ON THE ADAPTABILITY OF MULTIPASS PASCAL COMPILERS TO VARIANTS OF (PASCAL) P-CODE MACHINE ARCHITECTURES

by

MARK A. LITTEKEN

B. S., Pittsburg State University, Pittsburg, Kansas, 1979

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1981

Approved by:

Virg Wallentine

TABLE OF CONTENTS

1 Introduction 1

1.1 Motivation and Statement of Project 1

1.2 Report Organization 2

1.3 Nomenclature 3

1.4 Compiler Geneology 4

2 Architecture and Organization of the Pascal Virtual Machine . 7

2.1 Virtual P-Machine 7

2.2 Interpreter 10

2.3 Support for Concurrent Processes (Kernel) 13

3 Architecture and Organization of the Pascal Microengine . . . 15

3.1 System Overview 15

3.2 P-Machine 15

3.2.1 Registers 16

3.2.2 Object Code File Format 17

3.2.3 Routine Invocation and Stack Organization 22

3.3 Support for Concurrent Processes 26

3.4 P-Machine Semantics for Concurrent Pascal 31

3.4.1 Code Segments 31

3.4.2 Data Spaces 42

4 Structure of the Concurrent Pascal Compiler 52

4.1 Overview and Summary of Passes One Through Five 52

4.2 Pass Six 57

4.3 Pass Seven 64

5 Compiler Modifications 67

5.1 Changes to Passes One Through Five 68

5.2 Changes to Pass Six 73

5.2.1 Pass Output 74

5.2.2 Pass Structure 81

5.2.3 Straightforward Instruction Conversions 91

5.2.4 "Difficult" Instruction Modifications 106

5.2.4.1 Value and Address PUSH Instructions . . . 106

5.2.4.2 INDEX and BYTE Instructions 111

5.2.4.3 CASEJUMP 114

5.2.4.4 Routine Invocation Instructions 117

5.2.4.5 BEGIN Instructions 123

5.2.4.6 ENTER Instructions 127

5.2.4.7 END and EXIT Instructions 133

5.2.4.8 PUSHLABEL (Interface Generation) 135

6 Conclusion 143

6.1 Results and Observations 143

6.2 Future Work 146

References 150

Appendices

A. Software Tools A-1

B. Source Code for MEPASCAL Compiler Pass Six: MEPASS6 . . . B-1

C. Source Code for MEPASCAL Compiler Driver: MEPASCAL C-1

D. Source Code Changes to MCPASCAL Passes 1-5 . . . . D-1

LIST OF FIGURES

1. Concurrent Pascal Compiler Geneoloy 5

2. Stack and Registers After Virtual Machine Function Call . . 9

3. Sample Object Code for Virtual Machine 11

4. Snapshot of Interpreter Execution 12

5. Logical Structure of Virtual Machine 14

6. Microengine Code File and Detail of Header Block 18

7. Microengine Code Segment and Detail of a Routine 20

8. Microengine Dynamic and Static Links 24

9. Stack Configuration After a Microengine Function Call . . . 25

10. Microengine Private Code Segment Linkage 27

11. Microengine Shared Code Segment Linkage 29

12. Microengine Run-time Code Segment Configuration 30

13. Psuedocode for Routine to Handle Kernel Calls 33

14. Pointer Configuration for Sequential Program Invocation . . 36

15. Sample Sequential Program and Interface Definitions 37

16. Virtual Machine Stack After Execution of CALLPROG 39

17. Pseudocode for Interface Segment 41

18. Concurrent Pascal Code Segments On the Microengine 43

19. Kernel and Initial Process Data Spaces 44

20. Stack Space Allocation for Processes 47

21. Boundary Between Process Stack Spaces 49

22. Control Structure of the MCPASCAL Compiler 53

23. Intermediate Code Flow in MCPASCAL 54

24. Heap Table Layout 61

25. Virtual Machine Global Base Location 72

26. Flow Diagram for MEPASS6 75

27. Data Structures Left in the Heap by MEPASS6 76

28. Addressing Mechanisms Used by the Two Target Machines . . . 85

29. Microengine Stack After a Call to a Process ENTRY Routine . 88

30. Virtual Machine Word Instructions and
Their Microengine Equivalents 93

31. Virtual Machine Real Instructions and
Their Microengine Equivalents 95

32. Virtual Machine Set and Structure Instructions, and
Their Microengine Equivalents 96

33. Microengine Code to Emulate NEW 98

34. Microengine Code to Emulate NEWINIT 100

35. Example of the VARIANT Virtual Machine Instruction 103

36. Example of Microengine Byte Pointer 112

37. Example of Virtual Machine CASE Statement 115

38. Control Flow During Life of a Typical Process 131

39. Global Base Pointers and Static Link Pointer
After Invocation of a Process ENTRY Routine 132

40. Skeletal Concurrent Process and Interface Definition 137

41. Layout of an Interface Segment 138

42. Record Offset Calculation for the Two Target Machines . . . 147

43. Input Buffer for File Transfer Program A-55

INTRODUCTION

1.1 MOTIVATION AND STATEMENT OF PROJECT

There are a number of situations where it is important that programs be portable among several machines. That is, it becomes important to be able to write a program in some high-level language and run it on a variety of different machine types (e.g., CDC 6600, IBM 370, PDP-11) without altering the source code. Such a situation occurs in a generalized network of heterogeneous computers where programs should not be unduly restricted from running on any available machine.

