

Naval Postgraduate School
Monterey, California

Approved for public release; distribution unlimited

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 (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.
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
E.S., United States Military Academy, 1974

and

James H. W. Inskeep, Jr.
Captain, United States Army
E.S., United States Military Academy, 1974

Submitted in partial fulfillment of the requirements for the degree of

MASTEE OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1983

Authors: Mark R. Kindl

James H. W. Inskeep, Jr.

Approved by: James P. Lawrence
Thesis Advisor

C. F. Latto
Second Reader

Chairman, Department of Computer Science

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-COMPILED ....................................... 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
   B. 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
APPENDIX N: TEST PROGRAM DATA/READ MAPPING FILE (READ?) 267
APPENDIX C: TEST PROGRAM LINK INTERFACE FILE (SCRATCH) 265
APPENDIX P: TEST PROGRAM LINKER OUTPUT .............. 232
LIST OF REFERENCES ........................................ 301
BIBLICGRAPHY .................................................. 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. BAX59 OPTION Statement Parameters ............... 58
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Cross-Compiler Contour (PROGRAM BAX59)</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>Reserved Word Table Arrays</td>
<td>34</td>
</tr>
<tr>
<td>4.3</td>
<td>Symbol Table</td>
<td>40</td>
</tr>
<tr>
<td>4.4</td>
<td>Code Data Structure</td>
<td>44</td>
</tr>
<tr>
<td>4.5</td>
<td>Linker Contour</td>
<td>59</td>
</tr>
<tr>
<td>4.6</td>
<td>System Utilities Contour</td>
<td>63</td>
</tr>
<tr>
<td>4.7</td>
<td>ELF_SEGTBL Contour</td>
<td>65</td>
</tr>
<tr>
<td>4.8</td>
<td>TI-59 Code</td>
<td>68</td>
</tr>
<tr>
<td>4.9</td>
<td>Sequential Segment Table</td>
<td>70</td>
</tr>
<tr>
<td>4.10</td>
<td>COALESCE Contour</td>
<td>72</td>
</tr>
<tr>
<td>4.11</td>
<td>Coalesced Segment Table</td>
<td>76</td>
</tr>
<tr>
<td>4.12</td>
<td>INSTRUCTIONS Contour</td>
<td>80</td>
</tr>
</tbody>
</table>
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 mainframe 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 straightforward 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 EASIC 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 EASIC Version 2.0 (W BASIC) [Ref. 2], was chosen as the specific source language to be implemented by this compiler. The power of the TI-59 compared to the W BASIC language places restrictions upon the set of W BASIC commands which can be translated. The specific W BASIC 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 (EAsic 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 alphanumericics 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 WBASIC 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 imple-
mented 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 lead 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 WBASEIC 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 WBASEIC 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 WBASEIC source program using a WBASEIC 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 BASIC 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-COMPIILER

The fundamental design of BAX59 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
WBIASIC 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 BAX59 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 RWTELF. 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 subse-
quently 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 EAX59. 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
cutside 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 TI-59 keycode text (C72XTF) 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 vari-
able 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 W BASIC "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 ("\n") 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 RWTELF 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 RWWORD 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 RWWORD array for that length index. The indexes of the RWWORD 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 WESBASIC 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 PGOTC 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 (PEERCR) 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 IDLOCKUP 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
### SLOT RECORD ENTRIES

<table>
<thead>
<tr>
<th>ARRAY OF POINTER BUCKETS</th>
<th>25</th>
<th>26</th>
<th>27</th>
<th>28</th>
<th>29</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VARID</strong> TOTAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>AUXREG1 AUXREG2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>VARID</strong> COUNT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>VARID</strong> PI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-314</td>
<td>AUXREG1 AUXREG2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>FNID</strong> FN_USER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>VARID</strong> ALPHA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>AUXREG1 AUXREG2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>VARID</strong> BETA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>AUXREG1 AUXREG2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ALPHA and BETA are formal parameters of FN_USER.

### KEY:

**BASE SLOT RECORD:**

<table>
<thead>
<tr>
<th>(VAR_ID) (TAG)</th>
<th>TYP</th>
<th>ID</th>
<th>IDENT</th>
<th>SLOT</th>
<th>VARIANT FIELDS</th>
<th>VARIANT TYPES:</th>
<th>FNID</th>
<th>FN1</th>
<th>FN2</th>
<th>FN3</th>
<th>FN4</th>
<th>FN5</th>
<th>FN6</th>
<th>FN7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VARIANT TYPES:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>VARID</strong></td>
<td>REGNO</td>
<td>AUXREG1</td>
<td>AUXREG2</td>
<td>FNID</td>
<td>FNREGNO</td>
<td>LSL</td>
<td>FNPLL</td>
<td>FNPLINK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTE 1:** VALUES OF "LONG" FUNCTION INTEGER ARRAY FN(1..15) NOT SHOWN.

**NOTE 2:** EMPTY FIELDS ARE CONSIDERED NIL POINTERS.

**NOTE 3:** IDENTIFIERS IN THIS DIAGRAM ARE NOT NECESSARILY HASHED CORRECTLY.

**NOTE 4:** FNPL is a special constant entered during initialization.

---

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 LABELP 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>
<thead>
<tr>
<th>TABLE I</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Production Rules for Expressions</strong></td>
</tr>
<tr>
<td><code>&lt;EXPRESSION&gt;</code> ::= <code>&lt;PRIMARY&gt;</code> { <code>&lt;BINARYOP&gt;</code> <code>&lt;PRIMARY&gt;</code> }</td>
</tr>
<tr>
<td><code>&lt;PRIMARY&gt;</code> ::= <code>[+-] </code>&lt;PRIMARY&gt;`</td>
</tr>
<tr>
<td><code>&lt;UNSIGNED NUMBER&gt;</code></td>
</tr>
<tr>
<td><code>&lt;IDENTIFIER&gt;</code></td>
</tr>
<tr>
<td>( <code>&lt;EXPRESSION&gt;</code> )</td>
</tr>
</tbody>
</table>

Table I lists the grammatical specification for a WEASIC 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: W BASIC line number nodes and TI-59 keycode nodes. Since unstructured constructs in W BASIC 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 W BASIC 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, BEGINC, ENDCP, LPLEAD, LPCRAIL, 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 SETJMP2EXT 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 GCSUB 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 GCSUB 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 FINDGOSUBLBL. 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>
<thead>
<tr>
<th>A = B</th>
<th>A &lt;&gt; B</th>
<th>A &gt;= B</th>
<th>A &lt;= B</th>
<th>A &gt; B</th>
<th>A &lt; B</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCL A</td>
<td>RCL A</td>
<td>RCL A</td>
<td>RCL A</td>
<td>RCL A</td>
<td>RCL A</td>
</tr>
<tr>
<td>X-&gt;T</td>
<td>X-&gt;T</td>
<td>X-&gt;T</td>
<td>X-&gt;T</td>
<td>X-&gt;T</td>
<td>X-&gt;T</td>
</tr>
<tr>
<td>INV</td>
<td>RCL E</td>
<td>X-&gt;T</td>
<td>RCL B</td>
<td>X-&gt;T</td>
<td>RCL E</td>
</tr>
<tr>
<td>X=T</td>
<td>X=T</td>
<td>INV</td>
<td>X&gt;=T</td>
<td>X&gt;=T</td>
<td>X&gt;=T</td>
</tr>
</tbody>
</table>

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 VARTIL 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 ENDDOOPSTACK. The effect is to force program control to exit the current loop. If control is not within a loop (i.e. ENDDOOPSTACK 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 IFSTACK, 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/trignometric 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 WEASIC 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 BIFNLFL files respectively, the code for both "quick" and "long" functions is entered into the symbol table during initialization. BIFNQF and BIFNLFL 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 FNLLEN in the constant declaration block of the EAXS9 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 LAELF 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 GENPARM 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 lookup 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 GCSUB 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 "(A CL nn)" and "(2.33E-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 NCP'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 placeholder. Because the TI-59 requires that particular keycodes be followed by labels, register numbers, or addresses it is easy to check keycode 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 (pointer 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 pointers 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 CPTION 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 WEASIC required restrictions.

While the WEASIC 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 WEASIC **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 OPTION 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Option</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>±0</td>
<td>Generate linker interface file</td>
<td>false</td>
</tr>
<tr>
<td>±1</td>
<td>Generate code for PC-100 printer</td>
<td>true</td>
</tr>
<tr>
<td>±2</td>
<td>Optimize out unnecessary parentheses</td>
<td>true</td>
</tr>
<tr>
<td>±3</td>
<td>Optimize out unnecessary NOP’s</td>
<td>true</td>
</tr>
<tr>
<td>±4</td>
<td>Translated TR-59 code to list file</td>
<td>true</td>
</tr>
<tr>
<td>±5</td>
<td>Image of symbol table to list file</td>
<td>false</td>
</tr>
<tr>
<td>±6</td>
<td>Contents of code structure to list file</td>
<td>false</td>
</tr>
<tr>
<td>±7</td>
<td>Each lexical token to terminal</td>
<td>false</td>
</tr>
<tr>
<td>±8</td>
<td>Each lexical token to list file</td>
<td>false</td>
</tr>
</tbody>
</table>

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.
These procedures are enclosed in dashed lines because they represent a conceptual grouping of system utility programs. As such, no actual scope lines exist where the dashed line occurs.

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 code. 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
### System Utilities

<table>
<thead>
<tr>
<th>System Utilities</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUMP_SEGTBL</td>
<td>INIT_SETS</td>
</tr>
<tr>
<td>DUMP_MEMODULENODES</td>
<td>ADVANCE_CODEPTR</td>
</tr>
<tr>
<td>WRITE_LEADZERO</td>
<td>PRINTLN_MSG</td>
</tr>
<tr>
<td>WRITE_LBL</td>
<td>PRINT_LINESMSG</td>
</tr>
<tr>
<td>WRITE_CODES</td>
<td>DET_LIMIT</td>
</tr>
<tr>
<td>WRITENUM</td>
<td>DIAGS_NEST65BRINVCHK</td>
</tr>
<tr>
<td>HANDLE_#STEP</td>
<td>DIAGS_NEST1LENGTHCHK</td>
</tr>
<tr>
<td>PRINT_CODELIST</td>
<td>RESET_INCLUDED</td>
</tr>
<tr>
<td>INPUT</td>
<td>FIND_MSG</td>
</tr>
<tr>
<td>SET_JMP</td>
<td>DIAGS_NEST1SBRBRK</td>
</tr>
<tr>
<td>SET_JMPTR</td>
<td>BELOW_BREAK</td>
</tr>
<tr>
<td>CLEAN</td>
<td>PRUNE_SAMES</td>
</tr>
<tr>
<td>PRUNE_SAMEF</td>
<td>PASS_BRK</td>
</tr>
<tr>
<td>PRUNE_CREATORS</td>
<td></td>
</tr>
</tbody>
</table>

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_PRIMSEGTBL 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_ADVSEGTBL 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_SEGTBL 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_PINSEGTL. In this operation two primary things happen. First, PROCESS_SBRSEGTL 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 BUILT_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_SUMTOTSER 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 boolean 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_MEMODULENODES 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
INSTRUCTIONS

BLD_MEMODULENODES

INSERT_MEMODULENODES

RECURSE_BLD_MEMMODULENODES

NOT_IN_MODULELIST

BLD_MEMODULECODE

BLD_A_MEMORY

FORM_MEMORY

MARK_INVOKED

PROCESS_SEG

ADD_PROFILE

ADD_CODE

FIND_LABEL

ADVANCE_CODE

GENCODE

OUTPUT_INSTR

OUTPUT_GOODSEG

OUTPUT_BADSEG

OUTPUT_MSGF2

OUTPUT_MSGF1

SET_STEP

PRINT_LINE

WRITE_LABEL

WRITE_NUMBER

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_MEMODULECODE 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_LINE_MSG. 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, CUTFJMSGF1, 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 WEASIC programs facilitated system design. Simple redefinition of the use of WEASIC 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 LBL 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 WEASIC 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 WEASIC 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 WEASIC 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.
E. 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.
C. RECOMMENDATIONS

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 WEASIC 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 runtime 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 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 SLCTRCD 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.
APPENDIX A
WBASIC SUBSET RECOGNIZED BY BAX59

Command Reserved Words:

DATA ENDIF GOTC NEXT REM
ELSE ENDLOOP IF PAUSE RETURN
ELSEIF FOR LET QUIT STOP
END GOSUB LOOF READ UNTIL

OPTION (special—does not follow WBASIC syntax)

Supplemental Reserved Words:

NCT STEP THEN IC
<> <= >= ! *
\ (special—recognized by scanner directly)

Unimplemented Reserved Words:

CHAIN LINEDIT RANDOMIZE SLEEP
CLOSE LOCK REMOVE SORT
DIM MAT RENAME TAGSORT
ENDDOSS ON RESUME UNLOCK
GUESS OPEN SCRATCH USE

OPTION (special—syntax of WBASIC not implemented)

AND OR
# $
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

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

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>

GOSUB: Transfer control to the line specified, until a RETURN is reached (see RETURN).
  GOSUB <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 WEASIC 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, ENDOF, UNTIL).
   LOOP
      ...statements to execute in loop
   ENDOF

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 [<c[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, ENDOF).

ENDCOP ...

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

ENDCOP
...

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

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

UBICN <opt-param> {<opt-param>}

where <opt-param> (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.

0 = generate linker interface file ...............false
1 = generate code for PC-100 printer............true
2 = optimize cut unnecessary parentheses........true
3 = optimize cut 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:

<>  net equal
<=  less than or equal
>=  greater than or equal
**  raise to the power
<  less than
>  greater than
=  equal
!  end of line cmt
+  addition
-  subtraction
*  multiplication
/  division
(  open expression
)  close expression
,  list item separator
.  decimal point

Special Characters:

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

REM This comment is too long for one line, so it &
& 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
LET FN_FACTORIAL...
**Built-in Functions:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS()</td>
<td>Returns the absolute value (magnitude) of parameter</td>
</tr>
<tr>
<td>ACOS()</td>
<td>Returns the arccosine (in radians) of parameter</td>
</tr>
<tr>
<td>ASIN()</td>
<td>Returns the arcsine (in radians) of parameter</td>
</tr>
<tr>
<td>ATN()</td>
<td>Returns the arctangent (in radians) of parameter</td>
</tr>
<tr>
<td>COT()</td>
<td>Returns the cotangent of parameter angle in radians</td>
</tr>
<tr>
<td>CSC()</td>
<td>Returns the cosecant of parameter angle in radians</td>
</tr>
<tr>
<td>EXP()</td>
<td>Returns the value of e raised to the power x</td>
</tr>
<tr>
<td>FP()</td>
<td>Returns the fractional part and sign of parameter</td>
</tr>
<tr>
<td>IP()</td>
<td>Returns the integer part and sign of real parameter</td>
</tr>
<tr>
<td>LOG()</td>
<td>Returns the natural logarithm (base e) of parameter</td>
</tr>
<tr>
<td>LOG10()</td>
<td>Returns the logarithm (base 10) of parameter</td>
</tr>
<tr>
<td>PI()</td>
<td>Returns the value of the constant pi</td>
</tr>
<tr>
<td>RND()</td>
<td>Returns a pseudo-random number in the range (0,1)</td>
</tr>
<tr>
<td>SEC()</td>
<td>Returns the secant of parameter angle in radians</td>
</tr>
<tr>
<td>SIN()</td>
<td>Returns the sine of parameter angle in radians</td>
</tr>
<tr>
<td>SQR()</td>
<td>Returns the square root of parameter</td>
</tr>
<tr>
<td>TAN()</td>
<td>Returns the tangent of parameter angle in radians</td>
</tr>
</tbody>
</table>
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 W BASIC 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. These are a few previously unmentioned items, many of which are essential to successful use of the system.

1. Whether you are translating a prewritten W BASIC 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 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 BIPQF or BIPNLF 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
keycodes 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 "STO 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 \textsc{wBASIC} source code.

19. The BAX59 \textsc{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 \textsc{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 WEASIC 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 WEASIC SYNTAX OR SEMANTIC
* ERRORS. WEASIC 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
(INFILE, OUTPUT, BASICE, MSGF,
READF, SCRATCH);

SYSTEM PARAMETERS

CONST DECLARATIONS (MAIN)

CONST RWCHARCT = 270; (* TOTAL # OF CHARs IN RW ARRAY *)
RWWORDSCT = 72; (* TOTAL # OF WORDS IN RW ARRAY *)
RWIDTHCT = 20; (* # OF CHARs IN LONGEST RW *)
MAXTOKEN = 20; (* MAX ACUM LENGTH -> MAX TOKEN *)
MAXLINLEN = 66; (* MAX LENGTH OF BASIC TEXT LINE *)
MAXEASIIN = 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 *)
FREE = 72; (* MAX # OF AVAILABLE LABELS *)
FUCLEN = 4; (* MAX # STEPS IN QUICK FUNCTION *)
FNLLEN = 15; (* MAX # STEPS IN LONG FUNCTION *)
FNREG = 10; (* REG USED FOR PARM OF LONG FNS *)
FNSTACKLIM = 6; (* MAX SR/BF NESTING LEVEL *)
TEXTLEN = 20; (* MAX # CHARs IN A CODE TEXT IN *)
**GLOBAL DECLARATIONS**

**TI-59 KEY CODES:**

<table>
<thead>
<tr>
<th>Key Code</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>K ZERO</td>
<td>0</td>
</tr>
<tr>
<td>K 1</td>
<td>1</td>
</tr>
<tr>
<td>K 2</td>
<td>2</td>
</tr>
<tr>
<td>K 3</td>
<td>3</td>
</tr>
<tr>
<td>K 4</td>
<td>4</td>
</tr>
<tr>
<td>K 5</td>
<td>5</td>
</tr>
<tr>
<td>K 6</td>
<td>6</td>
</tr>
<tr>
<td>K 7</td>
<td>7</td>
</tr>
<tr>
<td>K 8</td>
<td>8</td>
</tr>
<tr>
<td>K 9</td>
<td>9</td>
</tr>
<tr>
<td>K ZERO</td>
<td>0</td>
</tr>
<tr>
<td>K 1</td>
<td>1</td>
</tr>
<tr>
<td>K 2</td>
<td>2</td>
</tr>
<tr>
<td>K 3</td>
<td>3</td>
</tr>
<tr>
<td>K 4</td>
<td>4</td>
</tr>
<tr>
<td>K 5</td>
<td>5</td>
</tr>
<tr>
<td>K 6</td>
<td>6</td>
</tr>
<tr>
<td>K 7</td>
<td>7</td>
</tr>
<tr>
<td>K 8</td>
<td>8</td>
</tr>
<tr>
<td>K 9</td>
<td>9</td>
</tr>
</tbody>
</table>

**OTHER SYMBOLS:**

- BLANK = •
- ENDLIN = @
- END2IL = %
- PERIOD = .
- COMMA = ,
- UNDERSCORE = _
- EXCLAM = !
- QUOTE = ""
<table>
<thead>
<tr>
<th>Token Number</th>
<th>Token Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>IDENTOK</td>
<td>RWWORD + 1</td>
</tr>
<tr>
<td>1</td>
<td>NUMBERTOK</td>
<td>RWWORD + 2</td>
</tr>
<tr>
<td>2</td>
<td>ENDLINTOK</td>
<td>RWWORD + 3</td>
</tr>
<tr>
<td>3</td>
<td>ENDFILTOK</td>
<td>RWWORD + 4</td>
</tr>
<tr>
<td>4</td>
<td>TOTOK</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>OBTOK</td>
<td>21</td>
</tr>
<tr>
<td>6</td>
<td>CMTOKREM</td>
<td>24</td>
</tr>
<tr>
<td>7</td>
<td>NOTTOK</td>
<td>28</td>
</tr>
<tr>
<td>8</td>
<td>ANDTOK</td>
<td>30</td>
</tr>
<tr>
<td>9</td>
<td>THENTOK</td>
<td>33</td>
</tr>
<tr>
<td>10</td>
<td>ELSETOK</td>
<td>34</td>
</tr>
<tr>
<td>11</td>
<td>QUITTOK</td>
<td>38</td>
</tr>
<tr>
<td>12</td>
<td>STEP TOK</td>
<td>42</td>
</tr>
<tr>
<td>13</td>
<td>ENDLOOP TOK</td>
<td>67</td>
</tr>
</tbody>
</table>

*Token numbers used most often outside of main driver.*
TYPE DECLARATIONS (MAIN)

(* SUBRANGES *)

TYPE BASLINRNG = 0..MAXBASLIN;
TOKENRNG = 0..FWORDCT + 4;
HASHRNG = 0..HASHBASE;
ACCRNG = 0..MXTOLKEN;
LBLRNG = 0..LBLBASE + 1;
REGRNG = 0..99;
KEYRNG = 0..99;
CTEXTRNG = -2..99;

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

LEVEL1 = ARRAY (1..RWCHARCT + 1) OF CHAR;
LEVEL2 = Array (1..RWWORDCT + 1) OF INTEGER;
LEVEL3 = ARRAY (1..RLEVELCT + 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 = RECORD
  NUM : TKSTRING;
  SIGN : CHAR
END;

DATASTORE = ARRAY (1..REGBASE) OF DATAITEM;

(* DATA STRUCTURE WHICH HOLDS THE TEXT TRANSLATION OF *)
(* ALL 11-59 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;
  CCECHAR : PACKED ARRAY (1..TEXTLEN) OF CHAR
END;

CTEXTSTORE = ARRAY ( . . CTEXTRNG . ) OF CODETEXT;
COERCER is a single node in the TI-59 code data structure; LINERCD is a single node in the chain of BASIC 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 BASIC line numbers.

CODEPTR = RECORD
  ADDR : INTEGER;
  KEY : INTEGER;
  JME : CODEPTR;
  SCUP : CODEPTR;
  BAKE : CODEPTR
END;

LINEPTR = RECORD
  LINC : BASLINRNG;
  LPTR : LINEPTR;
  CPTP : CODEPTR
END;

SLCTRCD are symbol table slots; slots are attached to different types: variable IDs, long function IDs, quick function IDs, and parameterless function IDs.

CASE TYPO: IDTPYE OF
  VARID : (REGNO : INTEGER; AUXREG1 : INTEGER; AUXREG2 : INTEGER);
  FNQID : (FNO : BKKEYSEQ);
  FNID : (FNL : LKEYSEQ);
  FNPLINK : SLOTPTR);
  FNPID : (FNREFNO : INTEGER;
  FNPLINK : SLOTPTR);
  FNPLINK : SLOTPTR)
END;
VAR DECLARATIONS (MAIN)

* OPTION 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 : BOOLEAN; (* OPTION 0 = FALSE *)
   PC100  : BOOLEAN; (* OPTION 1 = TRUE   *)
   OPTFA      : BOOLEAN; (* OPTION 2 = TRUE   *)
   CFTNOE     : BOOLEAN; (* OPTION 3 = TRUE   *)
   COFUME     : BOOLEAN; (* OPTION 4 = TRUE   *)
   SYDUME     : BOOLEAN; (* OPTION 5 = FALSE  *)
   DSDUME     : BOOLEAN; (* OPTION 6 = FALSE  *)
   TOKCUT     : BOOLEAN; (* OPTION 7 = FALSE  *)
   TOKHTS     : BOOLEAN; (* OPTION 8 = FALSE  *)

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

LETTERS, DIGITS, ALPANUM : SET OF CHAR;
DOUBLET, DOUBLET : SET OF CHAR;
SPECIALS, SIGNS : SET OF CHAR;
SUBLERROR, CRITICAL : SET OF CHAR;
BINCFTOKS, RELOPTOKS : SET OF TOKENRNG;
TRAILTOKS, SIGNTOKS : SET OF TOKENRNG;
BEGIN EXPRTOKS : SET OF TOKENRNG;
NUMERICKY : SET OF KEYRNG;
RESERVE_REG : SET OF REGRNG;

* RESERVED WORD TABLE CHARACTER AND INDEX ARRAYS.

RWCHAR : LEVEL 1;
FNDWCR : LEVEL 2;
RWLENG : LEVEL 3;

* SCANNER ASSOCIATED GLOBAL VARIABLES.

ACCUM : TKSTRING;
ACGINX : ACCANG;
LINFL  : LNSTRING;
LNFINX : 0..MAXLINLEN + 1;
LINUM, LLINUM, CLINUM : BASLINRNG;
TCKNUM, LTCKNUM : TOKENRNG;
LCOUNT, RCOUNT, ECOUNT : INTEGER;
FLAGCE : BOOLEAN;
* PARSE ASSOCIATED GLOBAL VARIABLES. *

ERRCRTC, WARNCT : INTEGER;
EXITREG : INTEGER;
EXITCT : LBLRNG;
RESERVED : LBLRNG;

CTEXT : CTEXTSTORE;
DATAIST : DATASTORE;
DATAIX, READIX : REGBASE;
INDEXERROR : BOOLEAN;
FIRSTREAD : BOOLEAN;

CLABEL : LELSTACK;
BUCKET : LELSTACK;
IDSLOT : HASH;
FP, LPCUR : SLOTPTR;
LPHEAD, LPTRAIL : LINEPTR;
CP, CECUR : CODEPTR;
FIRSTLP, LSLTP : LINEPTR;
BEGINCP, ENDCP : CODEPTR; (* MARKERS *)

FNSTACK, FNLIST : SLOTPTR;
FNSTACKTK : INTEGER;

IFSTACK, ENDIFSTACK : CODEPTR;
LOOFSACK, ENDCOPSTACK : CODEPTR;
FCRSTACK, NEXTSTACK : CODEPTR;

* FILES *

BWTFLP, LABELF, CTEXTF : TEXT; (* INITIAL FILES *)
BFINCF, BFINLP : TEXT; (* BUILT-IN FNS *)
BASEF, MSGF : TEXT; (* INPUT FILES *)
LISTF, NAMEF, READF : TEXT; (* OUTPUT FILES *)
OUTFILE : TEXT; (* TERMINAL FILE *)
SCRATCH : TEXT; (* LINKER INTERFACE *)
**PRIMITIVE CHAR ROUTINES**

*UPCASE CONVERTS ANY LOWER CASE EBCDIC (OR ASCII) CHAR TO UPPER CASE EQUIVALENT.*

```pascal
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", "s") THEN
    UPCASE := CHR(ORD(CH) - ORD("j") + ORD("J"))
  ELSE IF CH IN ("z") THEN
    UPCASE := CHR(ORD(CH) - ORD("s") + ORD("S"))
  ELSE
    UPCASE := CH
END;
(* UPPERCASE *)
```

*TRANSDIGIT RETURNS THE INTEGER VALUE FOR NUMERIC CHARS*

```pascal
FUNCTION TRANSDIGIT (CH : CHAR) : INTEGER;
BEGIN
  TRANSDIGIT := ORD(CH) - ORD("0")
END;
(* TRANSDIGIT *)
```

*XNUMBER RETURNS INTEGER VALUE OF A NUMERIC CHAR STRING*

```pascal
FUNCTION XNUMBER (ACCUM:TKSTRING; ACCINX:ACCRANG) : INTEGER;
VAR
  I, TEMPNR : INTEGER;
BEGIN
  TEMPNR := 0;
  FOR I := 1 TO ACCINX DO
    TEMPNR := TEMPNR * 10 + TRANSDIGIT (ACCUM (.I.));
  XNUMBER := TEMPNR
END;
(* XNUMBER *)
```

117
(* ZEROPAD WRITES INTEGERS TO AN OUTPUT FILE 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;
END;
(* ZEROPAD *)
---
**SCANNER**

* SCAN USES THE RESERVED WORD ARRAYS TO ISOLATE AND RETURN SINGLE TOKENS FROM THE BASIC 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 CHAR ARE DETECTED IN THE SOURCE FILE.

PROCEDURE SCAN (VAR TCKNUM : TCKENRNG);
VAR TCHAR, I : CHAR;
I : INTEGER;

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

(*-05-01-*
(* LINENR RETURNS NUMBER OF THE CURRENTLY SCANNED WEASIC *)
(* PADLE WITH ZEROS *)
(*---------------------------------------------*)

FUNCTION LINENR : BASLINRNG;
VAR I : EASLINRNG;
BEGIN
READ (BASICF, I);
ZERCPAD (LISTF, I, 5);
LINENR := I
END;

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

(*-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 CHAR EXCEEDS MAXLINLEN *)
(*---------------------------------------------*)

FUNCTION RDLINE : LNSTRING;
VAR I, J, LINLENGTH : INTEGER;
CH : CHAR;
BEGIN
I := 0;
WHILE NC(TOLN(BASICF))) AND (I < MAXLINLEN) DO
BEGIN
I := I + 1;
READ (BASICF, CH);
RDLINE[I] := CH;
WRITE (LISTF, CH)
END;
WRITELN (LISTF);
LINLENGTH := I;
LINBINS := 0;
IF LINLENGTH < MAXLINLEN THEN (* FILL UNUSED W/ BLANKS *)
FOR J := LINLENGTH + 1 TO MAXLINLEN DO
RDLINE[J] := ELANK;

119
IF (LINLENGTH = MAXLINLEN) AND (NOT EOLN (BASICF)) THEN
BEGIN
  WRITE (LISTP, '***F** SCAN ERROR: LENGTH OF TEXT');
  WRITE (LISTP, 'AFTER LINUM > ', MAXLINLEN; ', CHAPS');
  ERRORT := ERRORT + 1;
  REPEAT (* LOCATE THE EOLN CHAR TO RECOVER *)
    GET (BASICF)
  UNTIL EOLN (BASICF)
END;
RDLINE (.LINLENGTH + 1) := ENDLIN; (* INSERT EOLN CHAR *)
GET (BASICF); (* MOVE FILE PTR PAST PASCAL EOLN CHAR *)
IF EOF (BASICF) THEN (* OVERWRITE EOLN CHAR IF EOF *)
RDLINE (.LINLENGTH + 1) := ENDFIL (* RDLINE *)

(*-05-03------------------*)
(* GETNOBLANK 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;
    GETNOBLANK := LINBUF (.LNBINX)
  END;
  (* GETNOBLANK *)

(*-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 *)
FUNCTION PUTANDGET (CH : CHAR) : CHAR;
BEGIN
PUTCHAR (CH);
P Numbe error: unexpected end of file error
END;

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;
NUMSPOOL := I
ELSE
NUMSPOOL := 0
END;
END;
PROCEDURE EXPONENT;
BEGIN
IF UCase (TCHAR) = 'E' THEN
BEGIN
TCHAR := PUTANDGET(TCHAR);
IF TCHAR IN SIGNS THEN
TCHAR := PUTANDGET(TCHAR)
ELSE
TCHAR ('*');
END:
COUNT := NUMSPOOL
END;
(* EXPONENT *)
PROCEDURE DECIMALPT;
BEGIN
IF TCHAR = PERIOD THEN
BEGIN
TCHAR := PUTANIGET(TCHAR);
RCOUNT := NUMSFCOL
END;
END; (* DECIMALPT *)

PROCEDURE ADJUST;
BEGIN
FOR I := 1 TO (2 + ECOUNT) DO
BEGIN
ACCUM(9 + I) := ACCUM(.LCOUNT + RCOUNT + 1 + I);
ACCINX := 11 + ECOUNT
END;
END; (* ADJUST *)
PROCEDURE ADJEXP (DIFF: INTEGER);

VAR E1, E2, EXP, NEWEXP: INTEGER;