NADEX (Network ADaptable EXecutive) [YOUN79a, YOUN79b] is an operating system, implemented in Concurrent Pascal [BRIN77], which supports general graphs (software configurations) of communicating software nodes. The nodes of a software configuration may be sequential or concurrent programs and are not necessarily all resident on the same machine, or even the same machine type. So, for a NADEX network, there are three areas related to portability which must be addressed. First, some portion of NADEX must reside in each machine connected to the network; thus, that code must be executable on each machine type. Second, application programs should be executable on any machine in the network. Third, there may be some machines which are not capable of running compilers which produce object code for themselves, for example, because of small memory size.

One way to provide portable control and application programs is to write cross-compilers which will run on a machine and produce object code for each machine type. Source code can then be compiled for any machine type and sent via a very simple communications protocol to the hardware on which it is to be executed. This is the approach used to provide NADEX support on the Pascal Microengine [MICR79] and described in this report.

Kansas State University currently has an Interdata 8/32 and Western Digital Pascal Microengine (among other machines) which could be included in a network running under NADEX. Concurrent and Sequential Pascal compilers run on the 8/32 and produce object code for it. This report describes an effort to modify a multipass Concurrent Pascal compiler so that it will execute on the Interdata 8/32 and produce object code for the Microengine.

The compiler to be modified generates P-code which appears to be very close to the Microengine's machine code. However, there are subtle but significant differences. A primary objective of the project was to assess the impact of these differences on a multipass compiler.

1.2 REPORT ORGANIZATION

This report consists of five chapters, apart from this introduction. Chapters two and three contain a juxtaposition of the compiler's original target machine (virtual, Concurrent Pascal) and the new one (Pascal Microengine) for which it was to be modified. The overall structure of the Concurrent Pascal compiler is discussed in chapter four. The next chapter is a description of the changes which were made to each pass of the compiler. Since pass six was so extensively changed, the modifications to it are grouped by the affected area of the pass: the objects which are generated as output, the overall structure. and the generated code sequences. Chapter six consists of an identification of the results of the project and the work which remains. The reader is assumed to be familiar with the concepts of the Concurrent Pascal language [BRIN77].

1.3 NOMENCLATURE

Probably none of the terms used in this report will be new to the reader, but there could be some ambiguity surrounding their usage here.

In this report *routine* will be used to refer collectively to the Concurrent Pascal constructs PROCEDURE, FUNCTION (both ENTRY and non-ENTRY), and the initial statements (BEGIN...END. block) of processes, monitors, classes, and sequential programs. (The initial statement of a concurrent program is the same as the initial statement of a process.)

Stacks shown in the figures grow from the top of the page (high addresses) to the bottom (low addresses). The heap grows in the opposite direction. Generally, instructions can refer to three data areas in the stack-- local variables, global variables, and stack operands. *Local variables* are identical to Concurrent Pascal *temporary variables*. *Global variables* are the same as the *permanent*, or *shared*, variables of Concurrent Pascal. Instructions which push or pop values act on the *operand stack*.

Concurrent Pascal processes are synonymous with Microengine tasks-- each is the schedulable entity on the corresponding machine.

A sequential program running under a concurrent process may call certain ENTRY routines of the process. The accessible routines are named by the *interface definition* in the process. Those same routines are named in the sequential program's *prefix*. At run time, calls by the sequential program to its prefix routinesre mapped to a calls to the corresponding process ENTRY routines. In the Concurrent Pascal virtual machine this *mapping* is performed by a jump table. On the Microengine, a special code segment, called the interface segment, (which contains only one routine-- the interface routine) performs the same mapping.

Actions of the two compiler systems can be controlled by specifying options. There are compiler options and driver options. Compiler options are specified at the beginning of the source code being compiled [HART76]. Driver options are specified as parameters in the CSS invocation line which the user enters at the console to start one of the compilers.

Items on the operand stack are referred to by their position relative to the top of the stack. TOS refers to the item on top of the stack, regardless of its length. For instance, the TOS item could be an integer value, occupying one word, or a real, which occupies two words. TOS-1 refers to the item which was pushed onto the stack immediately before the TOS item. Similarly, TOS-2 is the item pushed before TOS-1.

1.4 COMPILER GENEOLOGY

Figure 1 is a graphic illustration of the relationships between the various compiler versions in the geneology described below. At first the intention was to modify the code generator in HCPASCAL-- the original version of Concurrent Pascal received several years ago from the California Institute of Technology [HART76]. The object code (P-code) which it generates seemed close to the Microengine instruction set. The source code for HCPASCAL was not available, however, so an extended version of it (MCPASCAL) was used for the

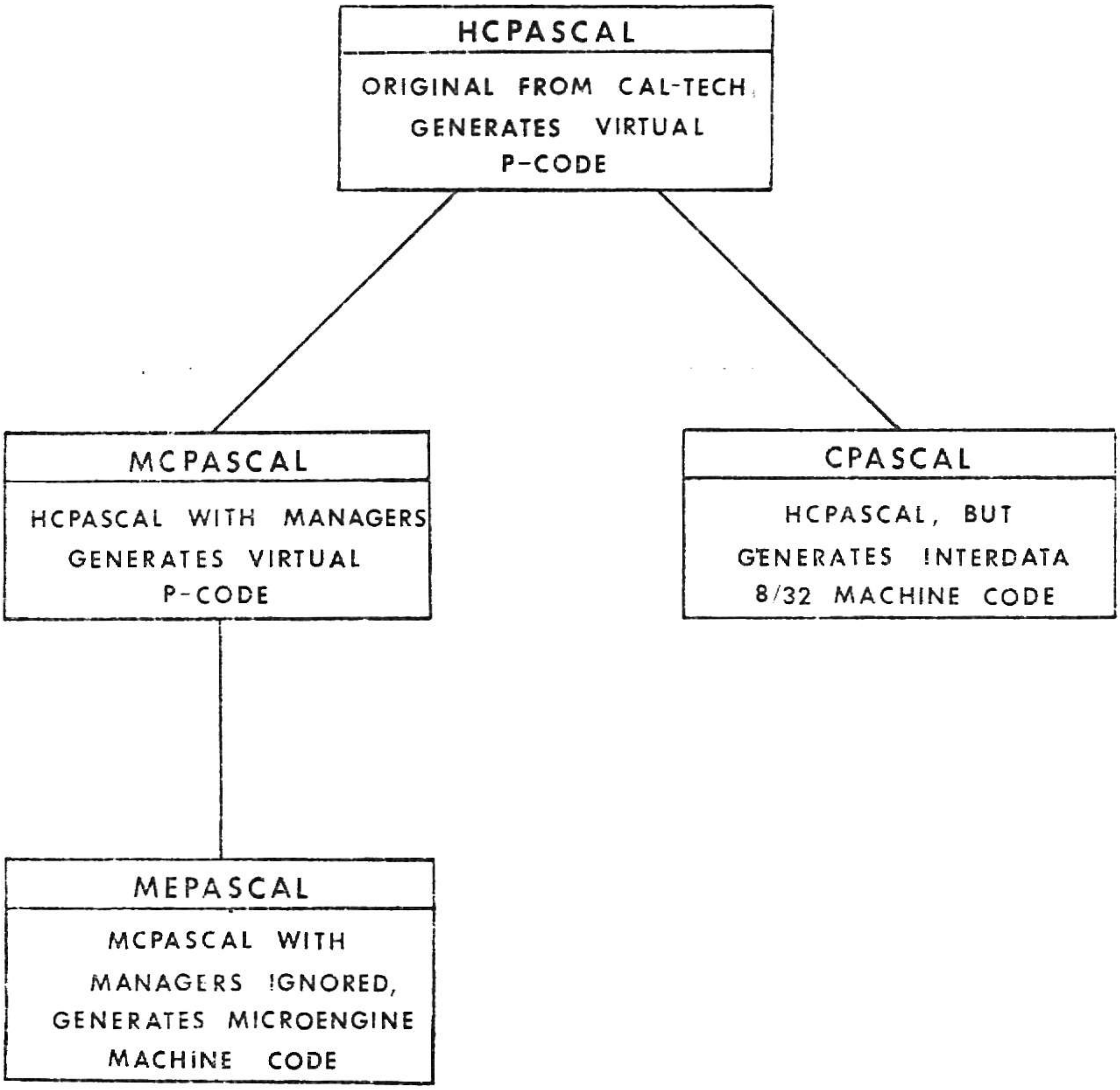


FIGURE 1. Geneology of various Concurrent Pascal compilers at Kansas State University.

project. MCPASCAL [SCHM77] is the same as HCPASCAL, except that it allows use of the manager construct in the source language. For the purposes of this project, the portions which deal with the compilation of managers were ignored. The version of MCPASCAL which generates Microengine machine code is MEPASCAL, and is the result of the work described in this document. CPASCAL, another version of HCPASCAL, accepts the same source language as HCPASCAL but produces object code for the Interdata 8/32. CPASCAL makes more passes over the intermediate code than its parent, but also performs machine-independent and -dependent optimization.

ARCHITECTURE AND ORGANIZATION OF THE PASCAL VIRTUAL MACHINE

The MCPASCAL compiler generates code (called P-code, or virtual code) for a hypothetical machine, not for any existent hardware. P-code will execute on a real machine only if it is run interpretively, or if it is further translated to the language of a real machine. The software which implements the virtual machine has three major aspects: the virtual P-machine (that is, the machine which would be visible to a P-code programmer), the interpreter which presents that view, and the software kernel which interfaces directly with the real hardware.

2.1 VIRTUAL P-MACHINE

The virtual machine executes P-code, has a stack architecture, and uses five virtual registers, designated Q, G, B, S, and H. The Q register is the virtual program counter. G is the global base register, and can point to either of two types of data areas. When P-code in the anonymous initial process is being executed, G points to the data area containing the concurrent program's global variables. However, within system components, G points to the data area containing that component's permanent (global, shared) variables. B is the local base register, and points to the local variable area of a routine, regardless of whether or not it is a system component ENTRY routine. The stack pointer, S, always points to the top of the stack (the last word which was pushed onto it). H is the heap pointer, and points to the first byte of free space.

When a routine is called, machine state information is pushed onto the stack by the CALL instruction and the first instruction in the called routine (for example, ENTER). This structure, called a markstack, consists of the return address in the caller's code (contents of caller's Q register), caller's global base (G), caller's local base (B), the value of S before the caller pushed the actual parameters (if any), and the source code line number of the called routine. The markstack fields are pushed on the stack in the order just given.

In general, routine calls proceed in the following manner. If the called routine is a function, the caller pushes enough space to hold the value to be returned. The caller pushes actual parameters (if any), proceeding left to right through the source code parameter list, and then the return address. The called routine builds the rest of the markstack, resets the local base register, pushes enough space onto the stack to accommodate its local variables, and resets the global base, if necessary. Figure 2 shows a stylized configuration of the data areas and registers after a function has begun execution.

The object code of a concurrent program compiled by MCPASCAL is a sequence of 16-bit integers, divided into three sections: initial process information, virtual code, and long constants. Five integers comprise the initial process information:

1) The byte address of the last word in the object code, relative to the first byte of the code file. That is, relative to the first byte of _this_ integer;
2) The number of bytes of virtual code;
3) The amount of stack space (bytes) required by the program;
4) The number of bytes required for the initial process' permanent variables;
5) The number of bytes in the long constant pool.