BEGIN
  E1 := CRD(ACCUM(-12.)) - ORD('0');
  E2 := CRD(ACCUM(-13.)) - ORD('0');
  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 (LCOUNT > 10) OR ((LCOUNT > 8) AND (SCOUNT <> 0)) THEN
    BEGIN
      ACCUM(-9.) := PERIOD;
      IF ECOUNT = 0 THEN
        BEGIN
          ACCUM(-10.) := 'E';
          ACCUM(-11.) := 'E';
          ACCUM(-12.) := '0';
          ACCUM(-13.) := '0';
          MCVEXP:
          IF SCOUNT = 1 THEN
            BEGIN
              ACCUM(-13.) := ACCUM(-12.);
              ACCUM(-12.) := 'C';
              ACCUM(-13.) := '0';
            END
        END
      ELSE
        BEGIN
          ACCUM(-13.) := ACCUM(-12.);
          ACCUM(-12.) := 'C';
          ACCUM(-13.) := '0';
        END
    END
  ELSE IF (LCOUNT + RCOUNT > 10) OR ((LCOUNT + RCOUNT > 8) AND (SCOUNT <> 0)) THEN
    BEGIN
      IF ECOUNT = 0 THEN
        BEGIN
          RCOUNT := 10 - LCOUNT
          RCOUNT := 0;
          SCOUNT := 2
        END
      ELSE IF (LCOUNT + RCOUNT > 10) OR ((LCOUNT + RCOUNT > 8) AND (SCOUNT <> 0)) THEN
        BEGIN
          IF ECOUNT = 0 THEN
            BEGIN
              RCOUNT := 10 - LCOUNT
              RCOUNT := 11
              RCOUNT := 0;
            END
          ELSE
            BEGIN
              MCVEXP:
              RCOUNT := 8 - LCOUNT
            END
        END
    END;
    (*) ADJUST (*)
  (*)-*)
**-05-11------------------------ LOOKS UP TOKEN IN RESERVE WORD TBL BASED UPON **
** TOKEN LENGTH; RETURNS TOKEN NUMBER; IF NOT FOUND, **
** TCKNUM = IDENTOK (IE. TOKEN IS ASSUMED IDENTIFIER). **
**
FUNCTION FWLOOKUP : INTEGER;
VAR MATCH : BOOLEAN;
CHINEX, WDINEX, LGINDX, ACINDX : INTEGER;
BEGIN
LGINDX := ACCINX;
WDINDX := RWLENG(LGINDX);
REPEAT
MATCH := TRUE;
ACINEX := 1;
CHINDX := RWWORD(WDINDX);
WHILE (MATCH) AND (ACINDX <= LGINDX) DO
BEGIN
IF UPCASE(ACCUM(ACINDX)) <> RWCHAR(CHINDX) THEN
MATCH := FALSE
ELSE
BEGIN
ACINDX := ACINDX + 1;
CHINDX := CHINDX + 1
END
END
WDINDX := WDINDX + 1
UNTIL (MATCH) OR (CHINDX = RWWORD(RWLENG(LGINDX+1)));
IF MATCH THEN
RWLOOKUP := WDINEX - 1 (* BACK-UP THE WORD INDEX *)
ELSE
RWLOOKUP := IDENTOK
(* RWLOOKUP *)
END;

**-05-12------------------------ CMTSCPCL READS AND DISREGARDS THE TEXT OF COMMENTS. **
** FLAGCMT IS USED FOR COMMENTS CONTINUED ON NEW LINE. **
**
PROCEDURE CMTSCPLO;
BEGIN
FLAGCMT := FALSE; (* RESET COMMENT CONTINUATION FLAG *)
WHILE NOT (TCHAR IN CRITICAL) DO
TCHAR := GETCHAR;
IF TCHAR = '*' THEN
FLAGCMT := TRUE
END;
(* CMTSCPLO *)

**-05-13------------------------ RECOVER SCANS AND DISREGARDS THE REMAINDER OF THE **
** CURRENT TOKEN AND STOPS AT START OF NEXT TOKEN. **
**
PROCEDURE RECOVER;
BEGIN
WHILE NOT (TCHAR IN (CRITICAL + BLANK*)) DO
TCHAR := GETCHAR (* SKIP TO NEXT TOKEN AND RETURN *)
END;
(* RECOVER *)

**********************************************************************************
BEGIN (* SCAN MAIN *)

LTOKNUM := 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 *)
RCOUNT := 0;  (* NO. OF DIGITS IN EXponent *)
TCHAR := GETNOBLANK;  (* GET NEXT NON-BLANK CHAR *)

IF (TCHAR = ENDFFIL) THEN
  TCKNUM := ENDFILTCK
ELSE IF (TCHAR = 'E') THEN
  BEGIN
    CLINUM := LINENUM;  (* READ LINE NO. OF CONT LINE *)
    LINBUF := RDLINE;  (* READ TEXT OF CONT LINE *)
    TCHAR := GETCHAR;  (* MOVE LINBUF PAST TRAIL "E" *)
    TCHAR := GETNOBLANK;  (* FIND LEADING "E" ON NEW LINE *)
    TCHAR := GETCHAR;  (* MOVE LINBUF PAST CONT "E" *)
    IF FLAGCMT THEN
      CMTSPool  (* COMMENT CONTINUATION *)
    ELSE
      SCAN (TOKNUM)  (* SCAN NEXT TOKEN AFTER "E" *)
  END
END

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

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

ELSE IF (TCHAR IN DIGITS) THEN
  BEGIN
    LCOUNT := NUMSECCL;
    IF TCHAR = PERIOD THEN
      DECIMALT
    ELSE IF (UPCASE(TCHAR) = 'E') THEN
      FETCHAR(PERIOD);
    EXponent;
    ADJOST;
    TCKNUM := NUMBERTOK
  END
ELSE IF (TCHAR = PERIOD) THEN
    BEGIN
        DECIMALPT;
        EXECNENT;
        ADJGST;
        TCKNUM := NUMBERTOK
    END

ELSE IF (TCHAR IN DAVBLE1) THEN
    BEGIN
        TCHAR := PUTANCGET (TCHAR);
        IF (TCHAR IN DAVBLE2) THEN
            TCHAR := PUTANCGET (TCHAR);
            TCKNUM := RWLOCKUP
        END

ELSE IF (TCHAR IN SPECIALS) THEN
    BEGIN
        TCHAR := PUTANCGET (TCHAR);
        TCKNUM := RWLOCKUP;
        IF TCKNUM = CMTCKEXC THEN (* LOOK FOR EXCLAM CMT *)
            CMTSEQOL
        END

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

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

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

    (* DEPPUGGING TOOL: LISTS TCKNUM AND TCKNUM AS IT IS READ *)
    IF TCKLIS THEN
        WRITE (OUTFILE, ' ':6, TCKNUM:2, ' |', ACCUM, ' |')
    (* OPTION 7 *)
    IF TCKLIS THEN
        WRITE (LISTF, ' ':6, TCKNUM:2, ' |', ACCUM, ' |')
    (* SCAN *)

END;
**** 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 (TOKNUM)
  END;
END (* PRECOVER *)

****-20-************
(*-07-************
PROCEDURE GENKEY (OPCCDE : INTEGER) ;
  FORWARD;

(*-08-************
(* ERROR 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: *)
(* INCREASES THE ERBCR 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
END (* PERROR *)

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

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

PROCEDURE CLOSELINE;
BEGIN
  IF NOT (TCKNUM IN TRAILTOKS) THEN
    SCAN (TCKNUM);
  IF NOT (TCKNUM IN TRAILTOKS) THEN
    PERR
END;

****

* CLOS LINE IS A GENERAL PURPOSE PROCEDURE USED WHEN THE *
* END OF A LINE IS EXPECTED BUT IT IS NOT KNOWN WHETHER *
* OR NOT THE LINE BUFFER INDEX IS IN FRONT OF OR AT THE *
* END OF LINE/FILE CHAR; MAY ALSO BE A COMMENT PRIOR TO. *

**-10-**

* CLOS LINE IS A GENERAL PURPOSE PROCEDURE USED WHEN THE *
* END OF A LINE IS EXPECTED BUT IT IS NOT KNOWN WHETHER *
* OR NOT THE LINE BUFFER INDEX IS IN FRONT OF OR AT THE *
* END OF LINE/FILE CHAR; MAY ALSO BE A COMMENT PRIOR TO. *

**-09-**

* PWARN IS USED TO LOCATE THE CAUSE OF A WARNING FOR THE *
* USER AND TO INCREMENT THE WARNING MSG COUNT; NOTE THAT *
* NORMAL COMPI IIATION CONTINUES. *
SYMBOL TABLE MANAGEMENT Routines

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;

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.);
  GETSLOT (.CURHASH.) := IDSLCT;
END;

FUNCTION FN_CHK (ACCUM : TKSTRING) : BOOLEAN;
BEGIN
  FN_CHK := FALSE;
  IF (UPCASE (ACCUM (.1.)) = 'F' AND
      (UPCASE (ACCUM (.2.)) = 'N' AND
      (ACCUM (.3.) = USCORE))
    THEN
      FN_CHK := TRUE
  END;
END;
FUNCTION NEWREG : INTEGER;
BEGIN
  WHILE NEXTREG IN RESERVE_REG DO
    NEXTREG := NEXTREG + 1;
  NEWREG := NEXTREG;
  IF NEXTREG = (REGBASE + STARTREG) THEN
    (* 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 */
  ELSE
    NEXTREG := NEXTREG + 1
  END;
END;
(* NEWREG *)

FUNCTION NEWLBL : LBLINDEX;
BEGIN
  NEWLBL := CLABEL (LELCT);
  LELCT := LELCT + 1;
  IF LELCT = LBLBASE + 1 THEN
    (* SUMMARY ITEM WHICH INDICATES NUMBER OF LABELS USED MAY *)
    PWARN;
    WRITE (LISTF, '***** LABEL OVERFLOW...> ');
    WRITE (LISTF, LBLBASE:1);
    WRITELN (LISTF, ' IN USE...RESET TO 0.');
END;
(* NEWLBL *)

*((** .......................................................... *)

130
FUNCTION FNSTACKLOOK (ACCUM : TKSTRING) : SLOTPTR;

VAR FLOCK, FARMPTR, TPARAMTR : SLOTPTR;
FOUND : BOOLEAN;

BEGIN
FLOCK := FNSTACK;
FOUND := FALSE;
PARAMTR := NIL;
PARAMTR := NIL;
WHILE (FLOCK <> NIL) AND (NOT FOUND) DO
BEGIN
PARAMTR := FLOCK.FNP;
IF PARAMTR <> NIL THEN
  REPEAT
    PARAMTR := PARAMTR.FNP;
    FOUND := (PARAMTR.IDENT = ACCUM);
    PARAMTR := PARAMTR.SLOT
    UNTIL (FOUND) OR (PARAMTR = NIL);
  FLOCK := FLOCK.FNP_LINK
END;
FNSTACKLOOK := PARAMTR;
IF FOUND THEN
  FNSTACKLOOK := TPARAMTR
END;

(* FNSTACKLOOK *)

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

FUNCTION ENTERID (ACCUM:TKSTRING; ACCINX:ACCRNG) : SLOTPTR;
VAR IDSICT : SLCTPTR;
BEGIN
IDSLOT := GETSLOT (ACCUM, ACCINX);
IF NOT ENCRK (ACCUM) THEN
  WITH IDSLOT DO
    BEGIN
      TIP := VARID;
      REGNC := NEWREG;
      AUXREG1 := -1;
      AUXREG2 := -1;
      WRITEX (NAMEF, ' : 8);
      WRITELN (NAMEF, ' : 3, IDENT)
    END
  ELSE
    WITH IDSLOT DO
      BEGIN
        TIP := FNPID;
        FNP := NIL;
        FNPPLK := NIL;
        FNREGNO := NEWREG;
        LEL := NEWL3I;
        WRITEX (NAMEF, ' : 5, FNREGNO:5, ' : 3, IDENT)
      END
  END;
END; (** ENTERID **)
BEGIN (* IDLOOKUP MAIN *)
  TLOCK := FNSTACKLOCK (ACCUM);
  IF TLOCK = NIL THEN
    BEGIN
      HLOCK := BUCKET (.HASHVAL (ACCUM, ACCINX));
      IF HLOCK <> NIL THEN
        BEGIN
          TLOOK := HLOOK;
          HLOOK := HLOOK.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
          IDLOOKUP := ENTERID (ACCUM, ACCINX)
        END
      ELSE
        IDLOOKUP := TLOOK
    END; (* IDLOOKUP *)
FUNCTION NEWCODE (OPCCDE : INTEGER) : CODEPTR;

VAR CP : CODEPTR;

BEGIN
  NEWCP := (CP):;
  NEWCP.SCEP := NIL;
  NEWCP.JUMP := NIL;
  NEWCP.EAKE := NIL;
  NEWCP.ADDR := -1;
  NEWCP.KEY := OPCODE;
  NEWCODE := CP;
END;

(* NEWCODE *)

PROCEDURE PUTKEY (VAR HOLDP : CODEPTR);

BEGIN
  IF HOLE = NIL THEN
    HOLE := NEWCODE (-1);
    ENDCP := SECP := CPCUR;
    ENDCP := ENDCP.SCEP
END;

(* PUTKEY *)

PROCEDURE GENKEY;

BEGIN
  CPCUR := CPCODE;
  CPCUR.SECP := CPCUR.SECE;
  ENDCP := ENDCP.SCEP
END;

(* GENKEY *)

(* FORWARD DECLARATION WITH *)
(* ERROR HANDLING ROUTINES *)
(* 21 *)

(* INSERTKEY IS USED AFTER ALL CODE HAS BEEN GENERATED TO *)
(* INSERT LABELS TO GOSUB REFERENCES; NOTE THAT ENDCF *)
(* 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 *)
(* POINTER PROCEDURES. *)

PROCEDURE INSERTKEY (CPCODE : INTEGER; VAR LOCUS : CODEPTR);
VAR CP : CODEPTR;
BEGIN
  CP := NEWCODE (CPCODE);
  CP@.SEC.P := LOCUS;
  LOCUS := CP
END;
(* INSERTKEY *)

(*****************************************************************)

(* 22 *)

(* GETNEWHDR CONSTRUCTS AND RETURNS A POINTER TO A LINE *)
(* AND CODE 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 ^BASIC SOURCE *)
(* LINE. (NOTE THAT NEWCODE INITIALIZES CODE NODE) *)

FUNCTION GETNEWHDR (LINUM : BASLINNNG) : LINEPTR;
VAR LF : LINEPTR;
CP : CODEPTR;
BEGIN
  NEW (LF);
  LP@.CPTR := NEWCODE (-1);
  LP@.LINC := LINUM;
  LP@.LPTR := NIL;
  GETNEWHDR := LF
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@.SEC.P := LP@.CPTR;
  CPCUR := LP@.CPTR;
  CPCUR@.EAKP := ENDCF
END;
(* PUTNEWHDR *)

(*****************************************************************)
(* 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 : BASLINENUM;
BEGIN
  LNUMBER := LINUM;
  IF TOKNUM = ENDLINTCK THEN
    LNUMBER := LINUM;
  IF LPCUR.LPTR <> NIL THEN (* IF LINE DATA NODES *)
    BEGIN (* IF LINE DATA NODES *)
      (* BLOCKED FROM JUMP REFS. *)
      IF (LPCUR.LPTR.LINO) = LNUMBER THEN (* IF LINE NUMBER *)
        BEGIN (* IF LINE NUMBER *)
          LPCUR := LPCUR.LPTR; (* CURRENT LINE NUMBER IS *)
          ENDCPSEQP := LPCUR.CPTR;
          CPCUR := LPCUR.CPTR;
          CPCUR.BAKP := ENDCP
          END
        ELSE
          BEGIN
            LP := GETNEWHDR (LNUMBER);
            PUTNEWHDR (LPCUR, LP)
            END
        END
      ELSE
        BEGIN
          LP := GETNEWHDR (LNUMBER);
          PUTNEWHDR (LPCUR, LP)
          END
        END
END; (* SETLINE *)

(***********************************************************************)
PROCEDURE SETJMPEXT (LINUM : BASLINRNG);
BEGIN
  IF LINUM > LPCUR@.LING THEN (* FORWARD JUMP *)
    LFLEAD := LPCUR@.LPTR;
    LPTAIL := LPCUR;
    WHILE (LPTAIL@.LINO < LINUM) AND (LPLEAD <> NIL) DO
      LFLEAD := LPLEAD@.LPTR;
      LPTAIL := LPLEAD@.LPTR
    END;
    IF LPTAIL@.LINO = LINUM THEN
      CFCUR@.JMPF := LPTRAIL@.CPTR; (* SET JMPF PTR *)
      LPTRAIL@.CPTR@.ADDR := 0 (* MARK JMPF TERMINAL *)
    ELSE
      LFLEAD := GETNEWHDR (LINUM);
      LSTLP := LPLEAD@.LPTR;
      CFCUR@.JMPF := LSTLP@.CPTR; (* SET JMPF PTR *)
      LSTLP@.CPTR@.ADDR := 0 (* MARK JMPF TERMINAL *)
    END
  ELSE (* BACKWARD JUMP *)
    LFLEAD := FIRSTLP;
    WHILE (LPLEAD@.LINO <> LINUM) AND (LPLEAD <> NIL) DO
      LFLEAD := LPLEAD@.LPTR;
    IF LPLEAD@.LING = LINUM THEN
      CFCUR@.JMPF := LPLEAD@.CPTR; (* SET JMPF PTR *)
      LPLEAD@.CPTR@.ADDR := 0 (* MARK JMPF TERMINAL *)
    END (* ASSUMES THAT A LINO = LINUM ALWAYS EXISTS ! *)
  END (* SETJMPEXT *)
END;
FUNCTION CALL ROUTINES

PROCEDURE PEXPR;

PROCEDURE PPRI3ARY;

PROCEDURE GENPNQ (OPND : SLOTPTR);

VAR  I : 1..FNQLEN;

BEGIN
  SCAN (TCKNUM);
  IF TCKNUM <> OPARENTOK THEN
    FERRCR
  ELSE
    BEGIN
      SCAN (TOKNUM);
      PEXPR:
      I := 1;
      WITH CPNDa DO
      REPEAT
        GENKEY (FNQ(.I.));
        I := I + 1;
        UNTIL (I >= FNQLEN) OR (FNQ(.I.) = K_NOP)
      END;
    END;
  END;

PROCEDURE CHKFNLLIST (VAR IDSLOT : SLOTPTR);

VAR  LISTPTR, HCLDPTR : SLOTPTR;

BEGIN
  LISTPTR := FNLLIST;
  IF LISTPTR <> NIL THEN
    BEGIN
      LISTPTR := OR (LISTPTR@.FNLLINK);
      UNTIL (LISTPTR = NIL) OR (LISTPTR@.IDENT) = IDSLOT;
      IF LISTPTR = NIL THEN
        BEGIN
          HCLDPTR := FNLLIST;
          FNLLIST := IDSLOT;
          IDSLOT@.FNLLINK := HCLDPTR
        END;
    END;
END;

END;
PROCEDURE GENFNL (VAR IDSLOT : SLOTPTR);
BEGIN
SCAN (TOKNUM);
IF TOKNUM <> OPARENTOK THEN
  PERCH
ELSE
BEGIN
  SCAN (TOKNUM);
  PERCH;
  GENKEY (K STO);
  GENKEY (FNLREG);
  GENKEY (K SBR);
  GENKEY (IDSLOT.LBL);
  CHKFNL1IST (IDSLOT)
END
END;
(* GENFNL *)

FUNCTION NEWPARM : SLCTPTR;
VAR PARMSLCT : SLOTPTR;
BEGIN
NEW (PARMSLOT);
WITH PARMSLOT DO
BEGIN
  TYPE := VARID;
  REGNC := NEWREG;
  WRITELN (NAMEF, ' : 5, REGNO: 5, ' : 5, ' (FN PARAMETER) ');
  SLCT := NIL
END;
NEWPARM := PARMSLOT
END;
(* NEWPARM *)

(* NEWPARM *)
PROCEDURE GENPARM (VAR IDSLOT : SLOTPTR);
VAR PARMPTR : SLOTPTR;
BEGIN
SCAN (TOKNUM);
IF TOKNUM = OPARENVCK THEN
BEGIN
PARMPTR := IDSLOT3.FNP;
IF PARMPTR = NIL THEN
BEGIN
PARMPTR := NEWPARM;
IDSLOT3.FNP := PARMPTR
END;
END;
REPEAT
SCAN (TOKNUM);
IF EXPR.
GENKEY (KSTC);
GENKEY (ERMTR.EGNO);
IF TOKNUM = CCMMATOK THEN
BEGIN
IF PARMPTR.SLOT = NIL THEN
PARMPTR.SLOT := NEWPARM;
PARMPTR := PARMPTR.SLOT
(* NEXT PARAMETER *)
END
UNTIL (TOKNUM = CPARENTO)
(* PEXPR WILL FIND ')
OR (TOKNUM IN TRAILTOKS)
(* OR WILL FIND END *)
END
(* GENPARM *)

(*GENFNFGENERATESCODESEQUENCEWHICHCALLSAPARMFNF.*)

PROCEDURE GENFNFG (VAR IDSLOT : SLOTPTR);
BEGIN
GENFARM (IDSLOT);
GENKEY (KCPAREN);
GENKEY (K_SBR);
GENKEY (IDSLOT.LBI);
GENKEY (K_CPAREN)
END:
(* GENFNPG *)
**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...');
      WRITELN (LISTF, 'RETURN ADDRESSES.');
    END
  END;
END; (* PUSHFN *)

(*-33---------------------------------------------*)

(* PCFN POPS A FNP SLOT OFF TOP OF FNP ACTIVATION STACK. *)

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

(*----------------------------------------------------------------*)
PROCEDURE FILLPAMIDS (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$FNP;
  SCAN (TOKNUM);
  IF TOKNUM <> IDENTCK THEN PERCR
  ELSE BEGIN
    WHILE TOKNUM <> CPARENTOK DO
      BEGIN
        FNPARM$IDENT := ACCUM;
        SCAN (TOKNUM);
        IF TOKNUM = COMMA TOK THEN
          BEGIN
            SCAN (TOKNUM); (* SCAN TOKEN AFTER ',' *)
          END
        END
      END;
  END;
END;

(* FILLPAMIDS * )

(* 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 (FNSSTACK$FNREGNO);
  GENKEY (K_INVSR);
  POPFN;
  IF NOT (TOKNUM IN TRAILTOKS) THEN (* GUARD AGAINST OVER *)
    SCAN (TOKNUM) (* SCANNING END LINE IF CALLED BY PDEF.*)
END;

(* PFNEND *)

(*-34-*)
PROCEDURE PDEF;
VAR FNSLCI : SLCTPTE;
BEGIN
  SCAN (TCKNUM);
  IF NOT FN_CHK (ACCUM) THEN
    PERROR
  ELSE
    BEGIN
      FNSLCI := IDLOCKUP (ACCUM, ACCINX);
      GENKEY (K LBL);
      GENKEY (K ZERO);
      GENKEY (K STO);
      GENKEY (FNSLOT . PNSREGN);
      PUSHFN (FNSLOT);
      SCAN (TOKNUM);
      IF TCKNUM = OPARENTOK THEN (* LOOKING FOR PARMIDS (FNSLOT) *)
        FILLPARMIDS (FNSLOT);
      IF TCKNUM = EQUALTOK THEN (* LOOK FOR ONE LINE FN *)
        BEGIN
          SCAN (TOKNUM);
          PEND (* GENERATE THE RETURN FROM ONE LINE FN *)
        END;
      CLOSELINE
    END;
  END;
END;

(* PDEF *)

(*******************************)

(* 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 ONCE AND ARE, THUS, ON THE FNLLIST. *)

PROCEDURE GETFNLS;
VAR LISTPTR : SLOTPTR;
I : 1 .. FNLEN + 1;
BEGIN
  LISTPTR := FNLLIST;
  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 . FN (. I .) <> K_NOP) DO
        BEGIN
          GENKEY (LISTPTR . FN (. I .));
          I := I + 1
        END;
      GENKEY (K INVERSE);
      LISTPTR := LISTPTR . FNLLINK (* NEXT LONG FN ON LIST *)
    END;
END;
EXPRESSION GENERATOR Routines

PROCEDURE GENID;
VAR CFNC : SLOTPTR;
BEGIN
OPND := IDLOOKUP (ACCUM, ACCINX);
CASE GENID@TYP OF
VARIL : BEGIN
IF OPND@REGNO = -314 THEN (* REGNO FOR PI *)
   GENKEY (K_PI)
   ELSE BEGIN
   GENKEY (K_RCL);
   GENKEY (OPND@REGNO)
   END;
FNQIC := GENFNQ (CFND);
FNLIE := GENFNL (CFND);
FPID := GENFNP (CFND)
END (* CASE *); (* GENID *)

PROCEDURE GENNUM;
VAR I, DECPTRLoc, ESIGNLOC : INTEGER;
BEGIN
DECPTRLoc := LCCUNT + 1;
FOR I := 1 TO LCOUNT DO
   GENKEY (TRANSDIGIT (ACCUM(.I.)))
IF ECCUNT > 0 THEN BEGIN
   GENKEY (K_DECPT);
   FOR I := (DECPTRLoc + 1) TO (DECPTRLoc + RCOUNT) DO
   GENKEY (TRANSDIGIT (ACCUM(.I.)))
   END;
IF ECCUNT > 0 THEN BEGIN
   ESIGNLOC := LCCUNT + 1 + RCOUNT + 2;
   IF ACCUM (.DECPTRLoc.) <> PERIOD THEN
      ESIGNLOC := ESIGNLOC + 1;
   GENKEY (K_NEG);
   IF ACCUM (.ESIGNLOC.) = '-' THEN
      GENKEY (K_NEG);
   FOR I := (ESIGNLOC + 1) TO (ESIGNLOC + RCOUNT) DO
      GENKEY (TRANSDIGIT (ACCUM(.I.)))
   END; (* GENNUM *)

144
(*-40-*)

(* PEXPR PARSES AND GENERATES CODE FOR EXPRESSIONS. *)

(*-*-*)

PROCEDURE PEXPR;

(* FORWARD DECLARATION WITH *)

(* FUNCTION CALL ROUTINES *)

BEGIN

GENKEY (K_OPAREN);

PPRIMARY;

WHILE TOKNUM IN BINCPTCKS DO

BEGIN

CASE TOKNUM OF

PLUSSTOK : GENKEY (K_ADDOP);

MINUSTOK : GENKEY (K_SUBOP);

MULTCK : GENKEY (K_MULOPT);

DIVSTOK : GENKEY (K_DIVOP);

EXPTOK : GENKEY (K_POWR);

END: (* CASE *)

IF NOT (TOKNUM IN TRAILTOKS) THEN BEGIN

SCAN (TOKNUM);

PPRIMARY

END

GENKEY (K_CPAREN)

END;

(* PEXPR *)

(*-41-*)

(* PPRIMARY PARSES AND GENERATES CODE FOR A PRIMARY ITEM *)

(* EXPECTED AS PART OF EXPRESSIONS. *)

(*-*)

PROCEDURE PPRIMARY;

(* FORWARD DECLARATION WITH *)

(* FUNCTION CALL ROUTINES *)

BEGIN

CASE TOKNUM OF

PLUSSTOK : BEGIN

SCAN (TOKNUM);

PPRIMARY

END;

MINUSTOK : 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)

NUMBRTCK : BEGIN

GENNUM;

SCAN (TOKNUM)

END

END (* CASE *)

(* PPRIMARY *)

(*-*)

145
PROCEDURE FCNDITION;

VAR RELCF : TOKENRING;
    INVERT : BOOLEAN;

BEGIN
    INVERT := FALSE;
    SCAN (TOKNUM);
    IF TOKNUM = NOTTOK THEN
        BEGIN
            INVERT := TRUE;
            SCAN (TOKNUM)
        END;
    PEXER;
    GENKEY (K X T);
    IF INVERT THEN
        CASE TOKNUM OF
            EQUALTCK : TOKNUM := NOTTOK;
            GTECK : TOKNUM := LTECTK;
            LTECTK : TOKNUM := GTECTK;
            GTCK : TOKNUM := LTCK;
            LTCK : TOKNUM := GTCK;
        END; (* CASE *)
    RELCF := TCKNUM;
    SCAN (TCKNUM);
    PEXER;
    CASE RELCF OF
        EQUALTCK : BEGIN
            GENKEY (K_INV);
            GENKEY (K_IPXEQT)
        END;
        NOTTOK : BEGIN
            GENKEY (K_IPXEQT);
            GENKEY (K_IPXGET)
        END;
        GTECK : BEGIN
            GENKEY (K_INV);
            GENKEY (K_IPXGET)
        END;
        LTECTK : BEGIN
            GENKEY (K_INV);
            GENKEY (K_IPXGET)
        END;
        LTCK : BEGIN
            GENKEY (K_X T);
            GENKEY (K_IPXGET)
        END;
        END (* CASE *)
    END; (* FCNDITION *)
(* LOOPING ROUTINES *)

(* PUSHCODE PUSHES THE RCODE DATA NODE ONTO THE LOOP/IF STACK DESIGNATED BY STACK. *)

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

(* POPCODE POPS AND RETURNS THE CODE DATA NODE ON THE TOP OF THE LOOP/IF STACK DESIGNATED. *)

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

(* 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 BE POPPED AND INSERTED INTO THE CODE AT THE APPROPRIATE POSITION. *)

PROCEDURE SETFWDJMP (STACK : CODEPTR);
BEGIN
CUR@.JMEP := 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 *)
(* SETBAKJMP IS SIMILAR TO SETWDLJMP EXCEPT IN THIS CASE *)
(* THE FIRST NODE OF A POTENTIAL ADDRESS SPACE PAIR IS *)
(* PUSHED ONTO THE DESIGNATED LOOP/IP STACK AFTER ITS *)
(* JMFP HAS BEEN SET TO A CODE 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
  JCODE := NEWCODE (-2);
  JCODE@.JMFP := CPCUR;
  CPCUR@.ADDR := 0;
  GENKEY (K_NOP);
  PUSHCODE (JCODE, STACK)
END; (* SETBAKJMP *)

(* 47 *)
(* FLOOP GENERATES CODE 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 FLOOP;
BEGIN
  SETBAKJMP (LOOPSTACK);
  PUSHCODE (NEWCODE (K_NOP), ENDLOOPSTACK);
  SCAN (TCKNUM);
  CLOSELINE
END; (* FLOOP *)

(* 48 *)
(* PWHILE GENERATES CODE FOR THE WHILE COMMAND *)
(* IT IS SIMILAR TO FLOOP EXCEPT IT INSERTS CODE TO *)
(* EVALUATE A BOOLEAN EXPRESSION (CONDITION) *)

PROCEDURE PWHILE;
BEGIN
  SETBAKJMP (LOOPSTACK);
  PCONDITICN;
  PUSHCODE (NEWCODE (K_NOP), ENDLOOPSTACK);
  SETWDLJMP (ENDLOOPSTACK)
END; (* PWHILE *)

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

PROCEDURE PENDLOCP;
VAR JCODE : CODEPTR;
BEGIN
  IF TCKNUM = ENDLOOPTOK THEN
    GENKEY (K GTO);
    JCODE := PPCODE (LOPSTACK);
    PUTKEY (JCODE);
    GENKEY (-2);
    JCODE := PCPCODE (ENDDLOPSTACK);
    PUTKEY (JCODE);
    CLOSeline
  END;
END;

---------------------------------------------------------------------
(* PENDLOCP *)
---------------------------------------------------------------------

PROCEDURE PUNTIL;
BEGIN
  PCCNDITION;
  PENDLOCP
END;

---------------------------------------------------------------------
(* PUNTIL *)
---------------------------------------------------------------------

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 @ PEGNC);
      GENKEY (K GTO);
      JCODE := PCPCODE (FORSTACK);
      PUTKEY (JCODE);
      GENKEY (-2);
      JCODE := FORPCODE (NEXTSTACK);
      PUTKEY (JCODE);
      CLOSeline
  END;
END;

---------------------------------------------------------------------
(* PNEXT *)
---------------------------------------------------------------------
PROCEDURE FFCR;

VAR ISICT : SLOTPTR;

BEGIN  
  SCAN (TCKNUM);
  ISICT := IDLOOKUP (ACCUM, ACCINX);
  ISLOTO_AUXREG1 := NEWREG;
  SCAN (TCKNUM);
  SCAN (TCKNUM);
  PEXER;
  GENKEY (K STO);
  GENKEY (ISLOTO_REGNO);
  SCAN (TCKNUM);
  PEXER;
  GENKEY (K STO);
  GENKEY (ISLOTO_AUXREG1);
  IF TCKNUM = STEPTOK THEN
      BEGIN
          SCAN (TOKNUM);
          PEXER;
          GENKEY (K STO);
          ISLOTO_AUXREG2 := NEWREG;
          GENKEY (ISLOTO_AUXREG2);
      END;
      PUSHCODE (NEWCODE (-2), FCSRSTACK);
      FORSTACK. JMP := CPCUR;
      GENKEY (K ECL);
      GENKEY (ISLOTO_REGNO);
      GENKEY (K X T);
      GENKEY (K ECL);
      GENKEY (ISLOTO_AUXREG1);
      GENKEY (K INV);
      GENKEY (K IPXGET);
      PUSHCODE (NEWCODE (K NOP), NEXTSTACK);
      CPCUR. JMP := NEXTSTACK;
      GENKEY (-2);
      GENKEY (-2);
      CLCSELINE
  END;

(* PFOR *)
IF-BRANCHING ROUTINES

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

PROCEDURE QUITERROR;
BEGIN
WRITE (LISTP, "***** ATTEMPT TO "QUIT" WHILE NOT ");
WRITE (LISTP, "INSIDE A LOOP.");
QUITERROR;
END;

(* QUITERROR *)