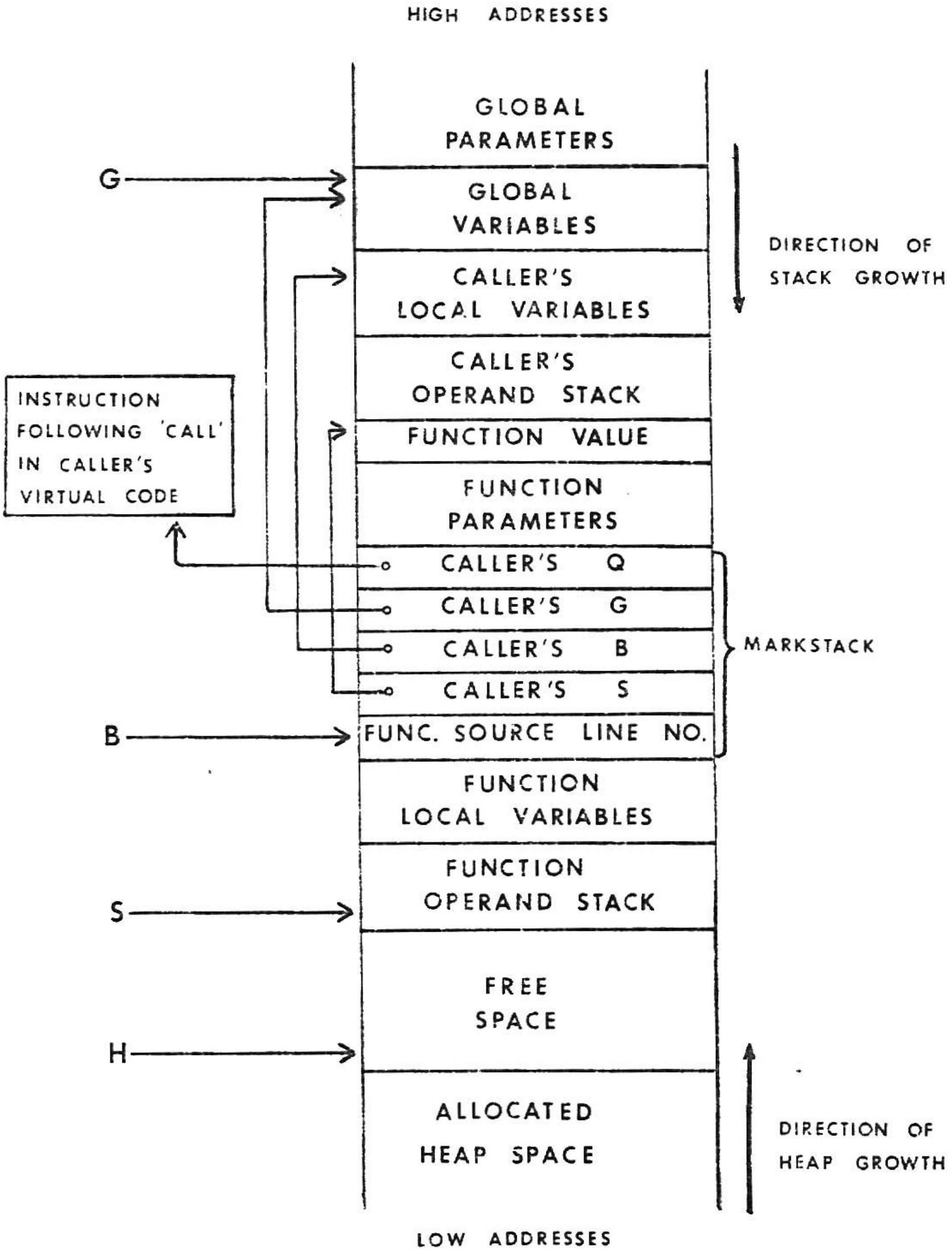


FIGURE 2. Stylized stack and register configuration after a Concurrent Pascal virtual P-machine function call. Calling routine and called function happen to share the same global variables.

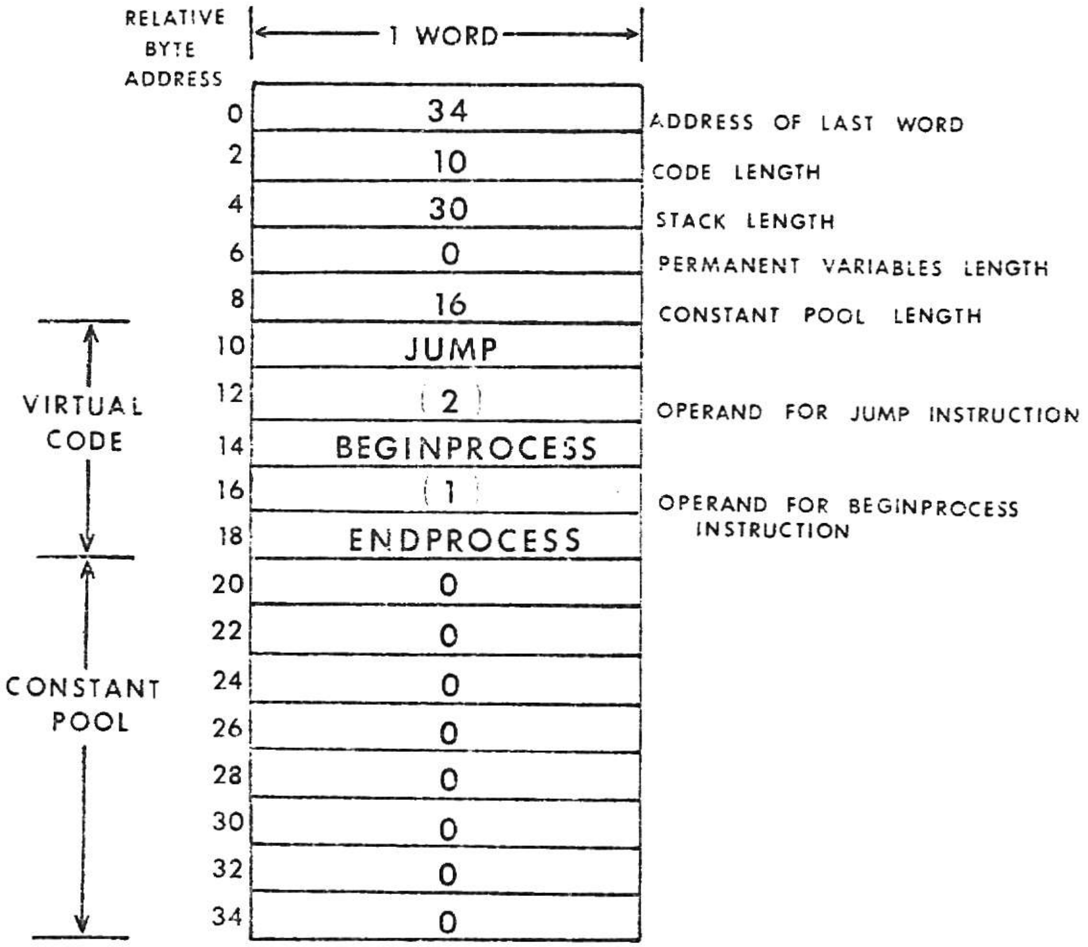
The second section is the virtual code itself. The virtual machine instruction set (P-code) is given in [ZEPK74], along with English and psuedocode descriptions of each instruction's operation. The third section is the pool of long constants (reals, strings (arrays of CHAR), and sets). The constant pool always contains at least 16 bytes since MCPASCAL always generates the null set, even if it is never used. Figure 3 shows the object code which results from compiling a null program.

2.2 INTERPRETER

Object code generated by MCPASCAL and HCPASCAL runs interpretively. The original interpreter [ZEPK74] ran on a PDP-11/45 and consisted of 1K bytes of assembly code. Structurally, it is a jump table and a series of code pieces which carry out the actual interpretation of virtual instructions. The virtual operation codes are indices into the jump table whose entries are the addresses of the corresponding code pieces. Each code piece ends with the PDP-11 assembly instruction

```
MOV    @(Q)+,P
```

where P is the real machine program counter. The Q register contains the address of the next virtual operation code to be interpreted. The single move instruction uses the contents of the word to which Q points (the opcode) to locate a word in the jump table. The content of that word is then loaded into the real machine's program counter, thereby jumping from the code piece which interpreted the current virtual instruction to the code which will interpret the next one. Figure 4 shows the arrangement of virtual code, interpreter, and program counters. [ZEPK74] gives a psuedocode description of the



RELATIVE BYTE ADDRESS	1 WORD	
0	34	ADDRESS OF LAST WORD
2	10	CODE LENGTH
4	30	STACK LENGTH
6	0	PERMANENT VARIABLES LENGTH
8	16	CONSTANT POOL LENGTH
10	JUMP	
12	(2)	OPERAND FOR JUMP INSTRUCTION
14	BEGINPROCESS	
16	(1)	OPERAND FOR BEGINPROCESS INSTRUCTION
18	ENDPROCESS	
20	0	
22	0	
24	0	
26	0	
28	0	
30	0	
32	0	
34	0	

FIGURE 3. Object code file generated by MCPASCAL from the source program: BEGIN END.