(* 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 (ENDLOCPSTACK = NIL) THEN QUITERROR;
ELSE BEGIN
   GENKEY (K-GTO);
   SETFELJMP (ENDLOCPSTACK);
   SCAN (TOKNUM):
   IF (TOKNUM = NUMETOK) THEN BEGIN
   PSUBERROR;
   WRITE (LISTP, "***** "QUIT" DOES NOT ACCEPT ");
   WRITE (LISTP, "LINE NUMBERS THIS IMPLEMENT.");
   END ELSE CLCSELINE
END;
END;

(* PQUIT *)

PROCEDURE PTHENLINE;

(* PTHENLINE *)
PROCEDURE PTHENELSE;
BEGIN
SCANN (TCKNUM); IF TCKNUM = NUMBERTOK THEN BEGIN
GENKEY (K_GTO);
PTHENLINE
END
ELSE IF TCKNUM = QUITTOK THEN FECUIT
ELSE BEGIN
WRITE (LISTF; "**** "IF-THEN-ELSE" LIMITED TO ");
WRITELN (LISTF; "QUIT" OR LINE NUMBERS.");
PERERROR
END;
(* PTHENELSE *)

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

PROCEDURE PTHENQUIT;
BEGIN
IF ENDICOPSTACK = NIL THEN CUITERROR
ELSE BEGIN
SETFWLJUMP (ENDICOPSTACK);
SCANN (TOKNUM);
IF TOKNUM = ELSETOK THEN PTHENELSE
ELSE CLOSELINE
END;
(* PTHENQUIT *)
PROCEDURE PIF;
BEGIN
  PCONDITION;
  IF TOKNUM = THENTOK THEN
    BEGIN
      SCAN (TOKNUM);
      IF TOKNUM = NUMERTOK THEN
        THENLINE
      ELSE IF TOKNUM = QUITTOK THEN
        THENQUIT
      ELSE
        ERROR
    END
  ELSE
    BEGIN
      PUSHCODE (NEWCCDE (K_NOE), ENDIFSTACK);
      CPCURB.BAKF := ENDIFSTACK;
      ENDIFSTACK.BAKE := CPCURB;
      PUSHCODE (NEWCCDE (K_NOP), IFSTACK);
      S2TFWDMP (IFSTACK);
      CLCSELINE
    END;
END;
(* PIF *)

(* ELSE_ADJUST PERFORMS HOUSE-KEEPING ON THE VARIOUS 'IF' *)
(* STATEMENTS ENCOUNTERED; IF-ENDIF REQUIRES A DIFFERENT *)
(* SEQUENCE OF PUSH/PUSH STACK THAN DO25 IF-ELSE-ELSE-ENDIF OR *)
(* IF-ELSE-ELSE-ELSE-ENDIF. *)

PROCEDURE ELSE_ADJUST;
BEGIN
  WITH ENDIFSTACK@ DC
  BEGIN
    IF BAKP <> NIL THEN
      BEGIN
        BAKP.BAKP := NIL;
        BAKP.BAKE := NIL;
        BAKP := NIL
      END
  END;
END;
(* ELSE_ADJUST *)

(*----------------------------------------------------------*)
PROCEDURE PELSEIF;
VAR JCODE: CODEPTR;
BEGIN
ELSE ADJUST;
GENKEY (K GTO);
SETFWDJMP (ENDIFSTACK);
JCODE := POPCODE (IFSTACK);
PUTKEY (JCODE);
PCCNDITION;
PUSHCODE (NEWCODE (K NOP), IFSTACK);
SETFWDJMP (IFSTACK);
CLOSELINE
END;
(* PELSEIF *)

PROCEDURE PELSE;
VAR JCODE: CODEPTR;
BEGIN
ELSE ADJUST;
GENKEY (K GTO);
SETFWDJMP (ENDIFSTACK);
JCODE := POPCODE (IFSTACK);
PUTKEY (JCODE);
SCAN (JTCNUM);
CLOSELINE
END;
(* FELSE *)
**PROCEDURE PENDIF;**

**VAR DUMPC, JCODE : CODEPTR;**

**BEGIN**

**WITH ENDIFSTACK@ DO**

**BEGIN**

**IF EAKP <> NIL THEN**

**BEGIN**

**EAKP@.EAKP := NIL;**

**EAKP := NIL;**

**DUMPC := PCFCODE (ENDIFSTACK);**

**DUMPC@.ADDR := -1;**

**JCCDS := PCFCODE (IFSTACK);**

**Putdown (JCCDS) ;**

**END**

**ELSE**

**BEGIN**

**JCODE := PCFCODE (ENDIFSTACK);**

**JCODE@.ADDR := 0;**

**Putdown (JCODE) ;**

**END**

**END;**

**SCAN (ICKNUM);**

**CLOSELINE**

**END:**

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

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

(********************************************************************)

(*-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 LOOPS, SEGS, OR FUNCTIONS WOULD RENDER THE DATA MEANINGLESS. THIS ROUTINE READS THE DATA STORED AS ITS PARAMETERS, COUNTS THEM, AND STORES THEM IN AN ARRAY OF RECORDS WHICH IS ACCESSED BY SUBSEQUENT REAL COMMANDS. (*-84-*)

PROCEDURE PDATA;
VAR CATASIGN : CHAR;

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

(CLOSELINE (* PDATA *))

(********************************************************************)
**PREAD**

* PREAD ONLY GENERATES A NOP INSTRUCTION TO ALLOW FOR *
* LINE-ORIENTED JUMP REFERENCES; OTHERWISE, THIS COMMAND *
* WRITES THE READF FILE WHICH INDICATES DATA VALUES FOR *
* RESPECTIVE REGISTERS AND THEIR W-BASIC VARIABLE NAMES; *
* THE READF FILE IS USED TO INPUT DATA PRIOR TO PROGRAM *
* EXECUTION ON THE TI-59, THUS, SAVING PROGRAM STEPS. *
* THE CONSTRUCT 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');
      WRITELN (READF, MSGF, 9);
      FIRSTREAD := PAISE
   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.');
               WRITE (READF, '******** READ PAST DATA ');
               WRITE (READF, 'INDEX...SUBSEQUENT ');
               WRITE (READF, 'READ/DATA IGNORED.');
               INDEXERROR := TRUE;
               PRECOVER
            END ELSE BEGIN
               WRITE (READF, '  :5);
               WRITE (READF, DATALIST (.READIX.) .SIGN);
               WRITE (READF, DATALIST (.READIX.) .NUM);
               WRITE (READF, ' :2');
               ZECPEAD (READP, REGNO, 2);
               WRITELN (READF, ' :3, IDENT);
               READIX := READIX + 1;
               SCAN (TOKNUM);
               IF TOKNUM = COMMATOK THEN BEGIN
               END ELSE PRECOVER;
               CLOSELINE
            END;
         END
      END
   END
END;

(* PREAD *)

(*******************************************************************************)
PROCEDURE PRESTORE;

BEGIN
  GENKEY (K_NOP);
  READIX := -1;
  SCAN (TOKNUM);
  CLOSeline
END;

PROCEDURE PINPUT;

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

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

END;
(*-67-*)
(* PRINT PARSES A LIMITED FORM OF THE WBasic "PRINT" *)
(* STATEMENT: IT ALLOWS EXPRESSIONS, VARIABLE NAMES, AND *)
(* LITERAL NUMERICS IN THE LIST OF PARAMETERS. *)

PROCEDURE PRINT:
BEGIN
SCANNUM;
WHILE SCANNUM IN BEGIN_EXPRTOKS DO
BEGIN
EXPR;
IF EC100 THEN
GENKEY (K_PRT)
(* WITH PC100 *)
ELSE
BEGIN
GENKEY (K_PAUSE);
GENKEY (K_RS)
END;
IF SCANNUM = COMMATOK THEN
(* CAN SEPARATE ITEMS BY *)
SCANNUM (* COMMAS OR BLANKS. *)
END;
IF EC100 THEN
GENKEY (K_ADV);
(* WITH PC100 *)
CLOSILINE
(* PPRINT *)
PROCEDURE POTION;
VAR
  SWITCH : BOOLEAN;
  TOGGLE : INTEGER;
BEGIN
  GENKEY (K_NOP):
  SCAN (TOKNUM):
    WHILE NOT (TOKNUM IN TRAILTOKS) DO BEGIN
      SWITCH := TRUE;
      IF TOKNUM IN SGGNTOKS THEN BEGIN
        IF TOKNUM = MINUSTOK THEN
          SWITCH := FALSE;
          SCAN (TOKNUM)
        END;
      IF TOKNUM = NUMERTOK THEN BEGIN
        TOGGLE := XNUMBER (ACCUM, ACCINX);
        IF TOGGLE IN ('0..8.') THEN CASE TOGGLE OF
          0 : LRK59 := TRUE;
          1 : PC100 := SWITCH;
          2 : OPTFAR := SWITCH;
          3 : OPTNOP := SWITCH;
          4 : CODUMP := SWITCH;
          5 : SYLUMP := SWITCH;
          6 : DSDUMP := SWITCH;
          7 : TOKCUT := SWITCH;
          8 : TOKIS := SWITCH;
        END (* CASE *)
      ELSE BEGIN
        PWARN;
        WRITE (LISTP, '**** NO SUCH OPTION...');
        WRITELN (LISTP, ACCUM)
      END
    END
  ELSE BEGIN
    PWARN;
    WRITE (LISTP, '**** OPTION PARAMETERS ARE ');
    WRITELN (LISTP, '-8..0..+8 ONLY.');
    SCAN (TOKNUM)
  END; GCSELINE
END;

(* POETICN *)
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 = VARI THEN GENKEY (RESULT@.REGNO)
ELSE IF RESULT@.TYP = FNPID THEN GENKEY (RESULT@.FNRREGNO)
ELSE PERROR
END;
END;

(* PNOLET *)

PROCEDURE PLET;
BEGIN
SCAN (TCKNUM);
PNCLET
END;

(* PLET *)

PROCEDURE PREM;
BEGIN
GENKEY (K_NOP);
SCAN (TCKNUM)
END;

(* PREM *)

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

(* PLET PARSES AND GENERATES CODE FOR A 'LET' STATEMENT. *)

(* PREM GUARDS AGAINST USE OF A GOTO DIRECTED TO A REM NODE WHICH CAN BE REFERENCED BY A JMP POINTER; THE SCANNER HAS RESPONSIBILITY FOR SKIPPING OVER THE CMT TEXT. *)
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 BASIC LINE NUMBER REFERENCED IN THE GOTO COMMAND.

PROCEDURE PGOTO;
BEGIN
GENKEY (K GTO);
SCAN (TCKNUM);
IF TCKNUM <> NUMBERTOK THEN
  ERRCR
ELSE
  BEGIN
    SETJMPFEXT (XNUMER (ACCUM, ACCINX));
    GENKEY (-2);
    GENKEY (-2);
  END;
CLCSELFINE
END;

(*************************************************************************)

PGOSUB GENERATES A CALL TO A SUBROUTINE REFERENCED BY BASIC LINE NUMBER; NOTE THAT ALTHOUGH BASIC CALLS ON SUBROUTINES BY LINE NUMBER, THE TI-59 CODE GENERATED 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 IS DONE AFTER ALL CODE HAS BEEN GENERATED. NOTE THAT THIS ROUTINE NEITHER CHECKS FOR NOR DOES IT KNOW 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 (TOKNUM);
IF TCKNUM <> NUMBERTOK THEN
  ERRER
ELSE
  BEGIN
    SETJMPFEXT (XNUMER (ACCUM, ACCINX));
    GENKEY (NEWLBL);
  END;
CLCSELFINE
END;

(*************************************************************************)
PROCEDURE PRETURN;

BEGIN
  GENKEY (K INVSBR);
  CLOSeline
END;

PROCEDURE PPAUSE;

BEGIN
  GENKEY (K);  (* VOID *)
  GENKEY (31);  (* LEARN *)
  CLOSeline
END;

(* PRETURN GENERATES THE RETURN FROM A SUBROUTINE *)
(* STRUCTURED PROGRAMMING DISCIPLINE DEMANDS A RETURN PER *)
(* EACH SUBROUTINE CALL 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 *)
(* GENERATION OF A ONE-LINE FUNCTION) *)

PPAUSE GENERATES (62) (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 *)
(62) (31) WHICH CANNOT BE ENTERED DIRECTLY ANYWAY; THEN *)
EACH 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)
FOLOWED 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 *)
LOCATION OF ANY 'PAUSE' STATEMENTS PLACED IN THE *)
WBASIC SOURCE CODE; THIS IMPLEMENTATION OF THE 'PAUSE' *)
INSTRUCTION PROVIDES A CONVENIENT AND RECOGNIZABLE *)
DEEUGGING/TRANSLATION TOOL WHICH CARRIES A LOW *)
OVERHEAD IN TERMS OF REGISTER/PROGRAM STEP USE. *)
PROCEDURE ESTOP;
VAR I : 1..4;
BEGIN
GENKEY (K_CE);
FOR I := 1 TO 3 DO
GENKEY (B);
GENKEY (K_RS);
CLOSILINE
END;
(* ESTOP *)

PROCEDURE PEND;
BEGIN
GENKEY (K_NOP);
TOKNUM := ENDFILTOK
END;
(* PEND *)
PROCEDURE PUTGOSUBLBL (LBL : LELRNG; VAR LBLP : CODEPTR);
BEGIN
  WITH LELFA.BAKP DO
  BEGIN
    INSERTKEY (LBL, SEQP);
    INSERTKEY (K_LBL, SEQP)
  END;
END; (* PUTGOSUBLBL *)

PROCEDURE FINDGOSUBLBL (VAR START : CODEPTR);
VAR TRAVELP, TAILP : CODEPTR;
BEGIN
  TRAVELP := START.SEQP;
  TAILP := START;
  WHILE TRAVELP <> ENDCP.SEQP DO
    BEGIN
      WITH TRAVELP DO
      BEGIN
        IF (JMPP <> NIL) AND (TAILP.KEY = K_SBR) THEN
          BEGIN
            IF JMPP.BAKP.SEQP.KEY = K_LBL THEN
              JMPP := JMPP.BAKP.SEQP.SEQP.KEY
            ELSE
              PUTGOSUBLBL (JMPP.KEY, JMPP);
            END;
            TAILP.ADDR := -1;
            JMPP := NIL
          END;
        END;
        TAILP := TRAVELP;
        TRAVELP := TRAVELP.SEQP
      END
    END;
END; (* FINDGOSUBLBL *)

(* CODE RESOLUTION ROUTINES *)

(* PUTGOSUBLBL USES PROCEDURE INSERTKEY TO ENTER THE LBL *)
(* REFERENCED BY THE GOSUB CALL INTO THE CODE SEQUENCE AT *)
(* LOCATION POINTED TO BY TO BY THE JMPP (LBLP). *)

(* FINDGOSUBLBL SEARCHES THE CODE DATA STRUCTURE TO FIND *)
(* SBR CALLS FOR WHICH THE JMPP HAS BEEN SET; THESE WILL *)
(* CORRESPOND TO WHIS RELATED GOSUB STATEMENTS; THE JMPP IS *)
(* CALLLED AND THE CORRECT LABEL IS INSERTED INTO THE *)
(* CODE SEQUENCE USING THE BAKP AND PROCEDURE PUTGOSUBLBL *)

(* ----------------------------------------------- ************
(*) PUTGOSUBLBL *)

(* ----------------------------------------------- ************
(*) FINDGOSUBLBL *)
(* 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--*)
(* COUNTREP COUNTS THE NUMBER OF SEQUENTIAL OCCURRENCES OF *)
(* KEYC AT A PARTICULAR LOCATION IN THE CODE DATA STRUCT *)
(* STRUCTURE: NOTE THAT IT ALSO CHECKS FOR JUMP POINTERS *)
(* TO THESE KEYS. *)

FUNCTION COUNTREP (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;

PROCEDURE NUMBERUN (VAR MOVEP : CODEPTR);
BEGIN WHILE (MOVEP.KEY IN NUMERICKEY) AND (MOVEP.ADDR = -1) DO BEGIN MOVEP := MOVEP.SEQP END;
(* 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 MOVEP <> NIL DO
    BEGIN
      IF (MOVEP@.KEY = K_OPAAREN) AND (MOVEP@.ADDR = -1) THEN
        BEGIN
          OPEN := TAILP;
          OPENCT := COUNTPRE (MOVEP, K_OPAAREN);
          IF (MOVEP@.KEY = K_BCL) AND (MOVEP@.ADDR = -1) THEN
            BEGIN
              MOVEP := MOVEP@.SEQP;
              CLOSE := MOVEP@.SEQP@.SEQP;
              MOVEP := MOVEP@.SEQP@.SEQP;
              CLOSECT := COUNTPRE (MOVEP, K_CPAREN);
              IF CLOSECT > 0 THEN
                (* IF EXTRAS, DELETE *)
                REMOVEPAREN (OPEN, CLOSE, OPENCT, CLOSECT)
            END
          ELSE IF (MOVEP@.KEY IN NUMERICKEY) AND
                  (MOVEP@.ADDR = -1) THEN
            BEGIN
              WHILE MOVEP@.SEQP@.KEY IN NUMERICKEY DO
                (* PASS OVER NUMBER *)
                MOVEP := MOVEP@.SEQP;
                CLOSE := MOVEP@.SEQP@.SEQP;
                MOVEP := MOVEP@.SEQP;
                CLOSECT := COUNTPRE (MOVEP, K_CPAREN);
                IF CLOSECT > 0 THEN
                  (* IF EXTRAS, DELETE *)
                  REMOVEPAREN (OPEN, CLOSE, OPENCT, CLOSECT)
              END
            END
          END
        END
      END
    END;
  END
END;

(* OSQPAREN *)

(*****************************************************************)
(* OSQCF (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 OSQCF (VAR START : CODEPTR);
VAR CUR : CODEPTR;
   INDEX : CTXTNRNG;
BEGIN
   CUR := START;
   WHILE CUR <> NIL DC
      BEGIN
         INDEX := CUR.KEY;
         FOR I := 1 TO (CTXT(INDEX).UNIT) DO
            IF (CUR.SEQP.KEY = K_NOP) AND (CUR.SEQP.ADDR = -1)
            THEN CUR.SEQP := CUR.SEQP.SEQP (* REMOVE NOP *)
            ELSE CUR := CUR.SEQP
               (* NEXT NODE *)
         END;
      END; (* SQUEEZE OUT NOPS *)
   END;
   CUR := CUR.SEQP
      (* CSQFOP *)
(* *********************************************** *)
(*-82--- ADDR FILLS THE ADDR FIELDS OF ALL TI-59 CODE *)
(* 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;

BEGIN (* RESOLVE_ADDR MAIN *)
   TRAVEL := START;
   I := 0;
   WHILE TRAVEL <> ENDCP@.SEQP DO (* INSERT ABSOLUTE ADDR *)
      BEGIN
         TRAVEL@.ADDR := I;
         TRAVEL := TRAVEL@.SEQP;
         I := I + 1
      END;
   TRAVEL := START;
   WHILE TRAVEL <> ENDCP@.SEQP DO (* FIND/JUSTIFY JMP ADDR *)
      BEGIN
         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;
   (* RESOLVE_ADDR *)

169
OUTPUT DUMP ROUTINES

PROEDURE FINDMSG (VAR MSGFILE : TEXT; VAR ESCCHAR : CHAR; MSG_NO : INTEGER);

VAR CH : CHAR;
   I : INTEGER;

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

PROCEDURE WRITELN; (* FWD DECL WITH I/O COMMAND ROUTINES *)

VAR CH, ESCCHAR : CHAR;
   I : INTEGER;

BEGIN
   FINDMSG (MSGFILE, ESCCHAR, MSG_NO);
   REPEAT
      READ (MSGFILE, CH);
      IF CH = ESCCHAR THEN (* CHECK FOR EMBEDDED ESCAPE CHAR *)
         READLN (MSGFILE, I) (* AND DISCARD IF FOUND. *)
      ELSE
         WRITE (WFILE, CH);
         IF EOLN (MSGFILE) THEN
            BEGIN
               READLN (MSGFILE);
               WRITELN (WFILE)
            END
      UNTIL (EOF (MSGFILE)) OR ((CH = ESCCHAR) AND (I = MSG_NO))
   END; (* WRITELN *)

(*******************************************************************)
PROCEDURE WRIT (VAR WFILE, MSGFILE : TEXT;
VAR CH, ESCHAR : CHAR;
VAR MSG_NO : INTEGER);
BEGIN
FINDMSG (MSGFILE, ESCHAR, MSG_NO);
REPEAT
READ (MSGFILE, CH);
WRITE (WFILE, CH)
UNTIL ECLN (MSGFILE)
END;
(* WRIT *)

PROCEDURE REPORT (VAR WFILE : TEXT);
VAR LTCTAI : LBLRNG;
RTCTAI : INTEGER;
BEGIN
WRITLN (WFILE, MSGF, 3);* WRITELN (WFILE, ERRRCT: 7, ' FATAL ERRORS.');
WRITLN (WFILE, WARENC: 7, ' WARNING MSGS.');
IF ERRRCT > 0 THEN (* CALCULATIONS INCOMPLETE *)
WRITLN (WFILE, MSGF, 14);
RTCTAI := NEXTREG - STARTREG;
LTCTAI := LBLCT - 1;
WRITLN (WFILE, MSGF, 4)
END;
(* REPORT *)

(* REPORT COMPUTES AND WRITES THE REGISTER/LABEL SUMMARY.*)
(* 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 CCDEDUMP (VAR WFILE : TEXT; VAR TICODE : CODEPTR);
VAR CUR, HCLD : CODEPTR;
I : 0..3;

(* CODEDUMP MAIN *)
BEGIN
CUR := TICODE:
WRITELN (WFILE, MSGF, 5); (* HEADER MSG *)
WRITELN (WFILE; ' $'); (* "$" MUST BE WRITTEN HERE *)
WRITELN (WFILE, MSGF, 6); (* ELSE WILL INTERFERE W/ WRITELN *)
WHILE CUR@.SEQP <> NIL DO
BEGIN
CCDEDUMP (CUR);
WRITELN (WFILE, CTEXT (.CUR@.KEY.), CODECHAR);
IF CUR@.KEY IN (.A SBH, K LBL.) THEN
BEGIN
(* MUST NOT TAKE SBR'S OR LBL'S LITERALLY *)
CUR := CUR@.SEQP;
CCDEDUMP (CUR);
WRITELN (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@.SEQP;
CCDEDUMP (CUR);
ZEROPAD (WFILE, CUR@.KEY, 2);
WRITELN (WFILE)
END
CUR := CUR@.SEQP
END
END
(* CODEDUMP *)

*****************************************************************************

172
(*-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, 'S', MSGNO: 1); (* WRITE MSG DELIMITER *)
WHILE NOT EOF (RFILE) DO
BEGIN
WHILE NOT EOLN (RFILE) DO
BEGIN
READ (RFILE, CH);
WRITE (WFILE, CH);
END;
WRITE (WFILE);
IF NOT EOF (RFILE) THEN
READLN (RFILE);
END;
WRITE (WFILE, '$', MSGNO: 1); (* WRITE MSG DELIMITER *)
WRITE (WFILE);
(* LOGTO *)
(*------------------------------------------------------------------*)
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 NOT FIRSTREAD THEN
(* DATA/READ MAP = MSG $4 *)
BEGIN
RESET (READF, 'NAME=READF.WBASIC.A');
LOGTO (SCRATCH, READF, 4)
END
END (* LINK_INTERFACE *)

(*************************************************************************)
(* SYMTBLDUMP IS A SPECIAL PURPOSE ROUTINE USED FOR *) (* DEBUGGING: IT WILL LUMP THE ENTIRE CONTENTS OF THE *) (* COMPILER SYMBOL TABLE BUCKET BY BUCKET; THIS ROUTINE *) (* IS TOGGLED USING OPTION NUMBER 5. *)

PROCEDURE SYMTBLDUMP (VAR WFILE : TEXT; BUCKET : HASH);

VAR I : INTEGER;
LOCK : SLOTPTR;
BEGIN
  WRITELN (WFILE, MSGF, 10); (* HEADER MSG *)
  FOR I := 0 TO HASHBASE DO (* SKIP EMPTY BUCKETS *)
    BEGIN
      ZEROPAD (WFILE, I, 2); (* BUCKET BOUNDARY MSG *)
      LOCK := BUCKET(i); (* UNTIL LOCK = NIL *)
      REPEAT (* CASE TYP OF *)
        VARID := BEGIN
          IF REGNO < 0 THEN (* PI = -314 *)
            ELSE
              BEGIN
                ZEROPAD (WFILE, REGNO, 2); (* IF USED *)
                IF AUXREG1<>-1 THEN (* IF USED *)
                  BEGIN
                    WRITE (WFILE, '::32); (* IF USED *)
                    ZEROPAD (WFILE, AUXREG1, 2); (* IF USED *)
                  END;
                  IF AUXREG2<>-1 THEN (* IF USED *)
                    BEGIN
                      WRITE (WFILE, '::32); (* IF USED *)
                      ZEROPAD (WFILE, AUXREG2, 2); (* IF USED *)
                    END
                END
            END
          IF FNQID<>-1 THEN (* IF USED *)
            BEGIN
              WRITE (WFILE, '..QUICK FN'); (* IF USED *)
              ZEROPAD (WFILE, FNLRSG, 2); (* IF USED *)
              WRITE (WFILE, 'LONG FN'); (* IF USED *)
            END;
          IF FNPID<>-1 THEN (* IF USED *)
            BEGIN
              WRITE (WFILE, 'PARAMETER FN'); (* IF USED *)
              ZEROPAD (WFILE, FNREGNO, 2); (* IF USED *)
            END
        END
      END;
      WRITELN (WFILE, MSGF, 12) (* SYMTBLDUMP *)
  END;
END;

(*******************************************************************)

174
PROCEDURE SEARCH (VAR WFILE : TEXT; LSTART : LINEPTR);
VAR LPSEARCH : LINEPTR; CODP : CODEPTR;
BEGIN
  WRITE (WFILE, MSGF, 13); (* HEADER MSG *)
  LPSEARCH := LSTART;
  REPEAT
    WRITE (WFILE, 'LINUM = ');
    ZEROPAD (WFILE, LPSEARCH@.LINO, 5); (* WBASEC LINE NO *)
    WRITE (WFILE);
    CODP := LPSEARCH@.CPTR;
    REPEAT
      WRITE (WFILE, ' :2);
      ZEROPAD (WFILE, CODP@.ADDR, 3);
      WRITE (WFILE, ' :2);
      WRITE (WFILE, CODP@.KEY, 2);
      WRITE (WFILE);
      CODP := CODP@.DONE
    UNTIL (CODP = LPSEARCH@.LPTR@.CPTR) OR (CODP = NIL);
    LPSEARCH := LPSEARCH@.LPTR
  UNTIL LPSEARCH@.LINO = MAXBASLIN (* MAXBASLIN IS END *)
END;
(* SEARCH *)
INITIALIZATION ROUTINE


PROCEDURE INITIALIZE;
 VAR I : INTEGER;

/* 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, RWWORDCT, 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..RWWORDCT + 1;
    LINX, LENG : 0..RWLENGCT + 1;
    CH : CHAR;
 BEGIN
    LINX := 0; (* INIT LENGTH INDEX *)
    CHINX := 1; (* INIT CHAR INDEX *)
    WHILE NOT EOF(RWTBLF) DO
    BEGIN
      STARTCHINX := CHINX;
      READ (RWTBLF, WINX); (* READ WORD INDEX (INTEGER) *)
      READ (RWTBLF, CH, CH); (* READ OFF 2 BLANK SPACES *)
      RWWORD (.WINX) := CHINX;
      REPEAT (* READ CHARs OF ONE WORD INTO CHAR ARRAY *)
      BEGIN
        READ (RWTBLF, RWCHAR (.CHINX));
        CHINX := CHINX + 1
      UNTIL EOFL(RWTBLF);
      REALLN (RWTBLF);
      LENG := CHINX - STARTCHINX;
      IF LENG > LINX THEN
      BEGIN
        LINX := LENG; (* INDEX ITS LOCATION IN *)
        RWLENG (.LINX) := WINX
      END;
      RWCHAR (.RWCHARCT + 1.) := BLANK (* SET DELIMITERS FOR *)
      RWWORD (.RWWORDCT + 1.) := NEWCHARCT + 1; (* ARRAYS AND *)
      RWLENG (.RWLENGCT + 1.) := RWWORDCT + 1 (* INDICES *)
    END;

(*)
PROCEDURE LCADLIE (VAR LIBFILE : TEXT; FNTYPE : IDTYF; SEQLEN : INTEGER);
VAR IDSICT : SLCTPTR;
I : INTEGER;

BEGIN
READLN (LIBFILE); READLN (LIBFILE); (* SKIP HEAD LINES *)
WHILE NOT EOF (LIBFILE) DO
BEGIN
ACCINX := 0;
(* INIT ACCUM INDEX *)
REPEAT
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 SYMTBL *)
IDSLOT.TYP := FNTYPE;
(* SET IDENT TYPE *)
FOR I := 1 TO SEQLEN DO
(* READ KEY CODES *)
CASE FNTYPE CP
FCF I := ACCINX TO MAXTOKLEN DO
(* READ KEY CODES *)
END;
END;
(* CASE *)
READLN (LIBFILE) (* SKIP TO NEXT LN *)
END;
(* LOADLIB *)

(* LCADCTEXT *)
PROCEDURE LCADCTEXT;
VAR I, K : INTEGER;
J, K : 1..TEXTLEN + 1;
CH : CHAR;

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

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

TERMOUT (OUTFILE);
RESET (BASICF, '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 (EINFNF, 'NAME=EINFNF.PASCAL.A' );
REWRITE (LISTF, 'NAME=LISTF.WBASIC.A' );
REWRITE (NAMEF, 'NAME=NAMEF.WBASIC.A' );
WRITELN (LISTF, MSGF, 2); (* HEADER MSG TO LISTF *)
WRITELN (OUTFILE, MSGF, 1); (* TERMINAL INVOKE MSG *)
WRITELN (NAMEF, MSGF, 6); (* HEADER MSG TO NAMEF *)

(* INITIALIZE OPTION TOGGLES *)

LINK59 := FALSE; (* OPTION 0 *)
PC100 := TRUE; (* OPTION 1 *)
OPTFAR := TRUE; (* OPTION 2 *)
OPTNOF := TRUE; (* OPTION 3 *)
CODUMP := TRUE; (* OPTION 4 *)
SYDUMP := FALSE; (* OPTION 5 *)
DSDUMP := FALSE; (* OPTION 6 *)
TOKCUT := FALSE; (* OPTION 7 *)
TOKIS := FALSE; (* OPTION 8 *)

(* INITIALIZE RESERVED WORD ARRAY INDEXES. *)

LCADRW (PVI TBLF );

(* INITIALIZE CHARACTER SETS. *)

LETTERS := ( 'A'..'I' ) + ( 'J'..'R' ) + ( 'S'..'Z' );
DIGITS := ( '0'..'9' );
ALFANUM := LETTERS + DIGITS;
SIGNS := ( '+-' ) + ( '->' ) ;
DOUBLE1 := ( '>' ) + ( 'L' ) + ( 'T' ) ;
DOUBLE2 := ( '<' ) + ( 'G' ) + ( 'T' ) + ( 'L' ) ;
SPECIALS := ( '.' ) + ( ';' ) + ( '!' ) + ( '(' ) + ( ')' ) + ( ',' );
SUBERRCR := ( 'E' ) + ( 'F' ) + ( 'S' );
CRITICAL := ( ENDLIN ) + ( ENDFIL ) + ( 'E' );
TAILICKS := ( CMICKEXC ) + ( ENDLINP ) + ( ENDLINTOK ) ;
BINCTICKS := ( PLUSTOK ) + ( MINUSTOK ) + ( MULTOK ) ;
RELCTICKS := ( EQUALTOK ) + ( NOTEQ TOK ) + ( GTEQ TOK ) ;
NUMERICKEY := ( K_INT ) + ( K_INT ) + ( K_SI G );
SIGTOKS := ( PLUSTOK ) + ( MINUSTOK ) ;
BEGIN_EXPTOKS := SIGTOKS + ( IDENTOK ) + ( OPAREN TOK ) ;
* INITIALIZE HASH TABLE AND REGISTER COUNT. *

FOR I := 0 TO HASHBASE DO
  BUCKET (.I.) := NIL;
  NEXTREG := STARTREG;

* INITIALIZE ARRAY HOLDING OUTPUT TEXT OF TI-59 CODE. *

LOADTEXT;

* INITIALIZE BUILT-IN FUNCTION LIBRARY. *

LOADLIB {EIPRF, FNID, FNLEN};
LOADLIB {EIPNL, FNII, FNLEN};

* ENTER 'PI' = 3.14159265359 IN SYMBOL TABLE. *

ACCUM (.1.) := 'P'; ACCUM (.2.) := 'I';
ACCINX := 3 TO MAXTCKLEN DC
ACCUM (.1.) := BLANK;
IDSLOT := GETSLOT (ACCUM, ACCINX);
IDSLOT2.TYP := VARID;
IDSLOT2.REGNO := -314; (* SPECIAL REGNO FOR 'PI' *)

* INITIALIZE LABEL STACK (ARRAY OF INTEGER KEY CODES). *

READIN (LABELF);
FOR I := 1 TO LBBLBASE DO
  READ (LABELF, CLAEEL (.I.));
  LBLCT := 1;

* INITIALIZE RESERVED REGISTER SET. *

READIN (LABELF), READIN (LABELF), READLN (LABELF),
RESERVEST := 0;
RESERVE REG := (.1.); (* INITIALIZE TO EMPTY SET *)
WHILE NOT EOP (LABELF) DO
  BEGIN
    WHILE NOT EOLN (LABELF) DO
      BEGIN
        READ (LABELF);
        RESERVE REG := RESERVE REG + (.1.); (* MAKE SET *)
        RESERVEST := RESERVEST + 1; (* COUNT MEMBERS *)
      END;
    READIN (LABELF)
  END;

* INITIALIZE PNF ACTIVATION STACK AND PNL USE LIST. *

PNSSTACKCT := 0;
PNSSTACK := NIL;
PNSLLIST := NIL;
(* INITIALIZE FIRST CALL TO SCAN. *)

LINEUF (.0.) := BLANK;
LINEUF (.1.) := ENDLIN;
TOKNUM := ENDINTCK;
LNFINX := 0;
ERRORC := 0;
WARNCT := 0;
FLAGCMT := FALSE;
LNUM := 0;
LLNUM := 0;
CLIBNUM := 0;

(* INITIALIZE LINKED DATA STRUCTURE FOR TI-59 CODE. *)

FIRSTLP := GETNEWHER (LNUM); (* SET A COMMON REF NODE *)
LPCUR := FIRSTLP; (* ANCHOR ALL MARKER PTRS TO IT *)
LASTLP := FIRSTLP;
ENDCP := FIRSTLP$CPTR;
BEGINCF := ENDCP;
CPCUR := ENDCP;
SETLINE (LPCUR, LP); (* SET UP FOR MAIN PROCEDURE LABEL *)
GENKEY (KLBL);
GENKEY (NEWLBL); (* MAIN PROCEDURE = LBL A *)
BEGINCF := BEGINCP$SEQP; (* BYPASS THE HEADER NODE *)

(* INITIALIZE LOOP/BRANCH STACKS. *)

IFSTACK := NIL;
ENDIFSTACK := NIL;
ENDLOCPSTACK := NIL;
FORSTACK := NIL;
NEXTSTACK := NIL;

(* INITIALIZE READ/DATA STATEMENT INDEXES/FLAGS. *)

READIX := 1;
DATAIX := 1;
INDEXERROR := FALSE;
FIRSTREAD := TRUE

END; (* INITIALIZE *)
BEGIN (* EAX59 MAIN *)

INITIALIZE;
REPEAT (* UNTIL TCKNUM = ENDFILTOK *)

SCAN (TCKNUM);
IF ERRORCT = 0 THEN (* SCAN FIRST WORD OF NEW LINE *)
BEGIN (* FIRST FATAL ERROR ENCOUNTERED *)
SETLINE (LPCUR, LP); (* NEW WBASIC LINE NO & LINE *)
CASE TCKNUM CF (* RECURSIVE DESCENT PARSE PRC *)

--------------------

* 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 IMPLEMENTED IN THIS SUBSET *
--------------------

ERRORTOK : BEGIN END;
CMTOKEXC : PREM;
7,3,4,5,6 : PERROR;
7,8,9,10,11 : PERROR;
12,13,14 : BEGIN END;
15,16,17,18 : PERROR;
19 : PIF:
20 : PERROR;
22 : PERROR :
23 : PERROR;
24 : PERROR :
25 : PERROR;
26 : PERROR;
27 : PERROR;
28 : PERROR;
29 : PERROR;
30 : PERROR:
31,32 : PERROR:
33 : PERROR :
34 : PERROR : PSUBERROR;
35 : PERROR : PSUBERROR;
36 : PERROR : PSUBERROR;
37 : PERROR : PSUBERROR;
38 : PERROR : PSUBERROR:
39 : PERROR : PSUBERROR;
40 : PERROR : PSUBERROR;
41 : PERROR : PSUBERROR:
42 : PERROR : PSUBERROR:
43,44,45 : PERROR:
46 : PERROR : PSUBERROR;
47 : PERROR : PSUBERROR;
48 : PERROR : PSUBERROR;
49,50 : PERROR :
51 : PERROR : PSUBERROR:
52 : PERROR : PSUBERROR:
53 : PERROR : PSUBERROR:
54 : PERROR : PSUBERROR:
55,56,57,58 : PSUBERROR:

--------------------

* IF ** *
* OR *** *
* ON ** *
* LET ** *
* REM ** *
* FOR ** *
* END ** *
* DEP ** *
* NOT ** *
* DIM ** *
* AND *** *
* ** *
* ** *
* ** *
* ** *
* ** *
* ** *
* ** *
* ** *
* ** *
* ** *
* ** *
* ** *
* ** *
* ** *
* ** *
* ** *
* ** *
* ** *
* ** *
* ** *
* ** *
* ** *
* ** *
* ** *
UNTIL TCKNUM = ENDFILTOK;
GETFNLS:
ENDCP@.SEC := NEWCCDE (-1); (* INSERT LONG FN BODIES *)
LPCUR@.IFTR := GETNEWHDR (MAXBASLIN); (* CLOSE LINE SEQ *)
FINEGOSUBLEL (BEGINCP); (* INSERT LABELS FOR SER *)
IF OPTPAR THEN (* OPTION 2 *)
OSQPAREN (BEGINCF); (* OPTIMIZE PARENTHESES *)
IF CETNCP THEN (* OPTION 3 *)
CSCNCF (BEGINCF); (* OPTIMIZE (OUT) NOP'S *)
RESOLVE_ADDS (BEGINCP); (* OVERLAY ABSOLUTE ADDR *)
REPORT (LISTF); (* ERROR/REG/LBL SUMMARY *)
REPORT (OUTFILE); (* OPTION 4 *)
CODELUMP (LISTF, BEGINCF); (* WRITE TRANSLATED CODE *)
IF LINK59 THEN (* OPTION 5 *)
LINK_INTERFACE; (* CREATE SCRATCH FILE *)
END; (* EAX59 *)

END (* CASE *)

(* DEBUUING TOOL: DUMPS EACH SLOT OF EACH BUCKET IN THE *)
(* SYMBOL TABLE TO NAME FILE. *)
(* SYDUMP THEN *)
SYMDUMP (LISTF, BUCKET);

(* DEBUUING TOOL: DUMPS ENTIRE CODE DATA STRUCTURE *)
(* INCLUDING LINE AND CODE NODES. *)
(* LSDUMP THEN *)
SEARCH (LISTF, FIRSTLP)

END;

(************************************************************************)

(************************************************************************)
APPENDIX D

BWTBLF FILE--ORDERED RESERVED WORDS

! + - * / < > = * * * < > =  
< = > =  
= =  
FOR NEXT ENDGUESS ELSEIF ELSE  
INPUT USE THEN  
WHILE ENDU  
PRINT  
CHAIN  
ALMIT  
CLOSE  
 
614.4x793.9
60 RETURN
61 CPTICN
62 INPUT
63 REMCVE
64 RENAME
65 RESUME
66 UNLOCK
67 ENDOOP
68 RESTORE
69 SCRATCH
70 TAGSORT
71 ENDCUESS
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 26 27 28 29 30 32
33 34 35 36 37 38 39 40 42 43 44
45 47 48 49 50 52 53 54 55 57
58 59 60 61 62 63 64 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

**BIFNQF/BIFNLF FILES—BUILT-IN FUNCTIONS**

**BUILT-IN "QUICK" FUNCTION NAMES AND TI-59 KEY CODES:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Code 1</th>
<th>Code 2</th>
<th>Code 3</th>
<th>Code 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>50</td>
<td>68</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>ACOS</td>
<td>27</td>
<td>39</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>ASIN</td>
<td>27</td>
<td>38</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>ATN</td>
<td>27</td>
<td>30</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>COS</td>
<td>39</td>
<td>68</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>COT</td>
<td>30</td>
<td>35</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>CSC</td>
<td>38</td>
<td>35</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>EXP</td>
<td>32</td>
<td>23</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>FP</td>
<td>27</td>
<td>59</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>IP</td>
<td>59</td>
<td>68</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>LOG</td>
<td>23</td>
<td>68</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>LOG10</td>
<td>28</td>
<td>68</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>SEC</td>
<td>39</td>
<td>35</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>SIN</td>
<td>38</td>
<td>68</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>SQR</td>
<td>34</td>
<td>68</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>TAN</td>
<td>30</td>
<td>68</td>
<td>68</td>
<td>68</td>
</tr>
</tbody>
</table>

**BUILT-IN "LCNG" FUNCTION NAMES AND TI-59 KEY CODE SEQUENCES:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Code 1</th>
<th>Code 2</th>
<th>Code 3</th>
<th>Code 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RND</td>
<td>36</td>
<td>15</td>
<td>10</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>36</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>71</td>
<td>88</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>68</td>
<td>68</td>
<td>68</td>
<td>68</td>
</tr>
</tbody>
</table>
## APPENDIX G

**CTEXTF FILE—TI-59 KEYCODE TRANSLATIONS**

<table>
<thead>
<tr>
<th>TI-59 KEY CODE TEXT</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-2 0</td>
<td>UNRESOLVED ADDR $$$$</td>
</tr>
<tr>
<td>00 0</td>
<td>UNPILLED CODE $$$$</td>
</tr>
<tr>
<td>01 0</td>
<td>1</td>
</tr>
<tr>
<td>02 0</td>
<td>2</td>
</tr>
<tr>
<td>03 0</td>
<td>3</td>
</tr>
<tr>
<td>04 0</td>
<td>4</td>
</tr>
<tr>
<td>05 0</td>
<td>5</td>
</tr>
<tr>
<td>06 0</td>
<td>6</td>
</tr>
<tr>
<td>07 0</td>
<td>7</td>
</tr>
<tr>
<td>08 0</td>
<td>8</td>
</tr>
<tr>
<td>09 0</td>
<td>9</td>
</tr>
<tr>
<td>10 0</td>
<td>2NL E</td>
</tr>
<tr>
<td>11 0</td>
<td>A</td>
</tr>
<tr>
<td>12 0</td>
<td>C</td>
</tr>
<tr>
<td>13 0</td>
<td>D</td>
</tr>
<tr>
<td>14 0</td>
<td>E</td>
</tr>
<tr>
<td>15 0</td>
<td>F</td>
</tr>
<tr>
<td>16 0</td>
<td>G</td>
</tr>
<tr>
<td>17 0</td>
<td>H</td>
</tr>
<tr>
<td>18 0</td>
<td>I</td>
</tr>
<tr>
<td>19 0</td>
<td>J</td>
</tr>
<tr>
<td>20 0</td>
<td>K</td>
</tr>
<tr>
<td>21 0</td>
<td>L</td>
</tr>
<tr>
<td>22 0</td>
<td>M</td>
</tr>
<tr>
<td>23 0</td>
<td>N</td>
</tr>
<tr>
<td>24 0</td>
<td>O</td>
</tr>
<tr>
<td>25 0</td>
<td>P</td>
</tr>
<tr>
<td>26 0</td>
<td>Q</td>
</tr>
<tr>
<td>27 0</td>
<td>R</td>
</tr>
<tr>
<td>28 0</td>
<td>S</td>
</tr>
<tr>
<td>29 0</td>
<td>T</td>
</tr>
<tr>
<td>30 0</td>
<td>U</td>
</tr>
<tr>
<td>31 0</td>
<td>V</td>
</tr>
<tr>
<td>32 0</td>
<td>W</td>
</tr>
<tr>
<td>33 0</td>
<td>X</td>
</tr>
<tr>
<td>34 0</td>
<td>Y</td>
</tr>
<tr>
<td>35 0</td>
<td>Z</td>
</tr>
<tr>
<td>36 0</td>
<td>0</td>
</tr>
<tr>
<td>37 0</td>
<td>1</td>
</tr>
<tr>
<td>38 0</td>
<td>2</td>
</tr>
<tr>
<td>39 0</td>
<td>3</td>
</tr>
<tr>
<td>40 0</td>
<td>4</td>
</tr>
<tr>
<td>41 0</td>
<td>5</td>
</tr>
<tr>
<td>42 0</td>
<td>6</td>
</tr>
<tr>
<td>43 0</td>
<td>7</td>
</tr>
<tr>
<td>44 0</td>
<td>8</td>
</tr>
<tr>
<td>45 0</td>
<td>9</td>
</tr>
<tr>
<td>46 0</td>
<td>**X **</td>
</tr>
<tr>
<td>47 0</td>
<td>**X **</td>
</tr>
<tr>
<td>48 0</td>
<td>**X **</td>
</tr>
<tr>
<td>49 0</td>
<td>**X **</td>
</tr>
<tr>
<td>50 0</td>
<td>**X **</td>
</tr>
<tr>
<td>51 0</td>
<td>**X **</td>
</tr>
<tr>
<td>52 0</td>
<td>**X **</td>
</tr>
<tr>
<td>53 0</td>
<td>**X **</td>
</tr>
<tr>
<td>54 0</td>
<td>**X **</td>
</tr>
</tbody>
</table>

187
55 0 / $EL $$$ $ERROR $$$$  
56 0  $  
57 0  2ND $FG  
58 0  2ND $FX  
59 0  2ND $FG  
60 0  2ND $DX  
61 1  2ND $FG  
62 1  2ND $FX  
63 1  2ND $PD  
64 1  2ND $PD  
65 0  2ND $PAUSE  
66 0  2ND $X=  
67 0  2ND $CF  
68 1  2ND $CE  
69 1  2ND $CE  
70 2  2ND $CE  
71 2  2ND $CE  
72 2  2ND $CE  
73 2  2ND $CE  
74 2  2ND $CE  
75 0  2ND $ISL  
76 0  2ND $X=  
77 0  2ND $SUMMATION  
78 0  2ND $SUMMATION  
79 0  2ND $SUMMATION  
80 0  2ND $SUMMATION  
81 0  2ND $SUMMATION  
82 0  2ND $SUMMATION  
83 0  2ND $SUMMATION  
84 0  2ND $SUMMATION  
85 0  2ND $SUMMATION  
86 0  2ND $STFLG  
87 0  2ND $STFLG  
88 0  2ND $STFLG  
89 0  2ND $STFLG  
90 0  2ND $STFLG  
91 0  2ND $STFLG  
92 0  2ND $STFLG  
93 0  2ND $STFLG  
94 0  2ND $STFLG  
95 0  2ND $STFLG  
96 0  2ND $STFLG  
97 0  2ND $STFLG  
98 0  2ND $STFLG  
99 0  2ND $STFLG
APPENDIX H

MSGP FILE--CROSS-COMPILER OUTPUT MESSAGES

$ IS THE ESCAPE CHAR (MSG DELIMITER) FOR THIS MESSAGE FILE.
$1 $$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1$$$$$$1

INVOKING WBSASIC CROSS-COMPILER FOR TI-59 PC
EAX59 VERSION 1.0

$1

WBSASIC PROGRAM LISTING

$2

COMPILATION SUMMARY

$3

COMPILATION TERMINATES

$4

TI-59 CODE TRANSLATED FROM WBSASIC
(UNSEGMENTED)

$5

ADDR CODE

$6

BEGIN TI-59 CODE

$7

2ND TI-59 CODE

$8

TI-59 REGISTER TO NAME MAPPING

REG# BASIC NAME

$9

INPUT DATA TO READ MAPPING

DATA REG NAME

189
<table>
<thead>
<tr>
<th>BUCKET CONTENTS</th>
<th>REG</th>
<th>TYP</th>
</tr>
</thead>
<tbody>
<tr>
<td>$19$</td>
<td>$10$</td>
<td>$10$</td>
</tr>
<tr>
<td>$11$</td>
<td>$11$</td>
<td>$11$</td>
</tr>
<tr>
<td>$12$</td>
<td>$12$</td>
<td>$12$</td>
</tr>
<tr>
<td>$13$</td>
<td>$13$</td>
<td>$13$</td>
</tr>
</tbody>
</table>

Before the first fatal error, these stats were computed:

| $14$ |
| $14$ |
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:
SCRATC: SCRATCH PASCAL
OUTFILE:"ANY DESIRED NAME"
YOUR OUTPUT FILE
TEMPFILE:"ANY DESIRED NAME"
A TEMPORARY SCRATCH PAD
MESSAGEFILE:"MESSAGEFILE FILE"
LINKER'S MESSAGES

***************
PROGRAM TSERIVER (INPUT,OUTPUT);

***************
DECLARATIONS:
***************

(* CONSTANTS:)
(*---.....---...---...---...*)

CONST
FJUMPCONST = 10; (* NUM STEPS FOR F JUMP CODE *)
SERCCONST = 15; (* NUM STEPS FOR SBR BRK CCDZ *)
SERCCSTCONST = 7; (* NUM STEPS FOR SBR BRK RTN *)
STO = 42;
LEL = 76;
DEOCLINE = 73;
STOIND = 72;
CE = 69;
DECIMAL = 93;
RS = 91;
CE = 24;
(* TI59 KEYCODES *)

(* END KEYCODES *)

DISPLAYREGSTORE = 00; (* TEMP STORE OF THE DISPLAY *)
RTNREGNUM = 6; (* NUMBER OF MANUAL RETURN REGISTERS *)
MANRTRGREG = 08; (* MANUAL SBR RETURN REGISTER *)

NCNF = 101; (* MESSAGE NUNTS *)
ASTR = 102;
YES = 103;
MCDEFCRETS = 100;
RTNRTGTOP = 104;
CCDENUM = 9;
STOINREG = 105;
EGMRTSTIS = 106;
PRTNUMS = 107;
MCDF = 108;
CARD1 = 109;
CARD2 = 110;
SIDE1 = 111;
SIDE2 = 112;
EXA = 81;
EXBI = 82;
EXLI = 83;
EXAL = 84;
UNSEC = 5;
SEQ = 6;
PAK = 7;
EDW = 8;
PSR = 9;
REAL = 4;
GFCT = 1;
ALPHBL = 99;

(* END MESSAGE NUMS *)

TYPE

(* END TYPE *)

LABELS = PACKED ARRAY (1..15) OF CHAR;
TYPELABELS = ARRAY (.0..99.) OF LABELS; (* TI59 KEYS *)

(*) SHOULD HAVE MADE A VARI *)

INSTR_SET = SET CF 0 .. 99 ; (* RANGE INSTRUCTION SET*)

NCDE = (TABLE,SBRPTR,SBRBREAK,FWD_JUMP,MEMODULE,CODE);
INCLPTR = @NODES;
NCDES = RECORD
CASE TAG: NODE OF
TABLE: (NEST:INTEGER;
START_ADDR:INTEGER;
STOP_ADDR:INTEGER;
LENGTH:INTEGER;
INCLUDED:BOOLEAN;
COALESCED: BOOLEAN;
SBRLIST:TBLPTR;
NUMF:INTEGER;
JUMP_LIST:TBLPTR;
TABLE_LIST:TBLPTR) ; (* NEXT TABLE *)

SBRPTR: (SBR:TBLPTR;
FROM:INTEGER;
NEXT_SBR:TBLPTR) ; (* NEXT PTR *)

MEMODULE: (MEMNUM:INTEGER;
OFFSET:INTEGER;
HIGHTOFFSET:INTEGER;
LOWOFFSET:INTEGER;
RETURNCODE NEEDED:BOOLEAN;
SEGtbls:TBLPTR;
CODELIST:TBLPTR;
NEXT:TBLPTR) ;

192
CODE : (ADDRESS : INTEGER;
       ADDR : INTEGER;
       KEYCODE : INTEGER;
       SEQUENTIAL : TBLPTR);

SBREREAK : (SBRZ : TBLPTR);

PFDJUMP : (JUMP_ADDR : INTEGER;
            JUMP_ADDR0 : INTEGER;
            JUMP_ADDR01 : INTEGER; (* HUNDREDS*)
            JUMP_ADDR02 : INTEGER; (* TEN/UNIT*)
            ADDR : INTEGER;
            MEM_ADDR : INTEGER; (* MEMNUM*)
            JUMP_INTADDR01 : INTEGER; (* LOCAL*)
            JUMP_INTADDR02 : INTEGER;
            NEXT_PJUMP : TBLPTR);

END;

(* SEGMENT TABLE STRUCTURE *)

VAR

CUTFILE : TEXT:
TEAMFILE : TEXT:
SCRATCH : TEXT;
MESSAGEFILE : TEXT;

(* OUTPUT FILE *)

(* NEST DIAGS TEMP FILE *)

(* INFORMATION FROM COMPILER FILE *)

(* MESSAGE INPUT FILE *)

(* SEGMENT TABLE STRUCTURE *)

PARTITION : REAL;
REGCNT : INTEGER;
GOOD SEGMENT : BOOLEAN;
SEGNEST : INTEGER;
NUMBANKS : INTEGER;
PART_NUM : INTEGER;
LIMIT : INTEGER;

(* SBR NEST LEVEL CHECK *)

(* MEMORY SIZE LIMIT *)

(*) CODE TBL VARS*)

(* CODE TBL VARS*)

(* TABLE VARS *)

(* CODE TBL VARS*)

(* TABLE VARS *)

(* INS SET VAR *)

(* PROGRAM LABELS *)

(* 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 CEEUG.

```pascal
PROCEDURE DUMP SEGTEBL(VAR OUTFILE: TEXT; HDRPTR: TBLPTR);

VAR F_JMPLINK: SBRLINK, SBR, CURTP, SBRTF: TBLPTR;

BEGIN
  SBRTF := HDRPTR <> NIL DO
  BEGIN
    WRITELN (OUTFILE): ;
    WRITELN (OUTFILE); ;==----------------------------------------------------"
    WRITE (OUTFILE, ' SBR CODE NUMBER '); ;
    WRITELN (OUTFILE, 'SBR # STOP ADDR:2'); ;
    WRITELN (OUTFILE, 'SBR # STOP ADDR:2'); ;
    CURTP := SBRTF).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_JMPLIST;
        WHILE (SBRLINK <> NIL) OR (F_JMPLINK <> NIL) DO
          BEGIN
            IF SBRLINK <> NIL THEN
              BEGIN
                SBRTF := SBRLINK). SBR;
                CASE SBRTF). TAG OF
                TABLE: SBRTF := SBR;
                SBRBREAK:
                BEGIN
                  SBRTF := SBR). SBRTF;
                  WRITELN (OUTFILE, '*** BREAK ***
                  END;
                END;
                WRITE (OUTFILE, ' SBR INVOKE FROM', SBRLINK). FROM:SBRTF). TO
                SBRTF). START ADDR:5 ***
                SBRLINK := SBRLINK). NEXT_SBR;
                END
              ELSE
                WRITE (OUTFILE,' 
                IF F_JMPLINK <> NIL THEN
                BEGIN
                  F_JMPLINK). JUMP_ADDRTO:5
                  BEGIN
                    F_JMPLINK := F_JMPLINK). NEXT_FJUMP;
                    WRITELN (OUTFILE);
                    END
                    END
                  END;
```
CURTP:=CURTP@.TABLELIST;
END;
SBRTP:=SBRTP@.SBRLIST;
END;

(* DUMP SEGTEST TEST ROUTINE *)

(* DUMP MEMODULENODES: PRINTS OUT THE CONTENTS OF THE
MEMODULENODES LIST FORMED. THIS IS A DEBUGGING
ROUTINE AND IS NOT INVOKED IN THE PROGRAM. *)
PROCEDURE DUMPMEMODULENODES(HEAD_MEMODULE:TBLPTR);
VAR S:TELPTR;
BEGIN
S:=HEAD_MEMODULE;
WHILE S<>NIL DC
BEGIN
WITH S@ DO
BEGIN
WRITELN(OUTFILE); 
WRITE(CUTFILE,"MEMNUM OFFSET HIGH LOW");
WRITELNL(OUTFILE,"SEGTBLSTART");
WRITELNL(OUTFILE,"MEMNUM:9,OFFSET:8,HIGHOFFSET:6
LOWOFFSET:5,SEGTBLSTART_ADDR:10");
WRITELNL(OUTFILE);
END;
S:=S@.NEXT;
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(CUTFILE,"NUM:1");
END;

(* WRITE_LEADZERO *)
* WRITELBL: WRITES OUT THE TI-59 CODED LABELS

PROCEDURE WRITELBL(VAR OUTFILE:TEXT; CODESS:INTEGER); 
BEGIN
  WRITELN (OUTFILE, TILBL (.CODESS. ));
END;

(* WRITELBL *)

* WRITECODES: WRITES THE ADDRESS AND KEYCODE TO LINE

PROCEDURE WRITECODES(VAR OUTFILE:TEXT; CUR:CODEPTR);
BEGIN
  WRITE LEADZERO (OUTFILE, CUR#. ADDR, 3);
  WRITE (OUTFILE, '
');
  WRITE LEADZERO (OUTFILE, CUR#. KEY, 2);
  WRITE (OUTFILE, '.
');
END;

(* WRITECODES *)

* WRITENUM: WRITES KEYCE AS A NUMBER NOT A LABEL

PROCEDURE WRITENUM(VAR OUTFILE:TEXT; CUR:CODEPTR);
BEGIN
  WRITE LEADZERO (OUTFILE, CUR#. KEY, 2);
END;

(* WRITENUM *)

* HANDLE_ISTEPS: 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);
  WRITENUM (OUTFILE, CUR);
END;  (* HANDLE_ISTEP *)
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 ONLY. *)

PROCEDURE PRINT_CODELIST(VAR OUTFILE:TEXT;
VAR BUILTCODE:CODEPTR);
VAR CUR:CODEPTR;
BEGIN
  CUR:=BUILTCODE;
  WHILE CUR <> NIL DO
  BEGIN
    HANDLE_OSTEP(OUTFILE,CUR);
    IF CUR@.KEY IN (.71,.76.) THEN
    BEGIN
      CUR:=CUR@.SEQ;
      HANDLE_OSTEP(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; (* PRINT CODELIST *)

(*-----------------------------*)
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)); (* FIND_MSG *)

PROCEDURE INIT_SETS (VAR TEMPFILE:TEXT; VAR STEP 0, STEP 1, STEP 2, STEP 3: INSTR SET; VAR GOOD SEGMENT: BOOLEAN; VAR MESSAGEFILE: TEXT; VAR TILBL: TYPETABLES; VAR SBRINVNEST: INTEGER); VAR C: CHAR; DIGIT, J, L, K, I: INTEGER;

PROCEDURE GET_REGCOUNT (VAR REGCOUNT: INTEGER); BEGIN FIND_MSG (SCRATCH, REGCT); READLN (SCRATCH, REGCOUNT); (* GET_REGCOUNT *) BEGIN SBRINVNEST := 0; (* INITIALIZES THE INVOKE NEST CHECK *) RESET (MESSAGEFILE); (* INITIALIZE TILABELS *) L := 1; REPEAT READ (MESSAGEFILE, C); IF C = $ THEN READLN (MESSAGEFILE, DIGIT) ELSE READLN (MESSAGEFILE); UNTIL (C = $) AND (DIGIT = ALPHABTL); L := 0; FOR I := 0 TO ALPHABTL 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;
ELSE
TILBL (: I:) := 'BLANK';
END;

GET_REGCOUNT (REGCOUNT);

REWRIIE(TEMPPFILE); (* OPENING AND MARKING *)
WRITE(TEmPPFILE, '/S9'); (* THE TEMPPFILE WITH MSG 9 *)
REWRIIE(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
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 L: INTEGER;
BEGIN
IF CUR2.KEY IN STEP_3 THEN
BEGIN
FOR L := 1 TO 4 DO
IF CUR2.SEQ <> NIL THEN
CUR2 := CUR2.SEQ
END
ELSE
IF CUR2.KEY IN STEP_2 THEN
BEGIN
FOR L := 1 TO 3 DO
IF CUR2.SEQ <> NIL THEN
CUR2 := CUR2.SEQ
END
ELSE
IF CUR2.KEY IN STEP_1 THEN
BEGIN
FOR L := 1 TO 2 DO
IF CUR2.SEQ <> NIL THEN
CUR2 := CUR2.SEQ
END
ELSE
IF CUR2.SEQ <> NIL THEN
CUR2 := CUR2.SEQ;
END;

(*-- ADVANCE_CODEPTR --*)

199
(* PRINTLN_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;
      END;
      READLN(MESSAGEFILE);
      WRITELN(OUTFILE);
      READ(MESSAGEFILE, C1);
    END;
END; (* PRINTLN_MSG *)

(* PRINT_MSGLINE1: 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; (* 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:=19.29;
      PART NUM:=3;
      LIMIT:=719;
    END
END
ELSE IF REGCOUNT*REGNUM 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;

(* DET_LIMIT *)

PROCEDURE CLEAN(VAR CURTP: TBLPTR; VAR DELETE: INTEGER);
VAR F, S: TBLPTR;

(* PRUNE_SAMEF: REMOVES SAME F JUMP 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#: JUMPADDRTO = F#: JUMPADDRTO THEN BEGIN
      T#: NEXT_FJUMP := S#: NEXT_FJUMP;
      DISPOSE(S#: FWD_JUMP);
      S := F#: NEXT_FJUMP
    END ELSE BEGIN
      T := F#: NEXT_FJUMP;
      S := S#: NEXT_FJUMP;
    END;
    IF F#: NEXT_FJUMP <> NIL THEN
      F := F#: NEXT_FJUMP;
  END;

(* PRUNE_SAMEF *)
Procedure Prune_Greator: Removes FJumps contained in SEGTBL

BEGIN
  WHILE F<>NIL DO
  BEGIN
    IF F@.JUMP_ADDR<=CURTP@.STOP_ADDR THEN
      BEGIN
        S@.NEXT_FJUMP:=F@.NEXT_FJUMP;
        LF:=S@.NEXT_FJUMP;
        DELETE:=DELETE+1;
      END
    ELSE
      BEGIN
        S@:=S@.NEXT_FJUMP;
        F@:=F@.NEXT_FJUMP
      END;
    END;
    IF CURTP@.F_JUMPLIST@.JUMP_ADDR<=CURTP@.STOP_ADDR THEN
      BEGIN
        F:=CURTP@.F_JUMPLIST;
        CURTP@.F_JUMPLIST:=F@.NEXT_FJUMP;
        DISPOSE(F@.FWD JUMP);
        DELETE:=DELETE+1;
      END;
  END;
  (* ---* PRUNE_Greator *)

Procedure Prune_Sames: Removes same SBR Invokes from SEGTBL

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;
    END;
    IF S<>NIL THEN
      BEGIN
        T@:=T@.NEXT_SBR;
        DISPOSE(T@.SBRPTR);
        S:=T@.NEXT_SBR;
        IF S<>NIL THEN
          SS:=PASS_BRK(S@.SBR);
      END;
  END;
  (* ---* PASS_BRK *)

FUNCTION PASS_BRK(F: TBLPTR): TBLPTR;
BEGIN
  CASE F@.TAG OF
    SBRBREAK: PASS_BRK:=F@.SBR;
    TABLE: PASS_BRK:=F;
  END;
END;

(* ---* PASS_BRK *)
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;

(* PRUNE_SAMES *)
BEGIN
DELETE:=0;
IF CURTP@.F_JUMPLIST<>NIL THEN
BEGIN
F:=CURTP@.F_JUMPLIST;
PRUNE_SAMES(F);
END;
IF CURTP@.F_JUMPLIST<>NIL THEN
BEGIN
S:=CURTP@.F_JUMPLIST;
F:=S@.NEXT_FJUMP;
PRUNE_CREATOR(F,S,DELETE);
END;
IF CURTP@.SBRLIST<>NIL THEN
BEGIN
F:=CURTP@.SBRLIST;
PRUNE_SAMES(F);
END;
END;

(* CLEAN *)