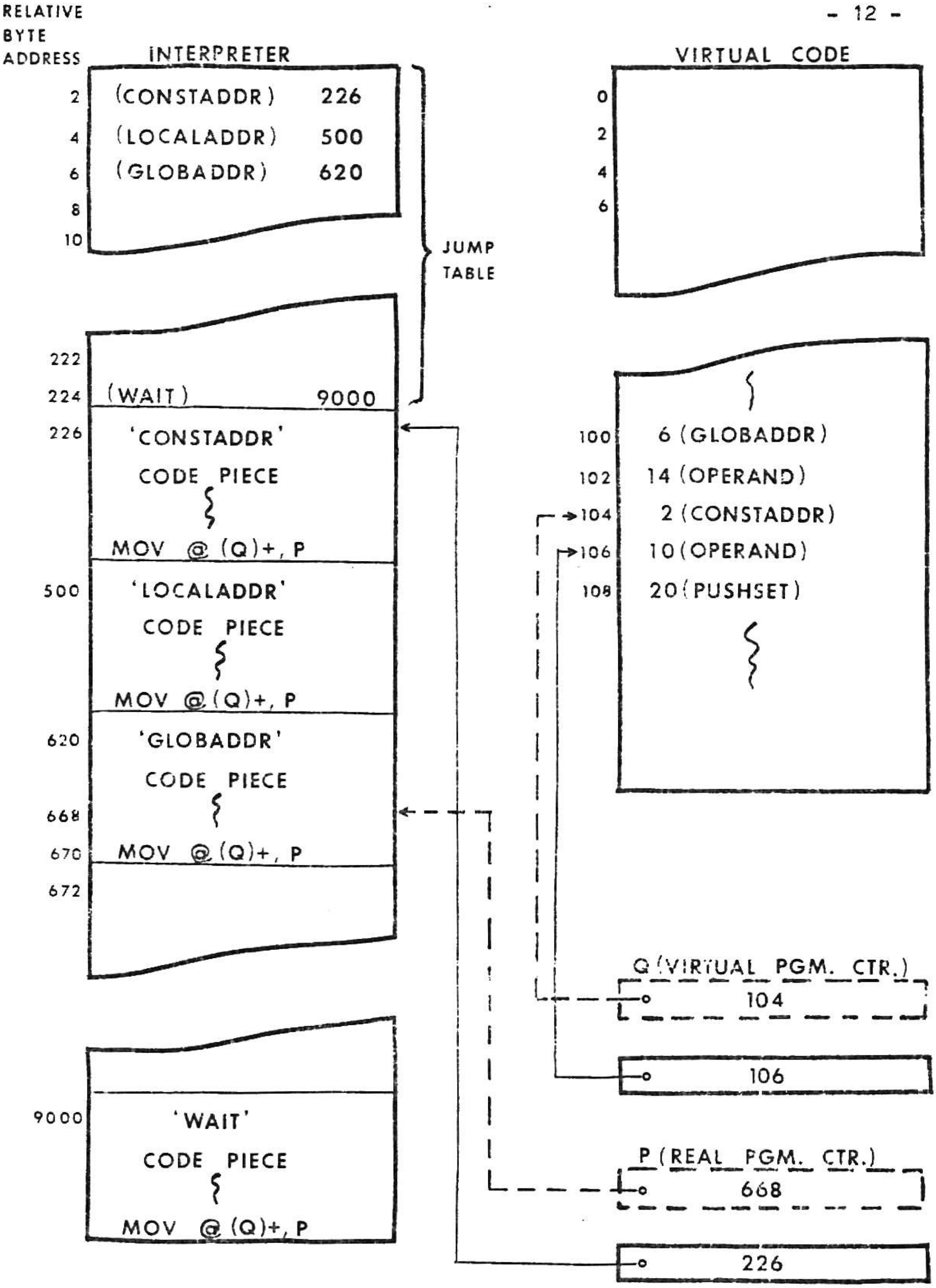


FIGURE 4. Interpreter, virtual code, and program counters before (dashed) and after (solid) execution of the MOV instruction at location 670 in the interpreter. The MOV terminates the interpretation of the GLOBADDR virtual instruction and automatically initiates the interpretation of CONSTADDR. Q points to the next word to be interpreted.

action of each code piece.

2.3 SUPPORT FOR CONCURRENT PROCESSES (KERNEL)

In the PDP-11/45 implementation of Concurrent Pascal a kernel (2.8K bytes of assembly code) handles processor multiplexing and guarantees that processes have exclusive access to monitors [BRIN75]. It also performs some hardware interface functions, such as I/O with peripherals. The interpreter has access to the kernel, via the KERNELCALL operation, in order to obtain the services which it provides. For instance, the virtual instruction INITPROC causes the creation of a new process. On encountering INITPROC, the interpreter calls on the kernel to generate a unique process identifier and process control block (PCB). The newly-created process is then placed in the ready queue for subsequent execution, and control returns to the interpreter.

Monitor access is controlled in the kernel by performing operations on a data structure, called a gate, which contains the state of the monitor (busy or free), to ensure mutual exclusion. Each monitor has its own gate, a record, the fields of which are a boolean (OPEN) and a pointer to a queue of PCBs. If a process is in the monitor (OPEN=FALSE) when another tries to enter, the PCB of the entering process is placed in the gate's queue. When the process currently in the monitor leaves, another one is selected from the queue and allowed to enter. If the queue is empty when a process exits, OPEN is made TRUE. Figure 5 depicts the logical relationship of the kernel to the interpreter and virtual code of a concurrent program.