(* TAGS_NEST1SRBBK: DIAGNOSTIC PRINTOUT IF THERE IS A *)
(* A SEG BREAK WITHIN AN ITERATIVE LOOP. NEEDS TO SET *)
(* GOOD SEGMENT VARIABLE FALSE *)
PROCEDURE TAGS_NEST1SRBBK(VAR TEMPPF:TEXT; SEG:TBLPTR;
VAR GOOD_SEGMENT:BOOLEAN);
VAR IS_BRK_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 SER@.SERLIST<>NIL THEN
BEGIN
SBRL:=SER@.SERLIST;
WHILE SBRL<>NIL DO
BEGIN
SER:=SBRL@.SBR;
IF SBRL@.TAG=SBBREAK THEN
IS_BRK_BELOW:=TRUE
ELSE
BELOW_BREAK(SBR,IS_BRK_BELOW);
SERL:=SBRL@.NEXT_SBR;
END;
END;
END;
BEGIN
  IS_BRK_BELOW := FALSE;
  BREAK(SEG, IS_BRK_BELOW);
  IF (IS_BRK_BELOW) AND (SEG.NEST = 1) THEN
    BEGIN
      GCOD_SEGMENT := FALSE;
      WRITE(TEMPFILE);
      WRITE(TEMPFILE, '* SBR BREAK WITHIN A LOOP');
      WRITE(TEMPFILE, ' LOOP BOUNDS', SEG.START_ADDR:4, TO', SEG.STOP_ADDR:4);
    END;
  END;
END;

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

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

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

PROCEDURE DIAGS_NESTLENGTHCHK(VAR TEMPFILE:TEXT;
                              CUR:TBPLTR; VAR GOOD_SEGMENT:Boolean); BEGIN
  IF (CUR.LENGTH > LIMIT) AND (CUR.NEST=1) THEN
    BEGIN
      GCOD_SEGMENT := FALSE;
      WRITE(TEMPFILE);
      WRITE(TEMPFILE, '* BACK JUMP NEST TOO LONG');
      WRITE(TEMPFILE, ' LOOP BOUNDS', CUR.START_ADDR:4, TO', CUR.STOP_ADDR:4);
    END;
END;

(*-----------------------------------------------*)
(*) DIAGS_NEST6SBRINVCHK: CHECKS THAT THE SBR NEST LEVEL *)
(*) DOES NOT EXCEED 6 *)
(* --------------------------------------------------------------------- *)
PROCEDURE DIAGS_NEST6SBRINVCHK(VAR TEMPFILE: TEXT;
CUR: TBLPTR; VAR GOOD SEGMENT: BOOLEAN;
SBRINVNEST: INTEGER);

BEGIN
IF SBRINVNEST > 7 THEN
BEGIN
GOOD SEGMENT:=FALSE;
WRITE(TEMPFILE, '* SBR INVOKE NEST LEVEL > 6');
WRITE(TEMPFILE, 'CALLES ROUTINE STARTS ');
WRITELN(TEMPFILE, '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, SBF: TBLPTR;
BEGIN
SBRLST:=SBRL;
WHILE SBRLST<>NIL DO
BEGIN
SBF:=SERLIST\$.SBF;
CASE SBF\$.TAG OF
TABLE:
SBF:=SBF\$.SBR;
SBR BREAK:
SBF:=SERLIST\$.SBF;
END;
IF (SBF\$.SERLIST<>
NIL) AND (SBF\$.COALESCED= TRUE) THEN
RESET_INCLUDED(SBF\$.SERLIST);
IF (SBF\$.COALESCED=TRUE) THEN
SBF\$.INCLUDED:=FALSE;
SERLIST:=SERLIST\$.NEXT_SBR;
END;
END;

(*) RESET_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: CODEPTR;
BEGIN
FIND MSG(SCRATCH, CODENUM);
READ(SCRATCH, TEMP);
IF TEMP > -1 THEN
BEGIN
NEW(CUR);
BUILT_CODE:=CUR;

205
PROCEDURE SETJmps(VAR BUILT_CODE:CODEPTR);
VAR CUR:CODEPTR;
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.AIADDR<>ADDRESS) DO
      SEARCH := SEARCH@.SEQ;
      MARKER@.SEQ@.JMP := SEARCH;
  END;
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 (.71,.76.) THEN
        CUR := CUR@.SEQ@.SEQ
      ELSE
        ADVANCE_CODEPTR (CUR);
    END;
  END;  (* TEST_SETJMP *)

(* INPUT *)
**BUI
t SEGMENT TA
BLE ROUTINES: ON THIS TABLE ALL
**
**OF THE COALESCING IS DONE, AND NOT THE CODE.
**

PROCEDURE BUI
LE_TB
I
TLE(BUIL
T_CODE:CODEPTR;
VAR SEG
I
BLE:TB
L
PLE TR;
LIMIT:INTEGER;
BUIL
T_CODE_COUNT:INTEGER);

VAR HERE
TR:TEL
P
TR;

PROCEDURE BUI
LE_PRIMSEG
TLE(BUIL
T_CODE:CO
DEPTR;
VAR HDR
PTR:TB
L
PLE TR);

VAR CUR
CP:CODEPTR;
CUR
TP:TL
P
TR;

PROCEDURE PRO
CESS_SB
R
BL
L(BU
IL
T_CODE:CO
DEPTR;
VAR CUR
CP:CO
DEPTR;
CUR
TP:TB
L
PLE TR);

VAR TRAILTP:TEL
P
TR;
BEGIN
TRAILTP:=CUR
TP;
NEW(CUR
TP,TA
BLE);
CUR
TP@.TAG:=TABLE;
CUR
TP@.TABLELIST:=NIL;
CUR
TP@.COALESCED:=FALSE;
CUR
TP@.INCLUDED:=FALSE;
CUR
TP@.SB
R
BLIST:=NIL;
CUR
TP@.START ADDR:=CUR
CP@.ADDR;
CUR
TP@.STOP ADDR:=1;
IF CUR
CP@.KEY = 76 THEN
BEGIN
CUR
TP@.STOP ADDR:=CUR
CP@.SEQ@.KEY;
CUR
CP:=CUR
CP@.SEQ@.SEQ
END
ELSE
CUR
TP@.STOP ADDR:=1;
IF TRAILTP<>NIL THEN
TRAILTP@.SB
R
BLIST:=CUR
TP;
END;

(* PROCESS_SB
R
BL
L *)
PROCEDURE PROCESS_SBR_CODE(VAR CURCP:CODEPTR; VAR SBRHDRTP:TABLEPTR);

VAR TOPTP,CURTP:TBLPTR;

(* IS_BCK_JMP: DETERMINES IF THE JUMP IS BACKWARDS *)
FUNCTION IS_BACK_JMP(CURCP:CODEPTR):BOOLEAN:
VAR ADDRESS:INTEGER;
BEGIN
IF (CURCP@.KEY IN STEP_2) THEN
   BEGIN
      ADDRESS:=CURCP@.SEQ@.SEQ@.JMP@.ADDR;
      IF ADDRESS > CURCP@.ADDR THEN
         IS_BACK_JMP:=FALSE
      ELSE
         IS_BACK_JMP:=TRUE;
   END
ELSE
IF (CURCP@.KEY IN STEP_3) THEN
   BEGIN
      ADDRESS:=CURCP@.SEQ@.SEQ@.SEQ@.JMP@.ADDR;
      IF ADDRESS > CURCP@.ADDR THEN
         IS_BACK_JMP:=FALSE
      ELSE
         IS_BACK_JMP:=TRUE;
   END
ELSE
   IS_BACK_JMP:=FALSE;
END;
(* IS_BACK_JMP *)

PROCEDURE APND_JMP_TBL(CURCP:CODEPTR; VAR TOPTP:TBLPTR);

VAR ADDRESSPR,ADDRESSSTO:INTEGER;

PROCEDURE INSERT_CRITS(ADDRESS:INTEGER; VAR TOPTP:TBLPTR);

VAR CURTP,TRAILTP,INSERTTP:TBLPTR;
BEGIN
TRAILTP:=TOPTP;
CURTP:=TCFTPA.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;

BEGIN CURTP:=TCFTP;
WHILE CURTP@.START_ADDR <> ADDRESSSTO DO
  BEGIN
    CURTP:=CURTP@.TABLELIST;
    IF CURTP@.NEST = 1 THEN CURTP@.NEST:=0;
  END;
END;

PROCEDURE SET_NESTS (ADDRESSFR,ADDRESSSTO:INTEGER;
VAR TOPTP:TBLPTR);
VAR CURTP:TEL?TR;
BEGIN
  CURTP:=TCFTP;
  WHILE CURTP@.START_ADDR <> ADDRESSSTO DO
    BEGIN
      CURTP:=CURTP@.TABLELIST;
      IF CURTP@.NEST = 1 THEN CURTP@.NEST:=0;
    END;
END;

BEGIN CURTP@.NEST:=1;
ADDRESSST:=CURCP@.SEQ3.SEQ3.JMP3.ADDR;
ADDRESSST:=CURCP@.SEQ3.SEQ3.JMP3.ADDR;
END

BEGIN
  NEW (CURTP,TABLE);
  CURTP@.TAG:=TABLE;
  CURTP@.START_ADDR:=SBRHDRTP@.START_ADDR;
  CURTP@.NEST:=0;
  CURTP@.COALESCED:=FALSE;
END;

(* APND_JMP_TEL *)
BEGIN
  DISPOSE(INSERTTP,TABLE);
  DISPOSE(INSERTTP,TOPTP);
  DISPOSE(INSERTTP,TOPTP);
END;

(* APND_JMP_TEL *)
CURTP. INCLUDED = FALSE;
CURTP. TABLELIST := NIL;
CURTP. SERLIST := NIL;
SBRHDRTP. TABLELIST := CURTP;
TCRP := SBRHDRTP;
WHILE ((CURTP. KEY <> 76) AND (CURCP@.SEQ <> NIL ))
DO
IF IS_BACK_JMF (CURCP) THEN
BEGIN
APPD JMF_TBL(CURCP,T0TP);
ADVANCE_CODECPR (CURCP)
END
ELSE
ADVANCE_CCDEPTR (CURCP);
IF CURCP@.SEQ = NIL THEN
TOPTP@.TABLELIST@.STOP_ADDR := CURCP@.ADDR;
ELSE
TOPTP@.TABLELIST@.STOP_ADDR := CURCP@.ADDR - 1;
END;

(* PROCESS SBRCODE *)

BEGIN
CURCP := BUILT_CODE;
CURT := NIL;
PROCESS_SBRLBL (CURCP, CURTP);
HDRTR := CURTP;
WHILE (CURCP@.SEQ <> NIL) DO
BEGIN
PROCESS_SERCODE (CURCP, CURTP);
IF CURCP@.KEY = 76 THEN
PROCESS_SBRLBL (CURCP, CURTP);
END;

(* BLD_PRIMSEGTEL *)

PROCEDURE BLD_ADVSEGTL (VAR HDRPTR:TBLPTR);
VAR SBRTP:TBLPTR; STOP: INTEGER;

PROCEDURE MERGE_OONES (VAR SBRTP:TBLPTR);
VAR MARK,ZERO,TABLE:

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 (LIST, TABLE);

BEGIN

MARK := SBRTP@TABLELIST;
WHILE MARK@TABLELIST <> NIL DO
BEGIN
  IF (MARK@NEST = 0) AND (MARK@TABLELIST<>NIL)
  THEN MARK := MARK@TABLELIST;
  ELSE 
  IF ONE <> ZERO THEN
  BEGIN
    MERGE (ONE, ZERO, MARK);
    MARK := ONE;
  END;
  END;
END;

IF MARK@TABLELIST <> NIL THEN
MARK := MARK@TABLELIST;

END; (* MERGE *)

(* ADD_ZEROS: FILLS IN GAPS IN TABLE WITH O SEG *)

PROCEDURE ADD_ZEROS (VAR SBRTP: TABLEPTR);
VAR CUR, TRAIL, INSERT, TBLPTR, STOP: INTEGER;
BEGIN
TRAIL := SBRTP;
CUR := SBRTP@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
SBRTP := HDRPTR;
WHILE SBRTP <> NIL DO
BEGIN
MERGE_ONES (SBRTP);
ADD_ZEROS (SBRTP);
END

END;
**SBRTP:=SERTP@.SBRLIST; END:**

(* BLD_ADVSEGTEL *)

(* BLD_FINSEGTEL: PROCESS CODE FOR SBR INVOKES AND FJUMPS. WHEN ENCOUNTERED IT PLACES INTO SEGtbl. 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/PEMPT CODE. ONLY SEQUENTIAL CONTINUATION CODE IS INCLUDED IN LENGTH CALCULATION TOGETHER WITH FJUMP PEIPT CODE. *)

PROCEDURE BLD_FINSEGTEL (BUILT CODE: CODEPTR; VAR HDRPTR:TBLPTR; LIMIT: INTEGER); VAR CURCP: CODEPTR; SERTP:TBLPTR;

(* PROCESS SBRSEGTEL: PLACES SBRs & FJMP INTO SEGTL *)
PROCEDURE PROCESS_SBRSEGTEL (VAR CURCP: CODEPTR; VAR HDRPTR, SERTP: TBLPTR);

(* HANDLE_FWDJMP: INSERTS FWD JUMPS INTO TABLE. *)
PROCEDURE HANDLE_FWDJMP (CURCP: CODEPTR; VAR HDRPTR, CURTP, FJMP: TBLPTR);

VAR ADDRESSSTO, ADDRESSSR: INTEGER;

INSERT: TBLPTR;
BEGIN IF CURCP@.KEY IN STEP.3 THEN BEGIN ADDRESSSR:=CURCP@.SEQ@.SEQ@.SEQ@.AMD; ADDRESSSTO:=CURCP@.SEQ@.SEQ@.SEQ@.AMD; END ELSE BEGIN ADDRESSSR:=CURCP@.SEQ@.SEQ@.AMD; ADDRESSSTO:=CURCP@.SEQ@.SEQ@.AMD; END;

NEW (INSERT, FWD JUMP); INSERT@.TAG:=FWD JUMP;
INSERT@.JUMP ADDR:=ADDRESSSR;
INSERT@.JUMP ADDRT:=ADDRESSSTO;
INSERT@.JUMP ADRT:=1;
INSERT@.JUMP ADRT01:=1;
INSERT@.JUMP ADRT02:=1;
INSERT@.NEXT FJUMP:=NIL;
IF CURCP@.F JUMPLIST = NIL THEN BEGIN CURCP@.F JUMPLIST:=INSERT;
FJMP:=INSERT; END ELSE BEGIN FJMP@.NEXT FJUMP:=INSERT;
FJMP:=INSERT END;

END; (* HANDLE_FWDJMP * )

212
```plaintext
(* 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@.ADDR;
IF ADDRESS > CURCP@.ADDR THEN
  IS_FWD_JMP:=TRUE
ELSE
  IS_FWD_JMP:=FALSE;
END
ELSE IF (CURCP@.KEY IN STEP_3) THEN
BEGIN
ADDRESS:=CURCP@.SEQ@.SEQ@.SEQ@.JMP@.ADDR;
IF ADDRESS > CURCP@.ADDR THEN
  IS_FWD_JMP:=TRUE
ELSE
  IS_FWD_JMP:=FALSE;
ELSE
  IS_FWD_JMP:=FALSE;
END; (* IS_FWD_JMP *)

(* HANDLE_SBR_INVOKE: PLACES SBR CALL INTO TABLE *)
PROCEDURE HANDLE_SBR_INVOKE (VAR CURCP:CODEPTR;
VAR HDRPTR,CURTP,SBR_INVOKE:TBLPTR);
VAR TOSBR,INSERT:TBLPTR;
KEYY: INTEGER;
BEGIN
TOSBR:=HDRPTR;
KEYY:=CURCP@.SEQ@.KEY;
WHILE TOSBR@.STOP ADDR > KEYY DO
  TOSBR:=TOSBR@.SBR_LIST;
NEW (INSERT,SBRPTR);
INSERTS.TAG:=SBRPTR;
INSERTS.FROM:=CURCP@.ADDR+1;
INSERTS.SER:=TOSBR@.TABLELIST;
INSERTS.NEXT_SBR:=NIL;
IF CURTP@.SBR_LIST = NIL THEN
BEGIN
  CURTP@.SBR_LIST:=INSERT;
  SBR_INVOKE:=INSERT
END
ELSE
BEGIN
  SBR_INVOKE@.NEXT_SBR:=INSERT;
  SBR_INVOKE:=INSERT;
END;
END; (* HANDLE_SBR_INVOKE *)

BEGIN
CURTP:=SBRTP@.TABLELIST;
REPEAT
  CURTP@.SER_LIST:=NIL;
  CURTP@.P_JUMPLIST:=NIL;
  SBR_INVOKE:=NIL;
  P JUMP:=NIL;
WHILE CURCP@.ADDR < CURTP@.STOP_ADDR DO
BEGIN
  IF CURCP@.KEY = 71 THEN
```
HANDLE_SBR_INVOKE(CURCP, HDRPTR, CURTP, SBR_INVOKE);
IF IS_FWD JUMP(CURCP) THEN
  HANDLE_FWD JUMP(CURCP, HDRPTR, CURTP, FJMP);
ELSE IF ((CURCP$@.KEY=76) OR (CURCP$@.KEY=71)) THEN
  CURCP := CURCP$@.SEQ$@.SEQ;
END;
ADVANCE_CODEPTR(CURCP);
END;
CURTP := CURTP$@.TABLELIST UNTIL (CURTP = 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 FJUMP LISTS THEN CONTINUE *)
(* NOTE: MAY LEAD TO PROBLEMS IS LIMIT IS *)
(* ARBITRARILY SMALL. *)
PROCEDURE SET_LENGTH(BUILT_CODE:CODEPTR;
VAR SBRT$@:TBLPTR; LIMIT: INTEGER);
VAR CURTP$@:TBLPTR;
LENGTH, DELETE, L_FJUMP: INTEGER;
BEGIN
CALCULATE: DETERMINES LENGTH OF A SEGMENT. WILL*
NOT ADD ADDITIONAL STEPS FOR DUPLICATE FJMP *
ADDRTOS
PROCEDURE CALCULATE(CURTP$@: TBLPTR;
VAR LENGTH: INTEGER);
VAR S, F: TBLPTR;
ADDITIONS: INTEGER;
BEGIN
LENGTH := CURTP$@.STOP_ADDR-CURTP$@.START_ADDR
         +FJUMP_CONST+1;
IF CURTP$@.F_JUMPLIST<>NIL THEN
  BEGIN
    F := CURTP$@.F_JUMPLIST;
    IF F$@.JUMP_ADDRTO > CURTP$@.STOP_ADDR THEN
      BEGIN
        ADDITIONS := 0;
        S := F$@.JUMPLIST;
        IF S$@.JUMP_ADDRTO > F$@.JUMP_ADDRTO THEN
          BEGIN
            ADDITIONS := ADDITIONS + 1;
            S := S$@.NEXT_FJUMP;
            WHILE ((S<>F) AND (S$@.JUMP_ADDRTO<>F$@.JUMP_ADDRTO)) DO
              S := S$@.NEXT_FJUMP;
            IF S<>F THEN
              ADDITIONS := ADDITIONS - 1;
          END;
        F := F$@.NEXT_FJUMP;
      END;
      LENGTH := LENGTH + (ADDITIONS) * (FJUMP_CONST);
  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;
         END;
   L_PossBR:=COUNT*SBRCONST;
   END; (* L_SBBRKS *)
(* DIVIDE: DIVIDES A SEG IN HALF AND RESETS FJMF *)
(* AND SBR POINTERS. *)

PROCEDURE DIVIDE(BUILT_CODE:CODEPTR;
VAR CURTP,TBLPTR);
VAR INSERT,TBLPTR;
S,P,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 F,T:CODEPTR;
BEGIN
P:=BUILT_CODE;
WHILE (P3.AADDR <= NEW_STOP) DO
BEGIN
T:=P;
IF (T3.KEY = 76) OR (T3.KEY = 71) THEN
F:=P3.SEQSEQ;
ELSE
ADVANCE_CODEPTR(P);
END;
IF (T3.KEY = 76) OR (T3.KEY = 71) THEN
T:=T3.SEQSEQ;
NEW_STOP:=T3.AADDR-1;
END;

(* FIND_INSSBLIST: DIVIDES UP THE SBRLIST *)
(* BETWEEN THE CLD AND NEW SEGMENTS *)

PROCEDURE FIND_INSSBLIST(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@.FROMKLIMIT) 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
216
procedure find_insjumplist (var curtp, f: tblptr; new_stop: integer);

var limit: integer;

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;

begin
f := f@.next_fjump;

begin
f := f@.next_fjump;

begin
f := f@.next_fjump;

end

end

end

end

end

else

begin

begin

begin

begin

end

end

end

end

end

end;

(* find_insjumplist *)
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 * JUMPCONST;
CURTP.LENGTH := CURTP.LENGTH - PJUMPCONST;
CURTP := CURTP . TABLELIST;
END
END
ELSE
BEGIN
CLEAN (CURTP, DELETE);
DELETE := 0; (*CALCULATE HAS THIS COVERED *)
CURTP.LENGTH := LENGTH - DELETE * PJUMPCONST;
CURTP := CURTP . TABLELIST
END;
UNTIL (CURTP = NIL);
END;
(*------ ----------------* )
BEGIN
CURCP := BUILT_CODE;
SBRTP := HDRPTR;
WHILE SBRTP <> NIL DO
BEGIN
PROCESS_SERSEGTL(CURCP, HDRPTR, SBRTP);
SET LENGTH (BUILT_CODE, SBRTP, LIMIT);
SBRTP := SBRTP . SRLIST
END;
END;
(*----------------------* )
BEGIN
IF BUILT_CODE_COUNT <= LIMIT THEN
BEGIN
NEW (SEGTL, TABLE) :
SEGTL . TAG := TABLE;
SEGTL . TABLELIST := NIL;
SEGTL . SBRLIST := NIL;
SEGTL . COALESCED := TRUE;
SEGTL . INCLUDED := FALSE;
SEGTL . START ADDR := 0;
SEGTL . STOP ADDR := BUILT_CODE . SEQ . KEY;
NEW (HDRPTR, TABLE) :
HDRPTR . START ADDR := 0;
HDRPTR . STOP ADDR := BUILT_CODE_COUNT;
HDRPTR . NEST := 0;
HDRPTR . SBRLIST := NIL;
HDRPTR . SRLIST := NIL;
HDRPTR . F JUMPLIST := NIL;
HDRPTR . COALESCED := TRUE;
HDRPTR . INCLUDED := FALSE;
HDRPTR . LENGTH := BUILT_CODE_COUNT + 1;
SEGTL . TABLELIST := HDRPTR;
END
ELSE
BEGIN
BLD_PRMSEGTL (BUILT_CODE, HDRPTR);
BLD_ADSEGTL (HDRPTR);
BLD_FINSEGTL (BUILT_CODE, HDRPTR, LIMIT);
SEGTL := HDRPTR;
END;
(* BLD_SEGTL *)
* COALESCE: COALESCES THE SEG TABLE MAKING GOOD ERK. *
* 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 SBR\_NEST: INTEGER); VAR CURSEG: TBLPTR;

(* SERSUM: SUMS ALL SBR\_S ON A SBR\_LIST. SETS INCLUDES TO TRUE. ADDS SBR\_CONST IF SBS\_BREAK IS ENCOUNTERED. *)

PROCEDURE SRS\_SUM(VAR SBR\_LIST: TBLPTR; VAR SUMS\_SBR: INTEGER);
VAR SRL, SR: TELPTR;
BEGIN
SRL := SBR\_LIST;
WHILE SRL > NIL DO
BEGIN
SR := SRL\_SER;
CASE SR\_TAG OF TABLE:
BEGIN
IF (SR\_COALESCED) THEN
IF NOT (SR\_INCLUDED) THEN
BEGIN
SR\_INCLUDED := TRUE;
SUMS\_SBR := SUMS\_SBR + SR\_LENGTH - SB\_CONTCONST;
IF SR\_SBR\_LIST > NIL THEN
SBS\_SUM(SR\_SBR\_LIST, SUMS\_SBR);
END;
SBS\_BREAK := SUMS\_SBR;
SUMS\_SBR := SUMS\_SBR + SB\_CONST; END;
END;
SRL := SRL\_NEXT\_SR;
END;
END:

(* SERSUM *)

(* SBS\_SUMLINK: ADDS ALL SBS\_S BELOW AN INVOKE NODE. *)

PROCEDURE SBS\_SUMLINK(VAR SBR\_PTR: TBLPTR; VAR SUMP: INTEGER);
VAR PTR, PTR2: TELPTR;
BEGIN
PTR := SBR\_PTR\_SER;
CASE PTR\_TAG OF TABLE:
IF (PTR\_COALESCED) AND (PTR\_TABLE\_LIST = NIL) THEN
IF NOT (PTR\_INCLUDED) THEN
BEGIN
PTR\_INCLUDED := TRUE;
SBS\_SUM(PTR\_SBR\_LIST, SUMP);
SUMP := SUMP + PTR\_LENGTH - SB\_CONTCONST;
END
ELSE
IF (PTR\_COALESCED) AND (NOT (PTR\_INCLUDED) THEN
SBS\_BREAK := SUMP + PTR\_LENGTH - SB\_CONTCONST;
SUMP := SUMP;
END;
END:

(* SBS\_SUMLINK *)
PROCEDURE CHK_SEGSIZE(VAR CURSEG: TBLPTR; LIMIT: INTEGER);
  VAR SEG, SBR: TBLPTR;
  LENGTH, SUMSBR: INTEGER;

  (* INSERT_SBRBRK: INSERTS THE SBRBREAK NODE *)
PROCEDURE INSERT_SBRBRK(SERNODE: TBLPTR);
  BEGIN CUR := SERNODE;
    IF CUR.SBR.TAG <> SBRBREAK THEN
      BEGIN PND := 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 CHK_SIZE := SEG.LENGTH;
        IF SEG.TABLELIST = NIL THEN
          CHK_SIZE := CHK_SIZE = SEGCONTCONST;
        IF SEG.SERLIST <> NIL THEN
          BEGIN SBRL := SEG.SERLIST;
            WHILE SBRL <> NIL DO
              BEGIN SER := SBRL.SBR;
                CASE SER.TAG OF
                  TABLE:
                    BEGIN IF (SER.COALESCD = FALSE) THEN
                      COALESC(SBR, LIMIT, GOOD SEGMENT, SBRINNEST);
                    END;
                    BEGIN INSERT_SBRBRK (SBR);
                      END;
                    SUMSBR := 0;
                    SBRSUMLINK (SBR, SUMSBR);
                    CHK_SIZE := CHK_SIZE + SUMSBR;
                    IF (CHK_SIZE > LIMIT) THEN
                      BEGIN INSERT_SBRBRK (SBR);
                        BEGIN CHK_SIZE := CHK_SIZE + SUMSBR;
                          RESET_INCLUDED (SEG.SERLIST);
                        END;
                        SBREAK := NIL;
IF SEG@.SERLIST<>NIL THEN
  RESET INCLUDED(SEG@.SERLIST);
  DIAG NESTS IS SBREAK(TEMPFIL,SEG,GOOD_SEGMENT);
  SEG=SEG@.TABLELIST;
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 AND IF THEY CAN BE MERGED. (*)

PROCEDURE CCMBINE(VAR CURSEG:TABLEPTR; LIMIT:INTEGER);
VAR SUMSBRE, SUMSBRG, SUMFWD, SUMCUR, SUMTOT:INTEGER;
CUR, FWD: TABLEPTR;
DELSUMSBRE, DELSUMSBRG, DELSUMFWD, DELETED:INTEGER;

(* MERGES: MERGES ADJACENT SEGMENTS TO INCLUDE THEIR RESPECTIVE JUMPLISTS AND SBRLISTS. IT THEN USES CLEAN TO REMOVE ANY DUPLICATE JUMPS/SERS. USES CLEAN'S DELETE FACILITY TO READJUST LENGTH. INCLUDED IN SEGMENT LENGTH, ROOM IS LEFT. *)

PROCEDURE MERGES(VAR CUR,FWD: TABLEPTR);
VAR SBRTAIL, JMPTAIL: TABLEPTR; 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;
          JMETAIL:=JMPTAIL@.NEXT_FJUMP;
        END;
      ELSE
        CUR@.F_JUMPLIST:=FWD@.F_JUMPLIST;
    END;
  END;

  IF FWD@.SERLIST<>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@.SERLIST;
      END;

      CUR@.STOP ADDR:=FWD@.STOP ADDR;
      CUR@.TABLELIST:=FWD@.TABLELIST;
      CUR@.NEST:=0;
      CLEAN(CUR, DELETE);
      CUR@.LENGTH:=CUR@.LENGTH+FWD@.LENGTH-((DELETE+1)*FJUMPCONST);
    DISPOSE(FWD, TABLE);
END;
(* MERGES *)

(*-----*)
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 T@JUMP_ADDRTO = T@JUMP_ADDRTO THEN FCOUNT:=FCOUNT+1; T:=T@NEXT_FJUMP; END; P:=P@NEXT_FJUMP; END; DELSUMFJMP:=(FCOUNT+1)*FJUMP_CONST; END; (* MOD_SUMTOTFJMP *)

MOD_SUMTOSBR SIMULATES THE CHANGE TO TOTAL LENGTH BECAUSE OF THE MERGING OF ADJACENT SEGMENTS. PROCEDURE MOD_SUMTOSBR (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 = SBR_BREAK THEN BEGIN T:=T@SBR; IF T@TAG = SBR_BREAK THEN BEGIN SCOUNT:=SCOUNT+1; END; T:=T@NEXT_SBR; END; P:=P@NEXT_SBR; END; DELSUMSBRZ:=SCOUNT*SBR_CONST; END; (* MOD_SUMTOSBR *)

BEGIN CUR:=CUR@SEG; ETAGS:=NEST@LENGTHCHK (TEMPFILE,CUR,GOOD_SEGMENT); SUMT:0; SUMSBR:=0; IF CUR@SBRLIST<>NIL THEN BEGIN SUMD:=CUR@SBRLIST@SUMSBR; SUMCUR:=CUR@LENGTH+SUMSBR; CUR:=CUR@TABLELIST; CUR:=CUR@SEG; END;
BEGIN
FWD:=CUR@.TABLELIST;
SUMSBRF:=0;
IF FWD@.SBRLIST<>NIL THEN
SSBSUM(FWD@.SBRLIST, SUMSBRF);
SUMFWD:=FWD@.LENGTH+SUMSBRF;
SUMTOT:=SUMCUR+SUMFWD;
MOD SUMTOTJMР (CUR,FWD, DELSUMFJМ);
MOD SUMCTSRB (CUR,FWD, DELSUMSBZ);
SUMTOT:=SUMTOT-DELSUMFJM-DELSUMBSRZ;
IF SUMTOT<=LIMIT THEN
BEGIN
MERGES(CUR, FWD);
CUR@.LENGTH:=CUR@.LENGTH-DELSUMSBZ;
SUMCUR:=SUMTOT;
END
ELSE BEGIN
RESET INCLUDED (FWD@.SBRLIST);
RESET INCLUDED (CUR@.SBRLIST);
CUR:=FWD;
DIAGS_NEST1LENGTHCHK(TMPFILE, CUR, GOOD SEGMENT);
SUMCUR:=CUR@.LENGTH;
SUMSRC:=0;
SBSUM (CUR@.SBRLIST, SUMSRC);
SUMCUR:=SUMCUR+SUMSRC;
END;
SUMTOT:=0;
END;
IF CUR@.SBRLIST<NIL THEN
RESET INCLUDED (CUR@.SBRLIST); (*ALL INCLUDES PST*)
CURSEG@.COALESCE:=TRUE;
END; (* COMBINE *)
BEGIN
CURSEG:=SBR;
SBRINVNEST:=SBRINVNEST+1;
DIAGS_NEST6SBRINVCHK (TMPFILE, CURSEG, GOOD SEGMENT, SBRINVNEST);
CHK SEGSIZE(CURSEG, LIMIT);
CCMERGE(CURSEG, LIMIT);
SBRINVNEST:=SBRINVNEST-1;
END; (* COALESCE *)

(*===============================================*)
PROCEDURE INSTRUCTIONS

VAR OUTFILE, MESSAGEFILE, TEMPFILE, TEXT, BUILT, CODE, CODEPTR;
SEGTL, TBLPTR, PART, NUM, INTEGER;
PARTITION, REAL, GOOD_SEGMENT, BOOLEAN;

VAR HEAD_MEMODULE: TBLPTR;

PROCEDURE BLD_MEMODULENODES(SEGTL: TBLPTR;
VAR HEAD_MEMODULE: TBLPTR;
MEMCOUNT: INTEGER);

PROCEDURE INSERT_MEMODULENODES(SEGTL: TBLPTR;
VAR HEAD_MEMODULE, TAIL_MEMODULE: TBLPTR;
VAR MEMCOUNT: INTEGER):

FUNCTION NOT_IN_MODULELIST(SEGTL: HEAD_MEMODULE:
TBLPTR): BOOLEAN;

VAR S: TBLPTR:
BEGIN
S:= HEAD_MEMODULE;
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;
(* NOT_IN_MODULELIST *)
BEGIN
IF NOT_IN_MODULELIST (SEGTL, HEAD_MEMODULE) THEN
BEGIN
NEW (INSERT, MEMODULE);
INSERT2.TAG:= MEMODULE;
INSERT2.MEMNUM:= MEMCOUNT;
MEMCOUNT:= MEMCOUNT+1;
INSERT2.OFFSET:= (SEGTL@.START_ADDR;
INSERT2.HIGHOFFSET:= (SEGTL@.START_ADDR DIV 100);
PROCEDURE RECURSE_BLD_MEMODULES(SEG: TBLPTR; VAR HEAD_MEMODULE, TAIL_MEMODULE, MEMMODULENODES: TBLPTR; VAR MEMCOUNT: INTEGER);

VAR SEG_SBR, SER, SBRLIST: TBLPTR;
BEGIN
SEG_SBR := SEG_TABLELIST;
WHILE SEG_SBR <> NIL DO
BEGIN
INSERT_MEMODULENODES(SEG_SBR, HEAD_MEMODULE, TAIL_MEMODULE, MEMCOUNT);
SEG_SBR := SEG_SBR_TABLELIST;
END;
SEG_SBR := SEG;
WHILE SEG_SBR.SBRLIST <> NIL DO
BEGIN
IF SEG_SBR.SBRLIST < SEG_SBR.SBRLIST
BEGIN
SBRL := SEG_SBR.SBRLIST;
WHILE SBRL < NIL DO
BEGIN
SBRL := SBRL.SBR;
IF SBRL.HEAD_MEMODULE = SBBREAK THEN
BEGIN
INSERT_MEMODULENODES(SBRL, SEG_SBR.HEAD_MEMODULE, TAIL_MEMODULE, MEMCOUNT);
END;
SBRL := SBRL.SBRLIST;
END;
END;
SEG_SBR := SEG_SBR_TABLELIST;
END;
END;
END; (* RECURSE_BLD_MEMODULENODES *)

BEGIN
HEAD_MEMODULE := NIL;
TAIL_MEMODULE := NIL;
MEMCOUNT := 1;
SEG := SEGTBL;
INSERT_MEMODULENODES(SEG, HEAD_MEMODULE, TAIL_MEMODULE, MEMCOUNT);
RECURSE_BLD_MEMODULES(SEG, HEAD_MEMODULE, TAIL_MEMODULE, MEMCOUNT);
END; (* BLD_MEMODULENODES *)
PROCEDURE BLD_MEMODULECODE(BUILT_CODE:CODEPTR;
VAR HEAD_MEMODULE:TABLEPTR);
VAR CURMEM:TABLEPTR;

PROCEDURE BLD_A_MEMORY(BUILT_CODE:CODEPTR;
VAR HEAD_MEMODULE:TABLEPTR;
CURMEM:TABLEPTR);
VAR ADDRESS:INTEGER;
CODE_H, CODE_T:TABLEPTR;
SEG:TABLEPTR;

PROCEDURE FORM_MEMORY(BUILT_CODE:CODEPTR;
VAR HEAD_MEMODULE, CURMEM, SEG, CODE_H, CODE_T:TABLEPTR;
VAR ADDRESSS:INTEGER);

PROCEDURE PROCESS_SEG(BUILT_CODE:CODEPTR;
VAR HEAD_MEMODULE, CURMEM, SEG, CODE_H, CODE_T:TABLEPTR;
VAR ADDRESSS:INTEGER);
VAR START, STOP: INTEGER;

PROCEDURE COPYCODE(BUILT_CODE:CODEPTR;
VAR CODE_H, CODE_T:TABLEPTR;
VAR ADDRESSS:INTEGER;
START, STOP: INTEGER);
VAR INSERT, CURTP: TABLEPTR;
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 := ADDRESSS;
ADDRESSS := ADDRESSS+1;
INSERT@KEYCODE := CURCP@KEY;
INSERT@SEQUENTIAL := NIL;
CODE_H := INSERT;
CODE_T := INSERT;
REPEAT
NEW(INsert, CODE);
CURCP := CURCP@SEQ;
INSERT@TAG := CODE;
END
INSERT3. ABS ADDR := CURCPS ADDR;
INSERT3. ADDRESS := ADDRESSS;
ADDRESSS := ADDRESSS + 1;
INSERT3. keycode := CURCPS KEY;
CODE_TT := SEQUENTIAL := INSERT;
CODE_TT := INSERT;
UNTIL (CURCPS ADDR = STOP);
CODE_TT := SEQUENTIAL := NIL;
END;

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

(* ADD_RETURNCODE: ADDS SBR RETURN CODE TO THE* )
(* TAIL SEGMENT OF THE INVOKED SBR * )
(*------------------------------------------* )
PROCEDURE ADD_RETURNCODE(VAR CODE_TT: TBLPTR;
VAR ADDRESSS: INTEGER);
VAR CODE_TT: CODE_TT;
BEGIN
CODE_TT := STO; (*INVSBR CHG 2 STC*)
NEW (INSERT, CODE);
INSERT3. TAG := CODE;
INSERT3. ADDRESS := ADDRESSS;
ADDRESSS := ADDRESSS + 1;
INSERT3. KEYCODE := DISPLAYREGSTORE; (*DISPLAY*)
CODE_TT := SEQUENTIAL := INSERT;
CODE_TT := INSERT;

NEW (INSERT, CODE);
INSERT3. TAG := CODE;
INSERT3. ADDRESS := ADDRESSS;
ADDRESSS := ADDRESSS + 1;
INSERT3. KEYCODE := RCLIND; (* RCL IND *)
CODE_TT := SEQUENTIAL := INSERT;
CODE_TT := INSERT;

NEW (INSERT, CODE);
INSERT3. TAG := CODE;
INSERT3. ADDRESS := ADDRESSS;
ADDRESSS := ADDRESSS + 1;
INSERT3. KEYCODE := MANRTNREG; (* MAN RTN REG *)
CODE_TT := SEQUENTIAL := INSERT;
CODE_TT := INSERT;

NEW (INSERT, CODE);
INSERT3. TAG := CODE;
INSERT3. ADDRESS := ADDRESSS;
ADDRESSS := ADDRESSS + 1;
INSERT3. KEYCODE := OP; (* OP *)
CODE_TT := SEQUENTIAL := INSERT;
CODE_TT := INSERT;

NEW (INSERT, CODE);
INSERT3. TAG := CODE;
INSERT3. ADDRESS := ADDRESSS;
ADDRESSS := ADDRESSS + 1;
INSERT3. KEYCODE := 30; (*MANRTNREG*)
CODE_TT := SEQUENTIAL := INSERT;
CODE_TT := INSERT;

NEW (INSERT, CODE);
INSERT3. TAG := CODE;
INSERT3. ADDRESS := ADDRESSS;
ADDRESSS := ADDRESSS + 1;
INSERT3. KEYCODE := RS; (* RUN/STCP *)
CODE_TT := SEQUENTIAL := INSERT;
CODE_TT := INSERT;

227
PROCEDURE ADDCODE_SBRBRK_MARKINVOKED
(VAR HEAD_MEMODULS,CURMEM: TBLPTR;
SEG: TBLPTR; VAR CODE TT: TBLPTR;
VAR ADDRESSS: INTEGER);
VAR INSERT, SBR, SBRL: TBLPTR;
LBLADDR: INTEGER;

FUNCTION FIND_LBL(BUILT_CODE:CODEPTR;
ADDRESSS:INTEGER):INTEGER;
VAR C:CODEPTR;
BEGIN
C:=BUILT_CODE;
WHILE C<>ADDRESSS DO
C:=C'.SEQ;
FIND_LBL:=C'.SEQ'.KEY;
END;

PROCEDURE GENCODESBR(VAR CODE TT: TBLPTR;
HEAD_MEMODULS, CURMEM, SBR: TBLPTR;
VAR ADDRESSSS:INTEGER);
VAR RELADDR, HUNDREDS, TENS, UNITS: INTEGER;
MEMPTR:=TBLPTR;
BEGIN
MEMPTR:=HEAD_MEMODULS;
WHILE (MEMPTR>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(INSET, CODE);
INSERT. TAG:=CODE;
INSERT. ADDRESS:=ADDRESSS;
ADDRESSS:=ADDRESSS+1;
INSERT. KEYCODE:=FIND_LBL;
IF CODE TT <> NIL THEN
CODETT. SEQUENTIAL:=INSERT;
CODETT: = INSERT;
NEW(INSET, CODE);
INSERT. TAG:=CODE;
INSERT. ADDRESS:=ADDRESSS;
ADDRESSS:=ADDRESSS+1;
INSERT. KEYCODE:=FIND_LBL
(BUILT_CODE, SBR&SBRZ'.START_ADDR);
CODETT. SEQUENTIAL:=INSERT;
CODETT:= INSERT;
NEW (INSERT, CODE):
  INSERT3. TAG := CODE;
  INSERT3. ADDRESS := ADDRESSS;
  ADDRESSS := ADDRESSS + 1;
  INSERT3. KEYCODE := ST0;
  CODE TT9. SEQUENTIAL := INSERT;
  CODE TT := INSERT;

NEW (INSERT, CODE):
  INSERT3. TAG := CODE;
  INSERT3. ADDRESS := ADDRESSS;
  ADDRESSS := ADDRESSS + 1;
  INSERT3. KEYCODE := DISPLAYREGSTORE; (*DISP*)
  CODE TT9. SEQUENTIAL := INSERT;
  CODE TT := INSERT;

NEW (INSERT, CODE):
  INSERT3. TAG := CODE;
  INSERT3. ADDRESS := ADDRESSS;
  ADDRESSS := ADDRESSS + 1;
  INSERT3. KEYCODE := 20+MANRTNREG; (*INCR*)
  CODE TT9. SEQUENTIAL := INSERT;
  CODE TT := INSERT;

NEW (INSERT, CODE):
  INSERT3. TAG := CODE;
  INSERT3. ADDRESS := ADDRESSS;
  ADDRESSS := ADDRESSS + 1;
  INSERT3. KEYCODE := 20+MANRTNREG; (*INCR*)
  CODE TT9. SEQUENTIAL := INSERT;
  CODE TT := INSERT;

NEW (INSERT, CODE):
  INSERT3. TAG := CODE;
  INSERT3. ADDRESS := ADDRESSS;
  ADDRESSS := ADDRESSS + 1;
  INSERT3. KEYCODE := 20+MANRTNREG; (*INCR*)
  CODE TT9. SEQUENTIAL := INSERT;
  CODE TT := INSERT;

NEW (INSERT, CODE):
  INSERT3. TAG := CODE;
  INSERT3. ADDRESS := ADDRESSS;
  ADDRESSS := ADDRESSS + 1;
  INSERT3. KEYCODE := 20+MANRTNREG; (*INCR*)
  CODE TT9. SEQUENTIAL := INSERT;
  CODE TT := INSERT;

NEW (INSERT, CODE):
  INSERT3. TAG := CODE;
  INSERT3. ADDRESS := ADDRESSS;
  ADDRESSS := ADDRESSS + 1;
  INSERT3. KEYCODE := 20+MANRTNREG; (*INCR*)
  CODE TT9. SEQUENTIAL := INSERT;
  CODE TT := INSERT;

NEW (INSERT, CODE):
  INSERT3. TAG := CODE;
  INSERT3. ADDRESS := ADDRESSS;
  ADDRESSS := ADDRESSS + 1;
  INSERT3. KEYCODE := 20+MANRTNREG; (*INCR*)
  CODE TT9. SEQUENTIAL := INSERT;
  CODE TT := INSERT;

NEW (INSERT, CODE):
  INSERT3. TAG := CODE;
  INSERT3. ADDRESS := ADDRESSS;
  ADDRESSS := ADDRESSS + 1;
  INSERT3. KEYCODE := 20+MANRTNREG; (*INCR*)
  CODE TT9. SEQUENTIAL := INSERT;
  CODE TT := INSERT;

NEW (INSERT, CODE):
  INSERT3. TAG := CODE;
  INSERT3. ADDRESS := ADDRESSS;
  ADDRESSS := ADDRESSS + 1;
  INSERT3. KEYCODE := 20+MANRTNREG; (*INCR*)
  CODE TT9. SEQUENTIAL := INSERT;
  CODE TT := INSERT;

NEW (INSERT, CODE):
  INSERT3. TAG := CODE;
  INSERT3. ADDRESS := ADDRESSS;
  ADDRESSS := ADDRESSS + 1;
  INSERT3. KEYCODE := 20+MANRTNREG; (*INCR*)
  CODE TT9. SEQUENTIAL := INSERT;
  CODE TT := INSERT;

NEW (INSERT, CODE):
  INSERT3. TAG := CODE;
  INSERT3. ADDRESS := ADDRESSS;
  ADDRESSS := ADDRESSS + 1;
  INSERT3. KEYCODE := 20+MANRTNREG; (*INCR*)
  CODE TT9. SEQUENTIAL := INSERT;
  CODE TT := INSERT;

NEW (INSERT, CODE):
  INSERT3. TAG := CODE;
  INSERT3. ADDRESS := ADDRESSS;
  ADDRESSS := ADDRESSS + 1;
  INSERT3. KEYCODE := 20+MANRTNREG; (*INCR*)
  CODE TT9. SEQUENTIAL := INSERT;
  CODE TT := INSERT;

NEW (INSERT, CODE):
  INSERT3. TAG := CODE;
  INSERT3. ADDRESS := ADDRESSS;
  ADDRESSS := ADDRESSS + 1;
  INSERT3. KEYCODE := 20+MANRTNREG; (*INCR*)
  CODE TT9. SEQUENTIAL := INSERT;
  CODE TT := INSERT;
END; (* GENCODE_SER *)

BEGIN; (* ADDCODE_SBRBRK_MARKINVOKED *)

END;
(* JUSTIFY_CODE: SETS ALL THE JUMPS AND ADDRS *)

PROCEDURE JUSTIFY_CODE(SEG, CURRENT_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
  ELSE IF F@.KEYCODE IN STEP 2 THEN
    F:=F@.SEQUENTIAL@.SEQUENTIAL
  ELSE IF F@.KEYCODE IN STEP 1 THEN
  THEN F:=F@.SEQUENTIAL@.SEQUENTIAL
  ELSE
    F:=F@.SEQUENTIAL;
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@.JUMPLIST;
            WHILE JMPTR@.JUMP_ADDR<>ABSADDR DO
              ABSDDR DIV 100;
            JMPTR:=JMPTR@.NEXT_FJUMP;
            T@.KEYCODE:=JMPTR@.JUMP_INADDRTC1;
            T@.SEQUENTIAL@.KEYCODE:=JMPTR@.JUMP_INADDRTC2;
          END;
      END;
      ADVANCE_CODE_TBL(F);
    END;
END;

(* JUSTIFY *)
PROCEDURE ADDCODE_FJMP(VAR HEAD_MEMMODULE, CUR_MEM,
SEG_CODE TT: TBLPTR;
VAR ADDRESS: INTEGER;
VAR CUR_INSERT, JUMPTR, MEM_PTR: TBLPTR;
ADDELETO1, ADDDTETO2, MEM_ADDR: INTEGER;
DELTA_ADDR: INTEGER;

PROCEDURE GEN_JUMPocode_SETINTADDR:
(VAR CODE TT: TBLPTR; VAR ADDR: INTEGER;
VAR JUMPR: TBLPTR;
VAR INSERT: TBLPTR;
BEGIN
HUNDREDS, TENS, UNITS, RELADDRESS: INTEGER;
BEGIN
HUNDREDS:=JUMPRJ. JUMP_ADDRTO1;
TENS:=JUMPRJ. JUMP_ADDRTO2 DIV 10;
UNITS:=JUMPRJ. JUMP_ADDRTO2-(10*TENS):

NEW(INSTTTT, CODE):
INSERT@. ADDRESS:= ADDRESS;
RELADDRESS:= ADDRESS:
ADDRESS:= ADDRESS;
INSERT@. KEYCODE:= CODE;
IF CODE TT<> NIL THEN
CODE TT@. SEQUENTIAL:= INSERT;
CODE TT:= INSERT;

NEW(INSTTTT, CODE):
INSERT@. ADDRESS:= ADDRESS;
ADDRESS:= ADDRESS;
INSERT@. KEYCODE:= CODE
* CE *)
CODE TT@. SEQUENTIAL:= INSERT;
CODE TT:= INSERT;

NEW(INSTTTT, CODE):
INSERT@. ADDRESS:= ADDRESS;
ADDRESS:= ADDRESS;
INSERT@. KEYCODE:= CODE
* DECIMAL *)
CODE TT@. SEQUENTIAL:= INSERT;
CODE TT:= INSERT;

232
NEW(INSERT,CODE):  
INSERT3.TAG:=CODE;  
INSERT3.ADDRESS:=ADDRESSS;  
ADDRESSS:=ADDRESSS+1;  
INSERT3.KEYCODE:=HUNDREDS;  
CODE_TTO.SEQUENTIAL:=INSERT;  
CODE_TT:=INSERT;  
(* 10S *)

NEW(INSERT,CODE):  
INSERT3.TAG:=CODE;  
INSERT3.ADDRESS:=ADDRESSS;  
ADDRESSS:=ADDRESSS+1;  
INSERT3.KEYCODE:=TENS;  
CODE_TTO.SEQUENTIAL:=INSERT;  
CODE_TT:=INSERT;  
(* 10S *)

NEW(INSERT,CODE):  
INSERT3.TAG:=CODE;  
INSERT3.ADDRESS:=ADDRESSS;  
ADDRESSS:=ADDRESSS+1;  
INSERT3.KEYCODE:=UNITS;  
CODE_TTO.SEQUENTIAL:=INSERT;  
CODE_TT:=INSERT;  
(* R/S *)

BEGIN           
IF SEG3.P_JUMPLIST<>NIL THEN 
BEGIN               
MEMPTR:=HEAD_MEMODULE; 
WHILE NOT(SEG3.STOP_ADDR+1 = MEMPTR3.  
SEGTLS3.START_ADDR) DO 
MEMPTR:=MEMPTR3.NEXT; 
ADDRT01:=MEMPTR3.SEGTBL3.START_ADDR  
DIV 100; 
ADDRT02:=MEMPTR3.SEGTBL3.START_ADDR  
-(100*ADDRT01); 
ADDRT01:=ADDRT01+MEMPTR3.HIGHOFFSET; 
ADDRT02:=ADDRT02+MEMPTR3.LOWOFFSET; 
MEM_ADDDR:=MEM_ADDDR++; 
NEWJUMPTR:=MEM_ADDDR; 
WITH JUMPTR DO 
BEGIN 
JUMP_ADDRT01:=ADDRT01;  
JUMP_ADDRT02:=ADDRT02;  
MEM_ADDR:=MEM_ADDDR;  
END;  
GEN_JUMPCODE_SETINTADDR  
(CODE_TT,ADDRESSS,JUMPTR);  
DISPOSE(JUMPTR,FWD_JUMP);  
END; 
IF SEG3.F_JUMPLIST<>NIL THEN 
BEGIN 
JUMPTR:=SEG3.F_JUMPLIST; 
WHILE JUMPTR<>NIL DO 
BEGIN 
MEMPTR:=HEAD_MEMODULE; 
233
WHILE NOT((JUMPTR@.JUMPADDRTC >= MEMPTR@.SEGTBL@.START ADDR) AND (JUMPTR@.JUMPADDRTO <= MEMPTR@.SEGTBL@.STOP ADDR)) DO MEMPTR@:=MEMPTR@.NEXT;

DELTA_ADDR:=JUMPTR@.JUMPADDRTO- MEMPTR@.SEGTBL@.START ADDR;
ADDRT01:=DELTA_ADDR DIV 100;
ADDRT02:=DELTA_ADDR-100*ADDRT01;
MEM_ADDRTO:=MEMPTR@.MEMNUN;
JUMPTR@.JUMPADDRTO:=ADDRT01;
JUMPTR@.JUMPADDRTO2:=ADDRT02;
MEM_ADDR:=MEMADDRTO;
GEN JUMP CODE SET INTADDRS
(CODE TT, ADDRESSS, 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, ADDRESSS, START, STOP);
IF (CURMEM@.RETURNCODE_NEEDED) AND (SEG=CURMEM@.SEGTBL) THEN

ADD RETURNCODE(CODE TT, ADDRESSS);
ADD CODE_FJMP(HEAD_MEMODULE, CURMEM, SEG, CODE TT, ADDRESSS);
ADD CODE_SBRBRK_MARKINVOKED(HEAD_MEMODULE, CURMEM, SEG, CODE TT, ADDRESSS);
JUSTIFY CODE(SEG, CURMEM, CODE HH);
END;

END;
(* MARKINVOKED: 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 SBR_BRKINCLUDE<TABLELIST<>NIL DO 
SBR_BRKINCLUDE:=SBR_BRKINCLUDE<TABLELIST;
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, ADDRESSS);
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 
SBRL:=SEG@.SBRLIST;
WHILE SBRL<>NIL DO 
BEGIN 
SBRL:=SBRL@.SBRL;
CASE SBRL@.TAG OF 
SBRRBREAK:
MARKINVOKED (HEAD_MEMODULE, SER);
TABLE:
FORM_MEMORY (BUILT_CODE, HEAD_MEMODULE, CURMEM, SBR, CODE_H, 
CODE_T, ADDRESSS); 
END; (( CASE *)) 
SBRL:=SBRL@.NEXT_SBR;
END;
END; 
END; (** FORM_MEMORY **) 
BEGIN 
SEG@=CURMEM@.SEGTBLS;
ADDRESSS:=0; CODE H:=NIL; CODE T:=NIL;
FORM_MEMORY (BUILT_CODE, HEAD_MEMODULE, CURMEM, SEG,
CODE_H,CODE_T, ADDRESSS); 
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; (** BLD_MEMODULECODE **) 

235
(*-- Procedure for output instructions --*)

PROCEDURE OUTPUT_INSTR((VAR OUTFILE, MESSAGEFILE,
TEMPFILE:TEXT; BUILT CODE:CODEPTR;
HEAD_MEMODULE:TBLPTR; PART_NUM:INTEGER;
PARTITION:REAL;
GOOD_SEGMENT:BOOLEAN));

(*-- Procedure to output good segment instructions --*)

PROCEDURE OUTPUT_GOODSEG((VAR OUTFILE, MESSAGEFILE:
TEXT; HEAD_MEMODULE:TBLPTR; PARTITION:REAL));

(*-- Procedure to output message file --*)

PROCEDURE OUTPUT_MSGF1((VAR OUTFILE, MESSAGEFILE:
TEXT; HEAD_MEMODULE:TBLPTR; PARTNUM:INTEGER;
PARTITION:REAL));

VAR P,T:TBLPTR;
STEP_TYPE:INTEGER;

(*-- Procedure to set step --*)

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;
(*-- Set Step --*)

236
PROCEDURE PRINT_LINE(VAR OUTFILE: TEXT; VAR T: TBLPTR; VAR STEP_TYPE: INTEGER);

PROCEDURE WRITELEBS(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 LABEL (OUTFILE, T.KeyCode);
END;

PROCEDURE WRITENUMS(VAR OUTFILE: TEXT; T:TBLPTR);
BEGIN
  IF STEP_TYPE IN (1, 3, 4, 5, 8, 12.) THEN
  WRITELEBS (OUTFILE, T)
ELSE
  WRITENUMS (OUTFILE, T);
  T := T.Key.SEQUENTIAL;
  STEP_TYPE := STEP_TYPE - 1;
END;

PRINTLN_MSG (OUTFILE, MESSAGEFILE, BAXINSTR);
PRINT_MSGLINE1 (OUTFILE, MESSAGEFILE, RTNRGTOP);
WRITELN (OUTFILE, REGCOUNT: 3);
PRINT_MSGLINE1 (OUTFILE, MESSAGEFILE, STOINRG);
WRITELN (OUTFILE, MANRTNREG: 3);
PRINT_MSGLINE1 (OUTFILE, MESSAGEFILE, PGMPARTIS);
WRITELN (OUTFILE, PARTITION: 4: 2);
WRITELN (OUTFILE);
PRINT_MSGLINE1 (OUTFILE, MESSAGEFILE, PARTNUMIS);
WRITELN (OUTFILE, PARTNUMAT: 1);
WRITELN (OUTFILE);
P := HEAD MEMODUL;
WHILE P<>NIL DO
BEGIN
  T := T.CodeLIST;
  STEP_TYPE := 0;
  WRITELN (OUTFILE); WRITELN (OUTFILE);
END;
BEGIN
IF T$ ADDRESS=240 THEN
BEGIN
WRITELN(OUTFILE):WRITELN(OUTFILE)
PRINTLN (OUTFILE,MESSAGEFILE,SIDE2);
END;
IF T$ ADDRESS = 480 THEN
BEGIN
WRITELN(OUTFILE):WRITELN(OUTFILE)
PRINTLN_MSG (OUTFILE,MESSAGEFILE,CARD[2]);
PRINTLN_MSG (OUTFILE,MESSAGEFILE,SIDE1);
END;
WHILE T$>NIL DO
BEGIN
IF T$ ADDRESS=240 THEN
BEGIN
WRITELN(OUTFILE):WRITELN(OUTFILE)
PRINTLN_MSG (OUTFILE,MESSAGEFILE,SIDE2);
END;
END
END;
END;

(* OUTPUT_MSGF1 *)

(* OUTPUT_MSGF2: OUTPUTS SPECIFIC PROMPTS AND *)
(* SPECIAL PROGRAM INSTRUCTIONS *)
PROCEDURE OUTPUT_MSGF2 (VAR OUTFILE, MESSAGEFILE: TEXT;
  HEAD_MEMODULE: TBLPTR);
VAR SBRL, SEP, SM, P, SEG: TBLPTR;
  IS_SBRERK: BOOLEAN;
BEGIN
P:=HEAD_MEMODULE:
PRINTLN_MSG (OUTFILE, MESSAGEFILE,SPECIFICS):
WHILE P<>NIL DO
BEGIN
SEG:=P$.SEGTLBS;
P:=P$.P JUMP LIST;
SBRL:=SEG$.SBRLIST;
WRITELN(OUTFILE):
PRINT_MSGLINE1 (OUTFILE,MESSAGEFILE,MOD PROMPTS):
WRITELN (OUTFILE,PA$.MEMNUM:1):
PRINTLN_MSG (OUTFILE,MESSAGEFILE,PPWDJ);
IF P= NIL THEN
PRINLN_MSG (OUTFILE,MESSAGEFILE,None)
ELSE
BEGIN
WHILE P<>NIL DO
BEGIN
PRINT_MSGLINE1 (OUTFILE,MESSAGEFILE,ASTER);
WRITE (OUTFILE,PA$.MEM_ADDR:1,.*)
WRITE_LEADZERO (OUTFILE,(PA$.JUMP_ADDRTO1*100
  PA$.JUMP_ADDRTO2),3);
WRITELN (OUTFILE):
P:=P$.NEXT_PJUMP;
END;
PRINTLN_MSG (OUTFILE,MESSAGEFILE,PSBRINV);
IF SPIF$=NIL THEN
PRINTLN_MSG (OUTFILE,MESSAGEFILE,None)
ELSE
BEGIN
IS_SBRBRK:=FALSE;
WHILE SBR<>NIL DO
BEGIN
SBR:=SRL@.SBR;
IF SRL@.TAG = SBRBREAK THEN
BEGIN
SM:=HEAD_MEMODULE;
SBR:=SRL@.SBRZ;
WHILE SM@.SEGTBLS<>SBR DO
SM:=SM@.NEXT;
IS_SBRBRK:=TRUE;
PRINT_MSGLINE1 (OUTFILE, MESSAGEFILE, ASTER);
WRITELN (OUTFILE, SM@.MEMNUM:1,'.000');
END;
SRL@:=SRL@.SBR3,NEXT;
END;
IF NOT IS_SBRBRK THEN
PRINTLN_MSG (OUTFILE, MESSAGEFILE, NONE)
END;
PRINTLN_MSG (OUTFILE, MESSAGEFILE, PMANRTN);
IF PG.RETURNCODE NEEDED THEN
PRINTLN_MSG (OUTFILE, MESSAGEFILE, YES)
ELSE
PRINTLN_MSG (OUTFILE, MESSAGEFILE, NONE);
PRINTLN_MSG (OUTFILE, MESSAGEFILE, SEQ);
IF SEGS7TABL LIST <> NIL THEN
BEGIN
PRINT_MSGLINE1 (OUTFILE, MESSAGEFILE, ASTER);
WRITELN (OUTFILE, P3.NEXT@.MEMNUM:1,'.000');
END
ELSE
PRINTLN_MSG (OUTFILE, MESSAGEFILE, NONE);
P:=P3.NEXT;
END;
WRITELN (OUTFILE);WRITELN (OUTFILE);
PRINTLN_MSG (OUTFILE, SCRATCH, DATAREAD);
PRINTLN_MSG (OUTFILE, SCRATCH, IDGMA);
PRINTLN_MSG (OUTFILE);WRITELN (OUTFILE):
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_BADSEG: 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, unsegcodelbl));
  writeln(outfile); writeln(outfile);
  print_code_list(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_memodule_nodes(sectbl, head_memodule);
  bld_memodule_code(built_code, head_memodule);
  output_instr(outfile, messagefile, tempfile, built_code, head_memodule, part_num, partition, good_segment);
end;

(* INSTRUCTIONS *)
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(SRATCH, BUILT_CODE, BUILT_CODE_COUNT);
  SETJMF5(BUILT_CODE);
  BUIL SEGTLBL(BUILT_CODE, SEGTBL, LIMIT, BUILT_CODE_COUNT);
  COALESCE(SEGTBL@.TABLELIST, LIMIT, GOOD SEGMENT
             SBRINVNEST);
  WRITE LN(TEMPFILE, '9'); (* CLOSFS 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
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 REFERRED 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:

<table>
<thead>
<tr>
<th>Partition</th>
<th>Registers</th>
<th>Program Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>719.29</td>
<td>479.59</td>
</tr>
<tr>
<td>4</td>
<td>719.29</td>
<td>239.69</td>
</tr>
</tbody>
</table>

**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 REFERENCE 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.

<table>
<thead>
<tr>
<th>VIRTUAL BANK</th>
<th>ACTUAL BANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCDULE #</td>
<td>BANK1</td>
</tr>
<tr>
<td>CARD1 SIDE1</td>
<td></td>
</tr>
<tr>
<td>MCDULE #</td>
<td>BANK2</td>
</tr>
<tr>
<td>CARD1 SIDE2</td>
<td></td>
</tr>
<tr>
<td>MCDULE #</td>
<td>BANK3</td>
</tr>
<tr>
<td>CARD2 SIDE1</td>
<td></td>
</tr>
</tbody>
</table>

==TI-59 LISTING==

$81
TI-59 PROGRAM SPECIFIC INSTRUCTIONS:

* THE FOLLOWING INFORMATION WILL TELL YOU HOW TO RUN YOUR PROGRAM.

* YCU 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:

    \[
    \text{X} \quad \text{2ND} \quad \text{08} \quad \text{17}
    \]

  * X IS THE PARTITION NUMBER GIVEN IN THE LISTING OF YOUR PROGRAM.

* WHEN TO PARTITION THE MEMORY
  * CNCE BEFORE READING IN CARDS.
  * CNCE BEFORE MANUALLY ENTERING PROGRAM IN ORDER TO WRITE TO CARDS.

* HOW TO START AND RUN YOUR PROGRAM
  * TURN ON 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 RETURN CONTROL STACK WITH THE FOLLOWING KEYSTROKES:

```
XX
STO
08
```

WHERE XX IS THE MANUAL RETURN REGISTER STACK TOF. (THIS IS GIVEN WITH THE PROGRAM LISTING NEAR THE PARTITION INFORMATION.)

* PRESS "A" TO START.

* 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 IS A MODULE NUMBER.

* ACTIONS:
  * LOAD ALL MODULE X CARDS.
  * PRESS FOLLOWING SEQUENCE TO INITIALIZE:
    RCL
    00
    INV
    SRC

  * 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:

$82
$83
$84

END BAX59 SEGMENTATION/INSTRUCTION: VERSION 1.0

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 SBR 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.