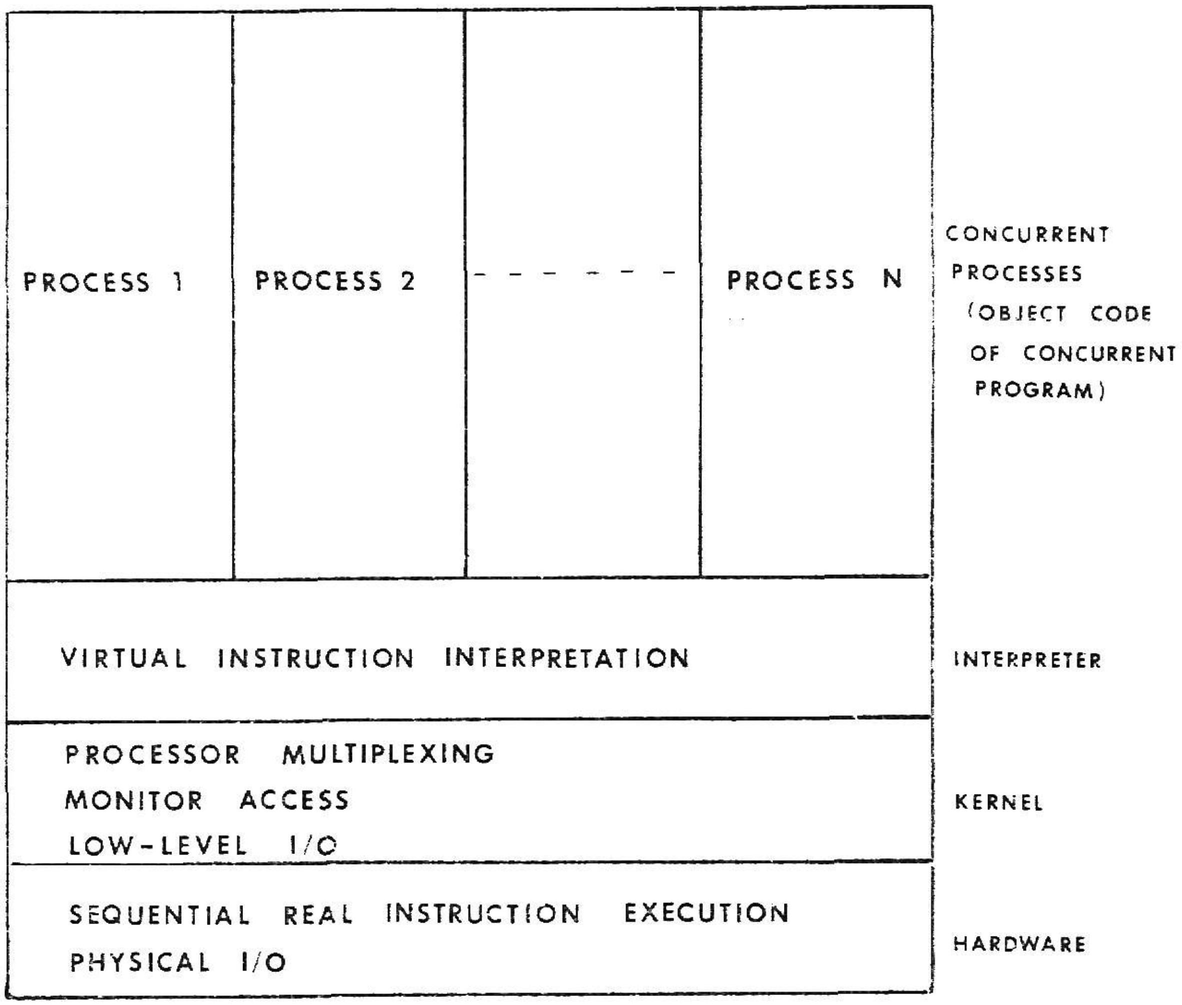

PROCESS 1
PROCESS 2
- - - - - - -
PROCESS N
CONCURRENT PROCESSES (OBJECT CODE OF CONCURRENT PROGRAM)
VIRTUAL INSTRUCTION INTERPRETATION
INTERPRETER
PROCESSOR MULTIPLEXING
MONITOR ACCESS
LOW-LEVEL I/O
KERNEL
SEQUENTIAL REAL INSTRUCTION EXECUTION
PHYSICAL I/O
HARDWARE

FIGURE 5. Logical relationship of hardware, kernel. interpreter, and concurrent program.

ARCHITECTURE AND ORGANIZATION OF THE PASCAL MICROENGINE

3.1 SYSTEM OVERVIEW

The Pascal Microengine [MICR79] is a 16-bit, word addressable stack machine. manufactured by Western Digital Company. It is a desktop microcomputer built around the WD/9000 processor chip set, and is a hardware implementation of the University of California at San Diego (UCSD) virtual P-machine. The five LSI/MOS chips which constitute the processor are the Data Chip which contains the microinstruction decoder and ALU; the Control Chip, where the macroinstruction decoder, microinstruction counter, and I/O control logic are housed; and three 22 X 512-bit MICROM chips for microinstruction storage. The machine has 64K bytes of RAM, two asynchronous serial ports, one 8-bit parallel port, and controllers which give disk units direct memory access. The system at Kansas State University has two eight-inch flexible-disk drives, a CRT, and a printer attached. The hardware is supported by the Pascal Operating System which was written by the University of California at San Diego. The system supports UCSD Pascal, a variant of standard (sequential) Pascal. Assembly language is not supported.

3.2 P-MACHINE

The microengine has an architecture which is similar to that of the Concurrent Pascal virtual machine described earlier. The next three sections describe some significant aspects of that architecture.

3.2.1 Registers

The Microengine has an extensive set of logical registers. Those which correspond to the Q, G, B, S, and H registers of the virtual Concurrent Pascal machine are the IPC, BP, MP, SP, and SPLOW registers. IPC (Interpreter Program Counter) is the program counter. It contains a pointer (byte measure), relative to the start of the code segment (described below), to the instruction _after_ the one currently under execution. BP (Base Pointer), the global base register, points to the markstack of a procedure at the outermost level of nesting in the source code. MP (current Markstack Pointer), the local base register, points to the markstack of the procedure invocation which is currently under execution. SP (Stack Pointer) points to the last word which was pushed onto the stack. SPLOW (Stack Pointer LOWer limit) points to the first free word in the heap, and also marks the last location into which the stack can grow. This is the stack's _lower_ limit since the stack grows from high to low addresses.

In addition, there are other logical registers which have no analogs in the Concurrent Pascal virtual machine. SPUPR (Stack Pointer UPPeR limit) points to the location where stack growth begins. SEGB (SEGment Base) points to the code segment currently under execution. PRIOR (PRIORity), an 8-bit register, holds the CPU priority designation of the current task. Another 8-bit register, FLAGS, contains task state flags, but these have not yet been defined [MICR79]. If the current task gets placed in a linked data structure (ready list or semaphore wait queue, for example) the WAITQ (WAIT Queue) register contains a pointer to the next task in the structure.

The registers mentioned so far are _logical_ registers in the

Microengine P-machine. There are three others which appear to actually exist in hardware. These are RQP (Ready Queue Pointer) which points to the first task on the ready list, CTP (Current Task Pointer) which points to the task presently running, and SDP (Segment Dictionary Pointer) which points to a vector of indirect pointers to code segments.