- **BELOW 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. SBR NESTING LEVEL DIAGNOSTICS ARE GIVEN FOR INVOKED ROUTINE DEFINITION.**

---

**EAX59 VERSION 1.0**

UNSEGMENTED ABSOLUTE COMPILED TI59 CODE FOLLOWS

---

**$84**

---

**$5**

---

**$6**

---

**$7**

---

**$9**

---

**$9**

---

**$10**

---

**$11**

---

**$12**

---

**$13**

---

**$14**

---

**$15**

---

**$16**

---

**$17**

---

**$18**

---

**$19**

---

**$20**

---

**$21**

---

**$22**

---

**$23**

---

**$24**

---

**$25**

---

**$26**

---

**$27**

---

**$28**

---

**$29**

---

**$30**

---

**$31**

---

**$32**

---

**$33**

---

**$34**

---

**$35**

---

**$36**

---

**$37**

---

**$38**

---

**$39**

---

**$40**

---

**$41**

---

**$42**

---

**$43**

---

**$44**

---

**$45**

---

**$46**

---

**$47**

---

**$48**

---

**$49**

---

**$50**

---

**$51**

---

**$52**

---

**$53**

---

**$54**

---

**$55**

---

**$56**

---

**$57**

---

**$58**

---

**$59**

---

**$60**

---

**$61**

---

**$62**

---

**$63**

---

**$64**

---

**$65**

---

**$66**

---

**$67**

---

**$68**

---

**$69**

---

**$70**

---

**$71**

---

**$72**

---

**$73**

---

**$74**

---

**$75**

---

**$76**

---

**$77**

---

**$78**

---

**$79**

---

**$80**

---

**$81**

---

**$82**

---

**$83**

---

**$84**

---

**$85**

---

**$86**

---

**$87**

---

**$88**

---

**$89**

---

**$90**

---

**$91**

---

**$92**

---

**$93**

---

**$94**

---

**$95**

---

**$96**

---

**$97**

---

**$98**

---

**$99**

---

**3**

---

**4**

---

**5**

---

**6**

---

**7**

---

**8**

---

**9**

---

**1**

---

**2**

---

**3**

---

**4**

---

**5**

---

**6**

---

**7**

---

**8**

---

**9**

---

**10**

---

**11**

---

**12**

---

**13**

---

**14**

---

**15**

---

**16**

---

**17**

---

**18**

---

**19**

---

**20**

---

**21**

---

**22**

---

**23**

---

**24**

---

**25**

---

**26**

---

**27**

---

**28**

---

**29**

---

**30**

---

**31**

---

**32**

---

**33**

---

**34**

---

**35**

---

**36**

---

**37**

---

**38**

---

**39**

---

**40**

---

**41**

---

**42**

---

**43**

---

**44**

---

**45**

---

**46**

---

**47**

---

**48**

---

**49**

---

**50**

---

**51**

---

**52**

---

**53**

---

**54**

---

**55**

---

**56**

---

**57**

---

**58**

---

**59**

---

**60**

---

**61**

---

**62**

---

**63**

---

**64**

---

**65**

---

**66**

---

**67**

---

**68**

---

**69**

---

**70**

---

**71**

---

**72**

---

**73**

---

**74**

---

**75**

---

**76**

---

**77**

---

**78**

---

**79**

---

**80**

---

**81**

---

**82**

---

**83**

---

**84**

---

**85**

---

**86**

---

**87**

---

**88**

---

**89**

---

**90**

---

**91**

---

**92**

---

**93**

---

**94**

---

**95**

---

**96**

---

**97**

---

**98**

---

**99**

---

---
* EXPECTED PROMPTS FOR MODULE # $ 

* NONE 

* $ 

* YES 

* MANUAL RETURN REGISTER TOP IS $ 

STORE IN REGISTER: $ 

* PROGRAM PARTITION IS $ 

* PARTITION NUMBER IS $ 

* MODULE # $ 

CAR #1 

CARD #2 

SIDE #1 

SIDE #2
APPENDIX K

ARTILLERY TEST PROGRAM SOURCE CODE

00005 OPTION 0 5
00010 REM **********************************************************************
00011 REM
00012 REM BAX59 TEST PROGRAM NUMBER1
00013 REM
00014 REM THIS TEST PROGRAM IS A TRANSLATION OF THE PROGRAM
00015 REM USED BY THE FIELD ARTILLERY IN THE COMPUTATION
00016 REM OF FIRING DATA FOR THEIR GUNS. THE ORIGINAL
00017 REM PROGRAM WAS WRITTEN FOR THE TI-59 CALCULATOR.
00018 REM THIS TEST WAS CHOSEN NOT ONLY TO EVALUATE THE
00019 REM THE COMPILER AND SEGMENTOR BUT TO COMPARE THE
00020 REM THE RELATIVE EFFICIENCY OF THE TRANSLATED WEASIC
00021 REM PROGRAM WITH THAT OF A HUMAN CODED PROGRAM. BOTH
00022 REM PROGRAMS ACCOMPLISH THE SAME TASK.
00023 REM
00024 REM **************************************************
00025 REM
00055 REM **************** DATA SECTION M109 ****************
00056 REM
00060 REM *CHARGE CONSTANTS M109A1 SELF PROPELLED
00061 REM *CHARGE 4
00062 00120 DATA -.0133670, 21.2691, -105.7
00063 00140 DATA -.00001499, -.06630, -.41
00064 00150 DATA .77, .01314, -.00001720
00065 00160 REM *CHARGE 5
00066 00170 DATA -.0149331, 24.3439, 64.7
00067 00180 DATA -.00001420, .07069, .06
00068 00190 DATA 1.26, .01508, -.00001678
00069 00200 REM *CHARGE 6
00070 00210 DATA -.0173835, 29.8741, 2255.2
00071 00220 DATA -.00001668, .09272, 5.74
00072 00230 DATA 1.36, .02891, -.00001410
00073 00240 REM
00074 00250 REM *M109 MAX RANGE OF CURVE FIT BY CHARGE
00075 00260 DATA 5700, 7000, 10800, 17600
00076 00270 REM
00077 00280 REM *M109 HIGH ANGLE CROSS OVER POINT MILS
00078 00290 DATA 715
00080 REM
00081 REM *BATTERY DATA/ BTRYE, BTRYN, BTRYA, BTRYL
00082 00340 DATA 0.0, 0.0, 0.0, 0.0
00083 00360 REM
00084 REM *REGISTRATION DATA/RNGK, DFCOR
00085 00380 REM
00087 REM *TARGET DATA/OBSERVER LOCATION (DUAL MEANING)
00088 00420 DATA 4000, 4000, 10
00089 00430 REM
00090 REM *OBSERVER DATA
00091 00450 DATA 4000, -400, 10
00092 00470 REM
00093 REM *SPECIFIC CORRECTION FACTORS DATA
00094 00471 DATA 1018, 5924, 1600, 3200
00095 00472 REM
00096 00473 REM
**** VARIABLE READ INITIALIZATION ****

**M109 ELLISTIC CONSTANTS BY CHARGE**

*CHARGE 4*
A24, A14, A04
C24, C14, C04
B04, B14, B24

*CHARGE 5*
A25, A15, A05
C25, C15, C05
B05, B15, B25

*CHARGE 7*
A27, A17, A07
C27, C17, C07
B07, B17, B27

*CHARGE 8*
A28, A18, A08
C28, C18, C08
B08, B18, B28

**M109 MAX RANGE OF CURVE FIT VARIABLES**
CHG4MAX, CHG5MAX, CHG7MAX, CHG8MAX

**M109 HIGH ANGLE CROSS OVER VARIABLE**
HACROSS

*BATTERY VARIABLES*
STRY, STRYN, STRYA, STRYL

*REGISTRATION VARIABLES*
RGK, DFCOR

*TARGET VARIABLES OR OBSERVER INIT LOCATION*
GRIDE, GRIDN, GRIDA

*OBSERVER VARIABLES*
OT, LATDEV, RGDEV

*SPECIFIC CORRECTION FACTORS VARIABLES*
MILRAD, ROTCOR, REFDEF

****************** MAIN PROGRAM BEGINS ******************

**START**

**COMPUTE TARGET GRID**
GOSUE 1050

**COMPUTE GUN RANGE, AZIMUTH**
GOSUE 1130

**COMPUTE FIRING DATA**
GOSUE 1240

**STOP**

****************** MAIN STOP ******************

****************** SUBROUTINES ******************

**COMPUTE NEW TARGET GRID FROM SHIFTS**

**START**

GRIDE = GRIDE + (RGDEV * COS((ROTCOR - OT) / MILRAD))

GRIDN = GRIDN + (RGDEV * SIN((ROTCOR - OT) / MILRAD))

**RETURN**

******************
START
TGTRG = SQR((GRIDX-TRYX)**2 + (GRIDN-BTRYN)**2)

IF GRIDE >= (GRIDN-BTRYN)/TGTRG * MILRAD
TGTZ = ROCR - TGTZ
ELSE
TGTZ = 3 * ROCR + TGTZ
ENDIF
RETURN

START
IF TGTRG <= CHG4MAX
INVCKE = FN_PD(A24,A14,A04,C24,C14,C04,
B24,E14,B04)
ELSEIF TGTRG <= CHG5MAX
INVCKE = FN_PD(A25,A15,A05,C25,C15,C05,
B25,E15,B05)
ELSEIF TGTRG <= CHG7MAX
INVCKE = FN_PD(A27,A17,A07,C27,C17,C07,
B27,E17,B07)
ELSEIF TGTRG <= CHG8MAX
INVCKE = FN_PD(A28,A18,A08,C28,C18,C08,
B28,E18,B08)
ELSE
PRINT TGTRG
ENDIF
RETURN

START
DEP = FN_PD(A2,A1,A0,C2,C1,C0,B2,B1,B0)
EL = (-A1+SQR(A1**2-4*A2*(A0-TGTRG*BGRK)))/(2*A2)
IF EL > HACROSS
PRINT TGTZ,TGTRG
ELSE
PRINT C0+C1*EL+C2*EL**2
PRINT REFDEF+DCOR+(BTRYL-TGTZ)+
(B0+B1*EL+B2*EL**2)
PRINT EL+((GRIDX-BTRYA+20)/TGTRG*1000)
ENDIF
FNEND
END EAX59 TEST PROGRAM NUMBER ONE

REM
*** COMPUTE GUN RANGE, AZIMUTH ******************
START
TGTRG = SQR((GRIDX-TRYX)**2 + (GRIDN-BTRYN)**2)
TGTZ = ASIN((GRIDN-BTRYN)/TGTRG) * MILRAD
IF GRIDE >= TRYX
TGTZ = ROCR - TGTZ
ELSE
TGTZ = 3 * ROCR + TGTZ
ENDIF
RETURN

REM
*** FINING DATA COMPUTATION ROUTINE ******************
START
IF TGTRG <= CHG4MAX
INVCKE = FN_PD(A24,A14,A04,C24,C14,C04,
B24,E14,B04)
ELSEIF TGTRG <= CHG5MAX
INVCKE = FN_PD(A25,A15,A05,C25,C15,C05,
B25,E15,B05)
ELSEIF TGTRG <= CHG7MAX
INVCKE = FN_PD(A27,A17,A07,C27,C17,C07,
B27,E17,B07)
ELSEIF TGTRG <= CHG8MAX
INVCKE = FN_PD(A28,A18,A08,C28,C18,C08,
B28,E18,B08)
ELSE
PRINT TGTRG
ENDIF
RETURN

REM
*** FINING DATA COMPUTATION FUNCTION ******************
START
DEP = FN_PD(A2,A1,A0,C2,C1,C0,B2,B1,B0)
EL = (-A1+SQR(A1**2-4*A2*(A0-TGTRG*BGRK)))/(2*A2)
IF EL > HACROSS
PRINT TGTZ,TGTRG
ELSE
PRINT C0+C1*EL+C2*EL**2
PRINT REFDEF+DCOR+(BTRYL-TGTZ)+
(B0+B1*EL+B2*EL**2)
PRINT EL+((GRIDX-BTRYA+20)/TGTRG*1000)
ENDIF
FNEND
END EAX59 TEST PROGRAM NUMBER ONE

REM

250
APPENDIX L

TEST PROGRAM LISTING FILE (LISTF)

=================================================================================================================

W BASIC PROGRAM LISTING

=================================================================================================================

00005 OPTION 0 5
00010 REM   ******************************************
00011 REM   BAX59 TEST PROGRAM NUMBER 1
00012 REM   ******************************************
00013 REM   *THIS TEST PROGRAM IS AN ADAPTATION OF THE PROGRAM
00014 REM   USED BY THE FIELD ARTILLERY IN THE COMPUTATION
00015 REM   OF FIRING DATA FOR THEIR GUNS. THE ORIGINAL
00016 REM   PROGRAM WAS WRITTEN FOR THE TI-59 CALCULATOR.
00017 REM   THIS TEST WAS CHOSEN TO NOT ONLY EVALUATE THE
00018 REM   THE COMPILER AND SEGMENTOR BUT TO COMPARE THE
00019 REM   THE RELATIVE EFFICIENCY OF THE TRANSLATED W BASIC
00020 REM   PROGRAM WITH THAT OF A HUMAN CODED PROGRAM. BOTH
00021 REM   PROGRAMS ACCOMPLISH THE SAME TASK.

00022 REM   ******************************************

00026 REM   ************** DATA SECTION M109 **************

00036 REM   *CHARGE CONSTANTS M109A1 SELF PROPELLED
00037 00040 REM   *CHARGE 4
00041 00044 REM   DATA
00045 00048 REM   -.0133670, 21.2691, -105.7
00050 00054 REM   -00001499, .06630, -.41
00059 00063 REM   .77, .01314, .00001720

00068 REM   *CHARGE 5
00072 00076 REM   DATA
00080 00084 REM   -.0149331, 24.3439, 64.7
00089 00093 REM   -00001420, .07069, .06
00098 00102 REM   1.26, .01508, .00001678

00107 REM   *CHARGE 7
00111 00115 REM   DATA
00119 00123 REM   -.0173835, 29.8741, 2255.2
00128 00132 REM   -00001663, .08472, 5.74
00137 00141 REM   1.3, .02713, .00001306

00146 REM   *CHARGE 8
00150 00154 REM   DATA
00159 00163 REM   -.0182137, 32.3731, 4107.4
00168 00172 REM   -00001665, .09272, .74
00177 00181 REM   1.35, .02891, .00001410

00186 REM   **M109 MAX RANGE OF CURVE FIT BY CHARGE
00190 00194 REM   5700, 7000, 10800, 17500

00203 REM   **M109 HIGH ANGLE CROSS OVER POINT MILS
00207 00211 REM   715

00216 REM   **BATTERY DATA/ BTRYE,BTRYN,BTRYA,BTRL
00220 00224 REM   0, 0, 0, 800

00233 REM   **REGISTRATION DATA/RNGK,DFCOR
00237 00241 REM   1.0, 0

00246 REM   **TARGET DATA/OBSERVER LOCATION (DUAL MEANING)
00250 00254 REM   4000, 4000, 10
00259 00263 REM   4000, -400, 10

00268 REM   **OBSERVER DATA
00272 00276 REM   4000, -400, 10

251
*SPECIFIC CORRECTION FACTORS DATA

DATA 1018.5924,1600,3200

***VARIABLE READ INITIALIZATION *****

*M109 BALLISTIC CONSTANTS BY CHARGE

*CHARGE 4

REM
00470 REM
00471 REM
00472 REM
00473 REM
00480 REM
00490 REM
00500 REM
00510 READ A24,A14,A04
00520 READ C24,C14,C04
00530 READ B04,E14,B24
00540 REM
00550 READ A25,A15,A05
00560 READ C25,C15,C05
00570 READ B05,B15,B25
00580 REM
00590 READ A27,A17,A07
00600 READ C27,C17,C07
00610 READ B07,B17,B27
00620 REM
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 HACRSS

00720 REM
00730 REM
*BATTERY VARIABLES

00740 READ ETYF,ETRN,STYF,STYL

00750 REM
00760 REM
*REGISTRATION VARIABLES

00770 READ RGK,LFCOR

00780 REM
00790 REM
*TARGET VARIABLES OR OBSERVER INIT LOCATION

00800 READ GRID,EGRIDN,GRIDA

00810 REM
00820 REM
*OBSERVER VARIABLES

00830 READ OT,LATDEV,RGDEV

00840 REM
00850 REM
*M109 SPECIFIC CORRECTION FACTORS VARIABLES

00860 READ MIISEL,ROTCOR,REFDEF

00870 REM
00880 REM
*********** MAIN PROGRAM BEGINS ***********

00890 REM
00900 REM
START

00910 REM
00920 REM
*COMPUTE TARGET GRID

00930 REM
00940 REM
*COMPUTE GUN RANGE,AZIMUTH

00950 REM
00960 REM
*COMPUTE FIRING DATA

00970 REM
00980 REM
00990 REM
01000 REM
STOP

01010 REM
01020 REM
01030 REM
01040 REM
01050 REM
01060 REM
01070 REM
01080 REM
01090 REM

*********** MAIN STOP ***********

*********** SUBROUTINES ***********

** COMPUTE NEW TARGET GRID FROM SHIFTS **********

01070 REM
01080 REM
01090 REM

GRIDN = GRIDN + (RGDEV*SIN((ROTCOR-OT)/MILRAD) - LATDEV*COS((ROTCOR-OT)/MILRAD))

GRIDE = GRIDE + (RGDEV*COS((ROTCOR-OT)/MILRAD) + LATDEV*SIN((ROTCOR-OT)/MILRAD))

252
REM **** COMPUTE GUN RANGE, AZIMUTH ***************
01135 START
01140 TGTRG = SQRT((GRID-E-BTRY)**2 + (GRIDN-BTRYN)**2)
01145 &
01150 TGTAZ = ASIN((GRIDN-BTRY)/TGTRG)*MILRAD
01155 IF GRID > BTRY
01160 TGTAZ = ROTCOR - TGTAZ
01165 ELSE
01170 TGTAZ = 3*ROTCOR + TGTAZ
01175 ENDIF
01180 RETURN
01185 REM **************************************************

REM **** FIRING DATA COMPUTATION ROUTINE ***************
01230 START
01240 IF TGTRG <= CHG4MAX
01245 INVCK2 = FN FD(A24,A14,A04,C24,C14,C04)
01250 ELSEIF TGTRG <= CHG5MAX
01255 INVCK2 = FN FD(A25,A15,A05,C25,C15,C05)
01260 ELSEIF TGTRG <= CHG7MAX
01265 INVCK2 = FN FD(A27,A17,A07,C27,C17,C07)
01270 ELSEIF TGTRG <= CHG8MAX
01275 INVCK2 = FN FD(A28,A18,A08,C28,C18,C08)
01280 ELSE
01285 PRINT TGTRG
01290 ENDIF
01295 RETURN
01300 REM **************************************************

REM *** FIRING DATA COMPUTATION FUNCTION ***************
01400 START
01410 DEF FN FD(A2,A1,A0,C2,C1,C0,B2,B1,B0)
01415 EL = (-A1+SQR(A1**2-(4*A2*(A0-TGTRG*EGK)**2)))/(2*A2)
01420 IF EL > HACROSS
01425 PRINT TGTAZ,TGTRG
01430 ELSE
01435 PRINT EL+C1*EL+C2*EL**2
01440 ENDIF
01445 PRINT TGTAZ, TGTRG
01450 IF EL > HACROSS
01455 PRINT EL+C1*EL+C2*EL**2
01460 ENDIF
01465 PRINT EL+(GRID-BTRYA+20)/TGTRG*1000
01470 END
01475 REM **************************************************

REM END FAX59 TEST PROGRAM NUMBER ONE
01530 REM **************************************************

COMPILATION SUMMARY

FATAL ERRORS: 0
WARNING MSGS: 0
TOTAL BEGIN AVAILABLE REGISTER: 11
TOTAL REGISTERS USED = 70
TOTAL LABELS USED = 5

-------------------------------------------------------------------------
COMPILATION TERMINATES
-------------------------------------------------------------------------

-------------------------------------------------------------------------
TI-59 CCE TRANSLATED FROM WBASIC
UNSSEGMENTED
-------------------------------------------------------------------------

<table>
<thead>
<tr>
<th>ADDR</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>76</td>
</tr>
<tr>
<td>001</td>
<td>11</td>
</tr>
<tr>
<td>002</td>
<td>11 E</td>
</tr>
<tr>
<td>003</td>
<td>12</td>
</tr>
<tr>
<td>004</td>
<td>68</td>
</tr>
<tr>
<td>005</td>
<td>71</td>
</tr>
<tr>
<td>006</td>
<td>13</td>
</tr>
<tr>
<td>007</td>
<td>68</td>
</tr>
<tr>
<td>008</td>
<td>71</td>
</tr>
<tr>
<td>009</td>
<td>14</td>
</tr>
<tr>
<td>010</td>
<td>68</td>
</tr>
<tr>
<td>011</td>
<td>22</td>
</tr>
<tr>
<td>012</td>
<td>98</td>
</tr>
<tr>
<td>013</td>
<td>76</td>
</tr>
<tr>
<td>014</td>
<td>76</td>
</tr>
<tr>
<td>015</td>
<td>76</td>
</tr>
<tr>
<td>016</td>
<td>12</td>
</tr>
<tr>
<td>017</td>
<td>23</td>
</tr>
<tr>
<td>018</td>
<td>59</td>
</tr>
<tr>
<td>019</td>
<td>59</td>
</tr>
<tr>
<td>020</td>
<td>59</td>
</tr>
<tr>
<td>021</td>
<td>59</td>
</tr>
<tr>
<td>022</td>
<td>59</td>
</tr>
<tr>
<td>023</td>
<td>59</td>
</tr>
<tr>
<td>024</td>
<td>59</td>
</tr>
<tr>
<td>025</td>
<td>59</td>
</tr>
<tr>
<td>026</td>
<td>59</td>
</tr>
<tr>
<td>027</td>
<td>59</td>
</tr>
<tr>
<td>028</td>
<td>59</td>
</tr>
<tr>
<td>029</td>
<td>59</td>
</tr>
<tr>
<td>030</td>
<td>59</td>
</tr>
<tr>
<td>031</td>
<td>59</td>
</tr>
<tr>
<td>032</td>
<td>59</td>
</tr>
<tr>
<td>033</td>
<td>59</td>
</tr>
<tr>
<td>034</td>
<td>59</td>
</tr>
<tr>
<td>035</td>
<td>59</td>
</tr>
<tr>
<td>036</td>
<td>59</td>
</tr>
<tr>
<td>037</td>
<td>59</td>
</tr>
<tr>
<td>038</td>
<td>59</td>
</tr>
<tr>
<td>039</td>
<td>59</td>
</tr>
<tr>
<td>040</td>
<td>59</td>
</tr>
<tr>
<td>041</td>
<td>59</td>
</tr>
<tr>
<td>042</td>
<td>59</td>
</tr>
<tr>
<td>043</td>
<td>59</td>
</tr>
<tr>
<td>044</td>
<td>59</td>
</tr>
<tr>
<td>045</td>
<td>59</td>
</tr>
<tr>
<td>046</td>
<td>59</td>
</tr>
<tr>
<td>047</td>
<td>59</td>
</tr>
<tr>
<td>048</td>
<td>59</td>
</tr>
<tr>
<td>049</td>
<td>59</td>
</tr>
<tr>
<td>050</td>
<td>59</td>
</tr>
<tr>
<td>051</td>
<td>59</td>
</tr>
<tr>
<td>052</td>
<td>59</td>
</tr>
<tr>
<td>053</td>
<td>59</td>
</tr>
<tr>
<td>054</td>
<td>59</td>
</tr>
<tr>
<td>055</td>
<td>59</td>
</tr>
</tbody>
</table>

BEGIN TI-59 CODE.
194 77 2ND X≥T
195 02 02
196 44 44
197 43 RCL
198 11 STO 71
199 43 STO 72
200 72 RCL
201 43 STO 73
202 73 RCL
203 43 STO 74
204 74 RCL
205 43 STO 75
206 75 RCL
207 43 STO 76
208 76 RCL
209 43 STO 77
210 77 RCL
211 42 STO 78
212 78 RCL
213 42 STO 79
214 79 RCL
215 13 INV
216 16 2ND X≥T
217 76 STO 76
218 76 RCL
219 13 STO 77
220 13 RCL
221 43 STO 77
222 77 RCL
223 43 STO 78
224 78 RCL
225 43 STO 79
226 79 X≥T
227 18 STO 78
228 18 RCL
229 43 STO 79
230 79 RCL
231 17 STO 79
232 79 STO 79
233 79 RCL
234 42 SBR E
235 15 54
236 54 STO 69
237 69 GTO 04
238 42 RCL 67
239 67 X<=>T
240 67 RCL
241 67 STO 69
242 69 RCL
243 69 STO 69
244 69 RCL
245 69 STO 69
246 69 RCL
247 69 STO 69
248 69 RCL
249 69 STO 69
250 69 RCL
251 69 STO 69
252 69 RCL
253 69 STO 69
254 69 RCL
255 69 STO 69
256 69 RCL
257 69 STO 69
258 69 RCL
259 69 STO 69
260 69 RCL
261 69 STO 69
262 69 RCL

257
263  22  STO
265  42  RCL
266  73  RCL
267  43  STO
268  23  RCL
269  42  RCL
270  74  STO
271  24  RCL
272  42  STO
273  75  RCL
274  25  STO
275  42  RCL
276  76  STO
277  76  RCL
278  26  STO
279  42  RCL
280  77  STO
281  77  RCL
282  27  STO
283  42  RCL
284  28  STO
285  42  RCL
286  29  STO
287  42  RCL
288  30  STO
289  42  RCL
290  31  STO
291  42  RCL
292  32  STO
293  42  RCL
294  33  STO
295  42  RCL
296  34  STO
297  42  RCL
298  35  STO
299  42  RCL
300  36  STO
301  42  RCL
302  37  STO
303  42  RCL
304  49  X<>T
305  42  RCL
306  49  X<>T
307  42  RCL
308  49  X<>T
309  42  RCL
310  49  X<>T
311  42  RCL
312  49  X<>T
313  42  RCL
314  49  X<>T
315  42  RCL
316  49  X<>T
317  42  RCL
318  49  X<>T
319  42  RCL
320  49  X<>T
321  42  RCL
322  49  X<>T
323  42  RCL
324  49  X<>T
325  42  RCL
326  49  X<>T
327  42  RCL
328  49  X<>T
329  42  RCL
330  49  X<>T
331  42  RCL
258
<table>
<thead>
<tr>
<th>BUCKET</th>
<th>CONTENTS</th>
<th>REG</th>
<th>TYP</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>REFDEF</td>
<td>66</td>
<td>GLOBAL VAR</td>
</tr>
<tr>
<td>05</td>
<td>LCG10</td>
<td></td>
<td>QUICK FN</td>
</tr>
<tr>
<td>06</td>
<td>GRIDA</td>
<td>60</td>
<td>GLOBAL VAR</td>
</tr>
<tr>
<td>08</td>
<td>EL</td>
<td>80</td>
<td>GLOBAL VAR</td>
</tr>
<tr>
<td>10</td>
<td>GRIDN</td>
<td>58</td>
<td>GLOBAL VAR</td>
</tr>
<tr>
<td>13</td>
<td>FP</td>
<td></td>
<td>QUICK FN</td>
</tr>
<tr>
<td>14</td>
<td>ABS</td>
<td></td>
<td>QUICK FN</td>
</tr>
<tr>
<td>15</td>
<td>FN_PD</td>
<td>70</td>
<td>PARAMETER FN</td>
</tr>
<tr>
<td>16</td>
<td>PI</td>
<td></td>
<td>CONSTANT</td>
</tr>
<tr>
<td>17</td>
<td>IP</td>
<td></td>
<td>QUICK FN</td>
</tr>
<tr>
<td>18</td>
<td>CSC</td>
<td></td>
<td>QUICK FN</td>
</tr>
<tr>
<td>19</td>
<td>SEC</td>
<td></td>
<td>QUICK FN</td>
</tr>
<tr>
<td>20</td>
<td>SEC</td>
<td></td>
<td>QUICK FN</td>
</tr>
<tr>
<td>24</td>
<td>DCOR</td>
<td>57</td>
<td>GLOBAL VAR</td>
</tr>
<tr>
<td>25</td>
<td>LOG</td>
<td></td>
<td>QUICK FN</td>
</tr>
<tr>
<td>26</td>
<td>GRIDN</td>
<td>59</td>
<td>GLOBAL VAR</td>
</tr>
<tr>
<td>27</td>
<td>RND</td>
<td>56</td>
<td>GLOBAL VAR</td>
</tr>
<tr>
<td>28</td>
<td>RND</td>
<td>10</td>
<td>LONG FN</td>
</tr>
<tr>
<td>29</td>
<td>ACOS</td>
<td></td>
<td>QUICK FN</td>
</tr>
<tr>
<td>30</td>
<td>MILRAD</td>
<td>64</td>
<td>GLOBAL VAR</td>
</tr>
<tr>
<td>33</td>
<td>TAN</td>
<td>QUICK FN</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>ATN</td>
<td>QUICK FN</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>ASIN</td>
<td>QUICK FN</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>COS</td>
<td>QUICK FN</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>COT</td>
<td>QUICK FN</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>RGDEV</td>
<td>GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>OT</td>
<td>61 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>EXP</td>
<td>QUICK FN</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>LATDEV</td>
<td>62 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>SQR</td>
<td>QUICK FN</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>BTYA</td>
<td>GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>INVOKE</td>
<td>69 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>BTYE</td>
<td>52 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>TGTRG</td>
<td>67 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>HACROSS</td>
<td>51 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>CHG4MAX</td>
<td>47 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>CHG5MAX</td>
<td>48 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>CHG7MAX</td>
<td>49 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>CHG8MAX</td>
<td>50 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>A05</td>
<td>22 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>E04</td>
<td>17 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>79</td>
<td>A14</td>
<td>12 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>TGTAYZ</td>
<td>68 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>E05</td>
<td>26 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>A15</td>
<td>41 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>B14</td>
<td>18 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>81</td>
<td>C04</td>
<td>16 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>82</td>
<td>A24</td>
<td>11 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>A07</td>
<td>31 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>E07</td>
<td>37 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>85</td>
<td>A17</td>
<td>35 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>86</td>
<td>B24</td>
<td>19 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>87</td>
<td>C14</td>
<td>13 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>BTRYL</td>
<td>55 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>89</td>
<td>A18</td>
<td>40 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>E07</td>
<td>35 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>91</td>
<td>A17</td>
<td>30 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>B24</td>
<td>28 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>93</td>
<td>C24</td>
<td>24 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>94</td>
<td>E08</td>
<td>44 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>95</td>
<td>A18</td>
<td>39 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>B17</td>
<td>36 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>97</td>
<td>C07</td>
<td>34 GLOBAL VAR</td>
<td></td>
</tr>
<tr>
<td>Column</td>
<td>Value</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>83</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A27</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C25</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>29</td>
<td>GLOBAL VAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23</td>
<td>GLOBAL VAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ETRYN</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>B18</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C08</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A28</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>E27</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C17</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>53</td>
<td>GLOBAL VAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>45</td>
<td>GLOBAL VAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>43</td>
<td>GLOBAL VAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>38</td>
<td>GLOBAL VAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>37</td>
<td>GLOBAL VAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>33</td>
<td>GLOBAL VAR</td>
</tr>
<tr>
<td>84</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ROTCOR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>B28</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C18</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C27</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>65</td>
<td>GLOBAL VAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>46</td>
<td>GLOBAL VAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>42</td>
<td>GLOBAL VAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32</td>
<td>GLOBAL VAR</td>
</tr>
<tr>
<td>85</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C28</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>41</td>
<td>GLOBAL VAR</td>
</tr>
</tbody>
</table>
# APPENDIX M

## TEST PROGRAM NAME MAPPING FILE (NAMEF)

#### TI-59 REGISTER TO NAME MAPPING

<table>
<thead>
<tr>
<th>REG#</th>
<th>BASIC NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>A 24</td>
</tr>
<tr>
<td>12</td>
<td>A 14</td>
</tr>
<tr>
<td>13</td>
<td>A 04</td>
</tr>
<tr>
<td>14</td>
<td>C 24</td>
</tr>
<tr>
<td>15</td>
<td>C 14</td>
</tr>
<tr>
<td>16</td>
<td>C 04</td>
</tr>
<tr>
<td>17</td>
<td>B 04</td>
</tr>
<tr>
<td>18</td>
<td>B 14</td>
</tr>
<tr>
<td>19</td>
<td>B 24</td>
</tr>
<tr>
<td>20</td>
<td>A 25</td>
</tr>
<tr>
<td>21</td>
<td>A 05</td>
</tr>
<tr>
<td>22</td>
<td>C 25</td>
</tr>
<tr>
<td>23</td>
<td>C 15</td>
</tr>
<tr>
<td>24</td>
<td>C 05</td>
</tr>
<tr>
<td>25</td>
<td>B 05</td>
</tr>
<tr>
<td>26</td>
<td>B 15</td>
</tr>
<tr>
<td>27</td>
<td>B 25</td>
</tr>
<tr>
<td>28</td>
<td>A 27</td>
</tr>
<tr>
<td>29</td>
<td>A 07</td>
</tr>
<tr>
<td>30</td>
<td>A 17</td>
</tr>
<tr>
<td>31</td>
<td>C 27</td>
</tr>
<tr>
<td>32</td>
<td>C 17</td>
</tr>
<tr>
<td>33</td>
<td>C 07</td>
</tr>
<tr>
<td>34</td>
<td>B 07</td>
</tr>
<tr>
<td>35</td>
<td>B 17</td>
</tr>
<tr>
<td>36</td>
<td>A 28</td>
</tr>
<tr>
<td>37</td>
<td>A 18</td>
</tr>
<tr>
<td>38</td>
<td>B 28</td>
</tr>
<tr>
<td>39</td>
<td>A 18</td>
</tr>
<tr>
<td>40</td>
<td>A 08</td>
</tr>
<tr>
<td>41</td>
<td>A 18</td>
</tr>
<tr>
<td>42</td>
<td>A 08</td>
</tr>
<tr>
<td>43</td>
<td>C 08</td>
</tr>
<tr>
<td>44</td>
<td>B 08</td>
</tr>
<tr>
<td>45</td>
<td>B 18</td>
</tr>
<tr>
<td>46</td>
<td>B 28</td>
</tr>
<tr>
<td>47</td>
<td>CHG 4 MAX</td>
</tr>
<tr>
<td>48</td>
<td>CHG 5 MAX</td>
</tr>
<tr>
<td>49</td>
<td>CHG 7 MAX</td>
</tr>
<tr>
<td>50</td>
<td>CHG 8 MAX</td>
</tr>
<tr>
<td>51</td>
<td>HACROSS</td>
</tr>
<tr>
<td>52</td>
<td>TRYE</td>
</tr>
<tr>
<td>53</td>
<td>TRYA</td>
</tr>
<tr>
<td>54</td>
<td>TRYL</td>
</tr>
<tr>
<td>55</td>
<td>RGK</td>
</tr>
<tr>
<td>56</td>
<td>DFCOR</td>
</tr>
<tr>
<td>57</td>
<td>GRIDB</td>
</tr>
<tr>
<td>58</td>
<td>GRIDN</td>
</tr>
<tr>
<td>59</td>
<td>GRIDA</td>
</tr>
<tr>
<td>60</td>
<td>GT</td>
</tr>
<tr>
<td>61</td>
<td>LAT DEV</td>
</tr>
<tr>
<td>62</td>
<td>RGDEV</td>
</tr>
<tr>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>
# APPENDIX N

**TEST PROGRAM DATA/READ MAPPING FILE (READP)**

---

### INPUT DATA TO READ MAPPING

<table>
<thead>
<tr>
<th>DATA</th>
<th>REG</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.133670</td>
<td>11</td>
<td>A24</td>
</tr>
<tr>
<td>21.2691</td>
<td>12</td>
<td>A14</td>
</tr>
<tr>
<td>-105.7</td>
<td>13</td>
<td>A04</td>
</tr>
<tr>
<td>-0.0001499</td>
<td>14</td>
<td>C24</td>
</tr>
<tr>
<td>0.0630</td>
<td>15</td>
<td>C14</td>
</tr>
<tr>
<td>-0.41</td>
<td>16</td>
<td>C04</td>
</tr>
<tr>
<td>-0.77</td>
<td>17</td>
<td>B04</td>
</tr>
<tr>
<td>0.01314</td>
<td>18</td>
<td>B14</td>
</tr>
<tr>
<td>-0.0001720</td>
<td>19</td>
<td>B24</td>
</tr>
<tr>
<td>-0.0149331</td>
<td>20</td>
<td>A25</td>
</tr>
<tr>
<td>24.3439</td>
<td>21</td>
<td>A15</td>
</tr>
<tr>
<td>64.7</td>
<td>22</td>
<td>A05</td>
</tr>
<tr>
<td>-0.0001420</td>
<td>23</td>
<td>C25</td>
</tr>
<tr>
<td>0.0769</td>
<td>24</td>
<td>C15</td>
</tr>
<tr>
<td>0.06</td>
<td>25</td>
<td>C05</td>
</tr>
<tr>
<td>1.26</td>
<td>26</td>
<td>B05</td>
</tr>
<tr>
<td>0.01508</td>
<td>27</td>
<td>B15</td>
</tr>
<tr>
<td>-0.0001678</td>
<td>28</td>
<td>B25</td>
</tr>
<tr>
<td>-0.0173835</td>
<td>29</td>
<td>A27</td>
</tr>
<tr>
<td>29.58741</td>
<td>30</td>
<td>A17</td>
</tr>
<tr>
<td>2235.2</td>
<td>31</td>
<td>A07</td>
</tr>
<tr>
<td>-0.0001668</td>
<td>32</td>
<td>C27</td>
</tr>
<tr>
<td>0.09487</td>
<td>33</td>
<td>C17</td>
</tr>
<tr>
<td>3.29</td>
<td>34</td>
<td>C07</td>
</tr>
<tr>
<td>1.3</td>
<td>35</td>
<td>B07</td>
</tr>
<tr>
<td>0.02713</td>
<td>36</td>
<td>B17</td>
</tr>
<tr>
<td>-0.0001306</td>
<td>37</td>
<td>B27</td>
</tr>
<tr>
<td>-0.0182137</td>
<td>38</td>
<td>A28</td>
</tr>
<tr>
<td>32.3731</td>
<td>39</td>
<td>A18</td>
</tr>
<tr>
<td>4167.4</td>
<td>40</td>
<td>A08</td>
</tr>
<tr>
<td>-0.0001668</td>
<td>41</td>
<td>C28</td>
</tr>
<tr>
<td>0.09727</td>
<td>42</td>
<td>C18</td>
</tr>
<tr>
<td>5.74</td>
<td>43</td>
<td>C08</td>
</tr>
<tr>
<td>1.36</td>
<td>44</td>
<td>B08</td>
</tr>
<tr>
<td>0.02891</td>
<td>45</td>
<td>B18</td>
</tr>
<tr>
<td>-0.0001410</td>
<td>46</td>
<td>B28</td>
</tr>
<tr>
<td>5700</td>
<td>47</td>
<td>CHG4MAX</td>
</tr>
<tr>
<td>7000</td>
<td>48</td>
<td>CHG5MAX</td>
</tr>
<tr>
<td>10800</td>
<td>49</td>
<td>CHG7MAX</td>
</tr>
<tr>
<td>17600</td>
<td>50</td>
<td>CHG8MAX</td>
</tr>
</tbody>
</table>

---

267
<table>
<thead>
<tr>
<th>715</th>
<th>51</th>
<th>HACROSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>52</td>
<td>BRYE</td>
</tr>
<tr>
<td>0</td>
<td>53</td>
<td>BRYN</td>
</tr>
<tr>
<td>800</td>
<td>54</td>
<td>BRYA</td>
</tr>
<tr>
<td>1.0</td>
<td>55</td>
<td>BRYL</td>
</tr>
<tr>
<td>0</td>
<td>56</td>
<td>EGK</td>
</tr>
<tr>
<td>4000</td>
<td>57</td>
<td>DFCOR</td>
</tr>
<tr>
<td>4000</td>
<td></td>
<td>GRIDE</td>
</tr>
<tr>
<td>10</td>
<td>58</td>
<td>GRIDN</td>
</tr>
<tr>
<td></td>
<td>59</td>
<td>GRIDA</td>
</tr>
<tr>
<td>4000</td>
<td>60</td>
<td>OT</td>
</tr>
<tr>
<td>-400</td>
<td>61</td>
<td>LATDEV</td>
</tr>
<tr>
<td>10</td>
<td>62</td>
<td>RGDEV</td>
</tr>
<tr>
<td>1018.5924</td>
<td>63</td>
<td>MILRAD</td>
</tr>
<tr>
<td>3200</td>
<td>64</td>
<td>HOTCOR</td>
</tr>
<tr>
<td></td>
<td>65</td>
<td>REFDEF</td>
</tr>
</tbody>
</table>
APPENDIX O
TEST PROGRAM LINK INTERFACE FILE (SCRATCH)

$1 IS NEXT AVAILABLE REG.

$2

==---------------------------------------==
| TI-59 CODE TRANSLATED FROM W-BASIC     |
| (UNSEGMENTED)                          |
|========================================|

ADDRESS CODE

$0     BEGIN TI-59 CODE.
000    76   2ND LEL
001    11   A
002    71   SBR
003    12   2ND NOP
004    68   2ND NOP
005    71   SBR
006    13   C
007    68   2ND NOP
008    71   SBR
009    14   2ND NOP
010    68   2ND NOP
011    24   CE
012    08   8
013    08   8
014    08   8
015    08   8
016    76   2ND LEL
017    12   B
018    53   RCL
019    59   +
020    53   RCL
021    53   RCL
022    63   63
023    63   63
024    63   63
025    63   63
026    63   63
027    63   63
028    63   63
029    63   63
030    63   63
031    63   63
032    63   63
033    63   63
034    63   63
035    63   63
036    63   63
037    63   63
038    63   63
039    63   63
040    63   63
041    63   63
042    63   63
043    63   63
044    63   63
045    63   63

269
2

(2)  
(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)

2

(2)
460 54 } STO 80
461 42 RCL 80
462 43 X<>T
463 80 RCL 51
464 77 2ND X>=T
465 68 RCL
466 99 2ND FPT
467 43 RCL 67
468 67 2ND FPT
469 61 2ND ADV
470 65 2ND ADV
471 05 2ND
472 66 RCL
473 75 RCL
474 65 RCL
475 80 RCL
476 65 RCL
477 75 RCL
478 65 RCL
479 80 RCL
480 65 RCL
481 02 RCL
482 2ND PRT
483 53 2ND ADV
484 43 RCL 66
485 66 RCL
486 55 RCL
487 79 RCL
488 79 RCL
489 55 RCL
490 80 RCL
491 60 2ND
492 80 2ND
493 53 2ND
494 43 RCL 60
495 43 RCL 60
496 43 RCL 60
497 43 RCL 60
END TI-59 CODE.
## TI-59 Register to Name Mapping

<table>
<thead>
<tr>
<th>REG#</th>
<th>BASIC NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>A24</td>
</tr>
<tr>
<td>12</td>
<td>A04</td>
</tr>
<tr>
<td>13</td>
<td>C24</td>
</tr>
<tr>
<td>14</td>
<td>C14</td>
</tr>
<tr>
<td>15</td>
<td>C04</td>
</tr>
<tr>
<td>16</td>
<td>B04</td>
</tr>
<tr>
<td>17</td>
<td>B14</td>
</tr>
<tr>
<td>18</td>
<td>B24</td>
</tr>
<tr>
<td>19</td>
<td>A25</td>
</tr>
<tr>
<td>20</td>
<td>A05</td>
</tr>
<tr>
<td>21</td>
<td>C15</td>
</tr>
<tr>
<td>22</td>
<td>C05</td>
</tr>
<tr>
<td>23</td>
<td>B15</td>
</tr>
<tr>
<td>24</td>
<td>B05</td>
</tr>
<tr>
<td>25</td>
<td>B25</td>
</tr>
<tr>
<td>26</td>
<td>A17</td>
</tr>
<tr>
<td>27</td>
<td>C27</td>
</tr>
<tr>
<td>28</td>
<td>C17</td>
</tr>
<tr>
<td>29</td>
<td>B07</td>
</tr>
<tr>
<td>30</td>
<td>B17</td>
</tr>
<tr>
<td>31</td>
<td>B27</td>
</tr>
<tr>
<td>32</td>
<td>A18</td>
</tr>
<tr>
<td>33</td>
<td>C28</td>
</tr>
<tr>
<td>34</td>
<td>C18</td>
</tr>
<tr>
<td>35</td>
<td>C08</td>
</tr>
<tr>
<td>36</td>
<td>B08</td>
</tr>
<tr>
<td>37</td>
<td>B18</td>
</tr>
<tr>
<td>38</td>
<td>B28</td>
</tr>
<tr>
<td>39</td>
<td>A19</td>
</tr>
<tr>
<td>40</td>
<td>A09</td>
</tr>
<tr>
<td>41</td>
<td>C29</td>
</tr>
<tr>
<td>42</td>
<td>C19</td>
</tr>
<tr>
<td>43</td>
<td>C09</td>
</tr>
<tr>
<td>44</td>
<td>B09</td>
</tr>
<tr>
<td>45</td>
<td>B19</td>
</tr>
<tr>
<td>46</td>
<td>CHG4MAX</td>
</tr>
<tr>
<td>47</td>
<td>CHG5MAX</td>
</tr>
<tr>
<td>48</td>
<td>CHG6MAX</td>
</tr>
<tr>
<td>49</td>
<td>CHG7MAX</td>
</tr>
<tr>
<td>50</td>
<td>CHG8MAX</td>
</tr>
<tr>
<td>51</td>
<td>HACHROSS</td>
</tr>
<tr>
<td>52</td>
<td>BTRYE</td>
</tr>
<tr>
<td>53</td>
<td>BTRYN</td>
</tr>
<tr>
<td>54</td>
<td>BTRYA</td>
</tr>
<tr>
<td>55</td>
<td>BTRYL</td>
</tr>
<tr>
<td>56</td>
<td>DFCOR</td>
</tr>
<tr>
<td>57</td>
<td>GRIDE</td>
</tr>
<tr>
<td>58</td>
<td>GRIDN</td>
</tr>
<tr>
<td>59</td>
<td>GRIDA</td>
</tr>
<tr>
<td>60</td>
<td>OT</td>
</tr>
<tr>
<td>61</td>
<td>LATDEV</td>
</tr>
<tr>
<td>62</td>
<td>RGDDEV</td>
</tr>
<tr>
<td>63</td>
<td>MILRAD</td>
</tr>
<tr>
<td>64</td>
<td>ROTCOR</td>
</tr>
<tr>
<td>65</td>
<td>REFDEP</td>
</tr>
<tr>
<td>66</td>
<td>TGTRG</td>
</tr>
<tr>
<td>67</td>
<td>TGTAZ</td>
</tr>
<tr>
<td>68</td>
<td>INVOKE</td>
</tr>
<tr>
<td>69</td>
<td>PND</td>
</tr>
<tr>
<td>70</td>
<td>(FN PARAMETER)</td>
</tr>
<tr>
<td>71</td>
<td>(FN PARAMETER)</td>
</tr>
<tr>
<td>72</td>
<td></td>
</tr>
</tbody>
</table>
**INPUT DATA TO READ MAPPING**

<table>
<thead>
<tr>
<th>DATA</th>
<th>REG</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0133670</td>
<td>11</td>
<td>A24</td>
</tr>
<tr>
<td>212691</td>
<td>12</td>
<td>A14</td>
</tr>
<tr>
<td>-105.7</td>
<td>13</td>
<td>A04</td>
</tr>
<tr>
<td>-00001499</td>
<td>14</td>
<td>C24</td>
</tr>
<tr>
<td>0.6630</td>
<td>15</td>
<td>C14</td>
</tr>
<tr>
<td>-4</td>
<td>16</td>
<td>C04</td>
</tr>
<tr>
<td>-01314</td>
<td>17</td>
<td>B04</td>
</tr>
<tr>
<td>00001720</td>
<td>18</td>
<td>B14</td>
</tr>
<tr>
<td>-0149331</td>
<td>19</td>
<td>B24</td>
</tr>
<tr>
<td>24.3439</td>
<td>20</td>
<td>A25</td>
</tr>
<tr>
<td>64.7</td>
<td>21</td>
<td>A15</td>
</tr>
<tr>
<td>-00001420</td>
<td>22</td>
<td>A05</td>
</tr>
<tr>
<td>0.7069</td>
<td>23</td>
<td>C25</td>
</tr>
<tr>
<td>0.06</td>
<td>24</td>
<td>C15</td>
</tr>
<tr>
<td>1.26</td>
<td>25</td>
<td>C05</td>
</tr>
<tr>
<td>0.01506</td>
<td>26</td>
<td>B05</td>
</tr>
<tr>
<td>00001678</td>
<td>27</td>
<td>B15</td>
</tr>
<tr>
<td>-0173835</td>
<td>28</td>
<td>B25</td>
</tr>
<tr>
<td>29.8741</td>
<td>29</td>
<td>A27</td>
</tr>
<tr>
<td>2255.2</td>
<td>30</td>
<td>A17</td>
</tr>
<tr>
<td>-00001668</td>
<td>31</td>
<td>A07</td>
</tr>
<tr>
<td>0.65487</td>
<td>32</td>
<td>C27</td>
</tr>
<tr>
<td>3.29</td>
<td>33</td>
<td>C17</td>
</tr>
<tr>
<td>3.29</td>
<td>34</td>
<td>C07</td>
</tr>
<tr>
<td>1 01313</td>
<td>35</td>
<td>B07</td>
</tr>
<tr>
<td>00001306</td>
<td>36</td>
<td>B17</td>
</tr>
<tr>
<td>-9182137</td>
<td>37</td>
<td>B27</td>
</tr>
<tr>
<td>3.82731</td>
<td>38</td>
<td>A28</td>
</tr>
<tr>
<td>4.10744</td>
<td>39</td>
<td>A18</td>
</tr>
<tr>
<td>-00001668</td>
<td>40</td>
<td>A08</td>
</tr>
<tr>
<td>0.09272</td>
<td>41</td>
<td>C28</td>
</tr>
<tr>
<td>5.74</td>
<td>42</td>
<td>C18</td>
</tr>
<tr>
<td>1.36</td>
<td>43</td>
<td>C08</td>
</tr>
<tr>
<td>0.02891</td>
<td>44</td>
<td>B08</td>
</tr>
<tr>
<td>0.00001410</td>
<td>45</td>
<td>B18</td>
</tr>
<tr>
<td>5706</td>
<td>46</td>
<td>B28</td>
</tr>
<tr>
<td>10000</td>
<td>47</td>
<td>CHG4 MAX</td>
</tr>
<tr>
<td>10800</td>
<td>48</td>
<td>CHG5 MAX</td>
</tr>
<tr>
<td>17600</td>
<td>49</td>
<td>CHG6 MAX</td>
</tr>
<tr>
<td>715</td>
<td>50</td>
<td>CHG8 MAX</td>
</tr>
<tr>
<td>51</td>
<td>51</td>
<td>HACROSS</td>
</tr>
<tr>
<td>0</td>
<td>52</td>
<td>BRYE</td>
</tr>
<tr>
<td>0</td>
<td>53</td>
<td>BRYN</td>
</tr>
<tr>
<td>800</td>
<td>54</td>
<td>BRYA</td>
</tr>
<tr>
<td>10</td>
<td>55</td>
<td>BTRYL</td>
</tr>
<tr>
<td>0</td>
<td>56</td>
<td>RGK</td>
</tr>
<tr>
<td>280</td>
<td></td>
<td>DFCOR</td>
</tr>
</tbody>
</table>
* CONGRATULATIONS, YOU HAVE JUST COMPILED A BASIC PROGRAM INTO A TI-59 PROGRAM. IN 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:

<table>
<thead>
<tr>
<th>PARTITION</th>
<th>PROGRAM STEPS</th>
<th>STORAGE REGISTERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>719.29</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>479.59</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>239.59</td>
<td></td>
</tr>
</tbody>
</table>

FORMAT: X YYY.ZZ
WHERE X STANDS FOR PARTITION NUMBER
YY STANDS FOR PROGRAM STEPS (0-YYY)
ZZ STANDS FOR REGISTERS (0-ZZ).
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 PROGRAMMERS 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

<table>
<thead>
<tr>
<th>MODULE #</th>
<th>CARD1</th>
<th>SIDE1</th>
<th>BANK1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MODULE #</th>
<th>CARD1</th>
<th>SIDE1</th>
<th>BANK2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MODULE #</th>
<th>CARD2</th>
<th>SIDE1</th>
<th>BANK3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TI-59 LISTING

* MANU. RETURN REGISTER TOP IS 81
STORE IN REGISTER: 8

* PROGRAM PARTITION IS 239.89
* PARTITION NUMBER IS 9

*MODULE # 1
CARD #1
SIDE #1

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>76</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>002</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>003</td>
<td>68</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>004</td>
<td>71</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>005</td>
<td>71</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>006</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>007</td>
<td>68</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>008</td>
<td>71</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>009</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>68</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>012</td>
<td>03</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>013</td>
<td>03</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

283
END
9-83
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A
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:

  \[
  \begin{align*}
  & X_{2nd} \\
  & \text{OP} \\
  & 17
  \end{align*}
  \]

  * \(X\) 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
    * LCAD 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 2ND OF THIS LISTING. MANUALLY ENTER THE GIVEN DATA INTO THE REGISTERS USING THE FOLLOWING KEYSTROKES:

    \[
    \text{DATA} \\
    \text{STO} \\
    \text{XX}
    \]

    WHERE \(XX\) IS THE DESIRED REGISTER NUMBER.

    * INITIALIZE THE MANUAL SBR RETURN CONTROL STACK WITH THE FOLLOWING KEYSTROKES:

    \[
    \text{XX} \\
    \text{STO} \\
    \text{08}
    \]

    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
        YY
      * 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 FLASHERS 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 CODE.
  * 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 CODE.
  * 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 CODE.
  * 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
<table>
<thead>
<tr>
<th>DATA</th>
<th>REG</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.0133670</td>
<td>11</td>
<td>A24</td>
</tr>
<tr>
<td>21.2691</td>
<td>12</td>
<td>A14</td>
</tr>
<tr>
<td>-105.7</td>
<td>13</td>
<td>A04</td>
</tr>
<tr>
<td>-0.0001499</td>
<td>14</td>
<td>C24</td>
</tr>
<tr>
<td>-0.06630</td>
<td>15</td>
<td>C14</td>
</tr>
<tr>
<td>-41.7</td>
<td>16</td>
<td>C04</td>
</tr>
<tr>
<td>0.77</td>
<td>17</td>
<td>B04</td>
</tr>
<tr>
<td>0.1314</td>
<td>18</td>
<td>B14</td>
</tr>
<tr>
<td>0.0001720</td>
<td>19</td>
<td>B24</td>
</tr>
<tr>
<td>-0.0149331</td>
<td>20</td>
<td>A25</td>
</tr>
<tr>
<td>24.3439</td>
<td>21</td>
<td>A15</td>
</tr>
<tr>
<td>64.7</td>
<td>22</td>
<td>A05</td>
</tr>
<tr>
<td>-0.0001420</td>
<td>23</td>
<td>C25</td>
</tr>
<tr>
<td>0.07069</td>
<td>24</td>
<td>C15</td>
</tr>
<tr>
<td>0.056</td>
<td>25</td>
<td>C05</td>
</tr>
<tr>
<td>1.26</td>
<td>26</td>
<td>B05</td>
</tr>
<tr>
<td>0.01506</td>
<td>27</td>
<td>B15</td>
</tr>
<tr>
<td>0.0001678</td>
<td>28</td>
<td>B25</td>
</tr>
<tr>
<td>-0.0173835</td>
<td>29</td>
<td>A27</td>
</tr>
<tr>
<td>29.6741</td>
<td>30</td>
<td>A17</td>
</tr>
<tr>
<td>2255.2</td>
<td>31</td>
<td>A07</td>
</tr>
<tr>
<td>-0.0001688</td>
<td>32</td>
<td>C27</td>
</tr>
<tr>
<td>0.6487</td>
<td>33</td>
<td>C17</td>
</tr>
<tr>
<td>3.29</td>
<td>34</td>
<td>C07</td>
</tr>
<tr>
<td>1.3</td>
<td>35</td>
<td>B07</td>
</tr>
<tr>
<td>0.02713</td>
<td>36</td>
<td>B17</td>
</tr>
<tr>
<td>0.0001306</td>
<td>37</td>
<td>B27</td>
</tr>
<tr>
<td>-0.0182137</td>
<td>38</td>
<td>A28</td>
</tr>
<tr>
<td>32.3734</td>
<td>39</td>
<td>A18</td>
</tr>
<tr>
<td>4107.4</td>
<td>40</td>
<td>A08</td>
</tr>
<tr>
<td>-0.0001668</td>
<td>41</td>
<td>C28</td>
</tr>
<tr>
<td>1.09272</td>
<td>42</td>
<td>C18</td>
</tr>
<tr>
<td>5.74</td>
<td>43</td>
<td>C08</td>
</tr>
<tr>
<td>1.36</td>
<td>44</td>
<td>B08</td>
</tr>
<tr>
<td>0.02891</td>
<td>45</td>
<td>B18</td>
</tr>
<tr>
<td>0.0001410</td>
<td>46</td>
<td>B28</td>
</tr>
<tr>
<td>5706</td>
<td>47</td>
<td>CHG4MAX</td>
</tr>
<tr>
<td>7000</td>
<td>48</td>
<td>CHG65MAX</td>
</tr>
<tr>
<td>10800</td>
<td>49</td>
<td>CHG7MAX</td>
</tr>
<tr>
<td>17600</td>
<td>50</td>
<td>CHG8MAX</td>
</tr>
<tr>
<td>715</td>
<td>51</td>
<td>MACROSS</td>
</tr>
<tr>
<td>0</td>
<td>52</td>
<td>BTRYE</td>
</tr>
<tr>
<td>0</td>
<td>53</td>
<td>BRYW</td>
</tr>
<tr>
<td>0</td>
<td>54</td>
<td>BRYA</td>
</tr>
<tr>
<td>800</td>
<td>55</td>
<td>BRYL</td>
</tr>
<tr>
<td>1-0</td>
<td>56</td>
<td>RGK</td>
</tr>
<tr>
<td>0</td>
<td>57</td>
<td>DFCOR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>4000</td>
<td>4300</td>
<td>10</td>
</tr>
<tr>
<td>4000</td>
<td>4000</td>
<td>10</td>
</tr>
</tbody>
</table>

Grids: GRIDE, GRIDN, GRIDA, OT, LATDEV, RGDEV, MILRAD, HOTCOR, REFDEF
<table>
<thead>
<tr>
<th>REG#</th>
<th>BASIC NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>A24</td>
</tr>
<tr>
<td>12</td>
<td>A14</td>
</tr>
<tr>
<td>13</td>
<td>A04</td>
</tr>
<tr>
<td>14</td>
<td>C24</td>
</tr>
<tr>
<td>15</td>
<td>C14</td>
</tr>
<tr>
<td>16</td>
<td>C04</td>
</tr>
<tr>
<td>17</td>
<td>B24</td>
</tr>
<tr>
<td>18</td>
<td>B14</td>
</tr>
<tr>
<td>19</td>
<td>B04</td>
</tr>
<tr>
<td>21</td>
<td>A25</td>
</tr>
<tr>
<td>22</td>
<td>A15</td>
</tr>
<tr>
<td>23</td>
<td>A05</td>
</tr>
<tr>
<td>24</td>
<td>C25</td>
</tr>
<tr>
<td>25</td>
<td>C15</td>
</tr>
<tr>
<td>26</td>
<td>C05</td>
</tr>
<tr>
<td>27</td>
<td>B25</td>
</tr>
<tr>
<td>28</td>
<td>B15</td>
</tr>
<tr>
<td>29</td>
<td>B05</td>
</tr>
<tr>
<td>31</td>
<td>A27</td>
</tr>
<tr>
<td>32</td>
<td>A17</td>
</tr>
<tr>
<td>33</td>
<td>A07</td>
</tr>
<tr>
<td>34</td>
<td>C27</td>
</tr>
<tr>
<td>35</td>
<td>C17</td>
</tr>
<tr>
<td>36</td>
<td>C07</td>
</tr>
<tr>
<td>37</td>
<td>B27</td>
</tr>
<tr>
<td>38</td>
<td>B17</td>
</tr>
<tr>
<td>39</td>
<td>B07</td>
</tr>
<tr>
<td>41</td>
<td>A28</td>
</tr>
<tr>
<td>42</td>
<td>A18</td>
</tr>
<tr>
<td>43</td>
<td>A08</td>
</tr>
<tr>
<td>44</td>
<td>C28</td>
</tr>
<tr>
<td>45</td>
<td>C18</td>
</tr>
<tr>
<td>46</td>
<td>C08</td>
</tr>
<tr>
<td>47</td>
<td>B28</td>
</tr>
<tr>
<td>48</td>
<td>B18</td>
</tr>
<tr>
<td>49</td>
<td>B08</td>
</tr>
<tr>
<td>51</td>
<td>CHG5MAX</td>
</tr>
<tr>
<td>52</td>
<td>HACROSS</td>
</tr>
<tr>
<td>53</td>
<td>3TRYE</td>
</tr>
<tr>
<td>54</td>
<td>3TRYN</td>
</tr>
<tr>
<td>55</td>
<td>3TRYL</td>
</tr>
<tr>
<td>56</td>
<td>RGK</td>
</tr>
<tr>
<td>57</td>
<td>DFCOR</td>
</tr>
<tr>
<td>58</td>
<td>GRIDN</td>
</tr>
<tr>
<td>59</td>
<td>G3ICA</td>
</tr>
<tr>
<td>60</td>
<td>OT</td>
</tr>
<tr>
<td>61</td>
<td>LATDEV</td>
</tr>
<tr>
<td>62</td>
<td>RGDDEV</td>
</tr>
<tr>
<td>63</td>
<td>MILLRAD</td>
</tr>
<tr>
<td>64</td>
<td>ROTCOR</td>
</tr>
<tr>
<td>65</td>
<td>2PDBF</td>
</tr>
<tr>
<td>66</td>
<td>TGTRG</td>
</tr>
<tr>
<td>67</td>
<td>TGTAZ</td>
</tr>
<tr>
<td>68</td>
<td>INVOKE</td>
</tr>
<tr>
<td>69</td>
<td>FN</td>
</tr>
<tr>
<td>70</td>
<td>FN</td>
</tr>
<tr>
<td>71</td>
<td>FN PARAMETER</td>
</tr>
<tr>
<td>72</td>
<td>FN PARAMETER</td>
</tr>
<tr>
<td>73</td>
<td>FN PARAMETER</td>
</tr>
<tr>
<td>Parameter</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td></td>
</tr>
<tr>
<td>79</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
</tr>
</tbody>
</table>

```
END BAX59 SEGMENTATION/INSTRUCTION: VERSION 1.0
```
LIST OF REFERENCES


BIBLIOGRAPHY


**INITIAL DISTRIBUTION LIST**

<table>
<thead>
<tr>
<th>No.</th>
<th>Copies</th>
<th>Distribution List</th>
</tr>
</thead>
</table>
| 1.  | 2      | Defense Technical Information Center  
  Cameron Station  
  Alexandria, Virginia 22314 |
| 2.  | 2      | Library, Code 0142  
  Naval Postgraduate School  
  Monterey, California 93940 |
| 3.  | 1      | Curricular Officer, Code 37  
  Computer Technology  
  Naval Postgraduate School  
  Monterey, California 93940 |
| 4.  | 1      | Department Chairman, Code 52HQ  
  Department of Computer Science  
  Naval Postgraduate School  
  Monterey, California 93940 |
| 5.  | 1      | Dr. Bruce J. MacLennan, Code 52ML  
  Department of Computer Science  
  Naval Postgraduate School  
  Monterey, California 93940 |
| 6.  | 1      | Dr. Gordon E. Latta, Code 53LZ  
  Department of Mathematics  
  Naval Postgraduate School  
  Monterey, California 93940 |
| 7.  | 1      | Dr. Richard W. Hamming, Code 52HG  
  Department of Computer Science  
  Naval Postgraduate School  
  Monterey, California 93940 |
| 8.  | 2      | CET Marie S. Kindl  
  413 E. Washington Street  
  Villa Park, Illinois 60181 |
| 9.  | 2      | CAPT James H.W. Inskeep, Jr  
  F.O. Box 444  
  Locust Grove, Virginia 22101 |
| 10. | 1      | CAPT T. F. Rogers, USN  
  Box 327  
  Lumberport, West Virginia 26386 |
| 11. | 1      | CAPT J. C. White, USMC  
  Computer Science Department  
  U.S. Naval Academy  
  Annapolis, Maryland 21401 |
| 12. | 1      | CET Thomas A. Gandy  
  SMC 1597  
  Naval Postgraduate School  
  Monterey, California 93940 |