The machine instructions refer to all registers by number:

-3: RQP   -2: SDP   -1: CTP   0: WAITQ
1: PRIOR   1: FLAGS   2: SPLOW   3: SPUPR
4: SP   5: MP   6: BP   7: IPC
8: SEGB.

In register number 1 PRIOR is the low order byte, and FLAGS is the high order byte.

3.2.2 Object Code File Format

In disk file directory listings, object code files are identified by the extension ".CODE". Stored on disk, a code file (figure 6) consists of a sequence of contiguous 512-byte physical disk blocks. The first block in the file (block number zero) is a header which describes each of the code segments (up to sixteen) which start in block number one.

The header block (figure 6) has four areas of data in it:

1) Segment dictionary-- Sixteen entries, one per segment, regardless of the number of segments actually in the file. Each entry consists of two integer fields. The first is the block number (relative to the header which is block zero) of the first block in the corresponding segment. The second entry is the number of meaningful words in the segment. Meaningless

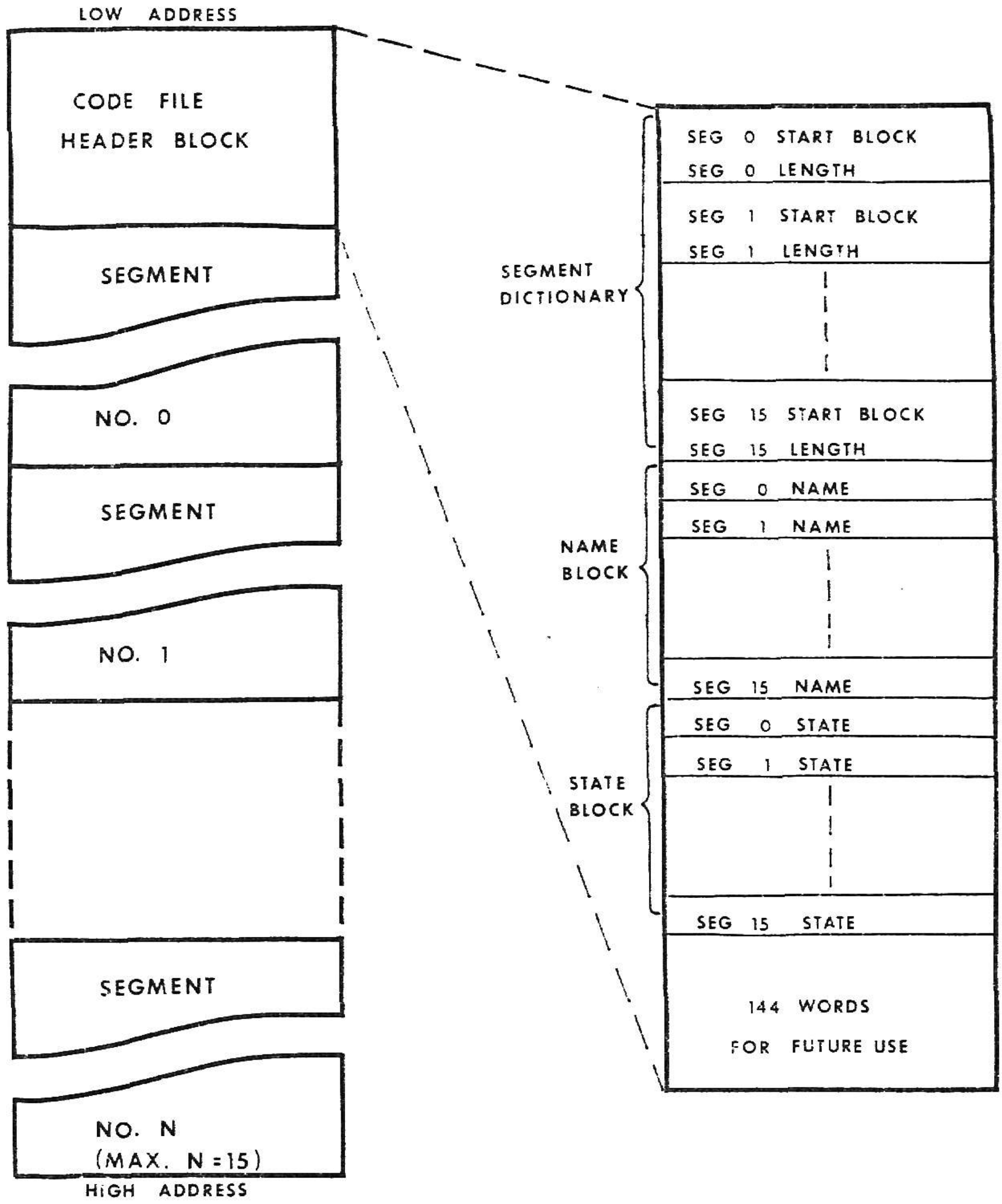


FIGURE 6. Microengine object code file general layout (left) and enlargement of header block (right). Each code segment may extend over several physical disk blocks. File may contain a variable number of segments, but each section of header contains 16 entries, some of which may be null.

words may occur at the end of a segment in order to pad to the end of a block. If the file contains fewer than sixteen segments zeros are used to indicate null entries.

2) Segment names-- Sixteen entries, as above. Each entry is a 16-byte string which is the ASCII character name of the segment, left justified, blank being the pad character. The entry is all blanks if the corresponding segment is null.

3) Segment state descriptors-- Sixteen integer entries, as above. Valid entry values are in the range zero through four. The entries encode linkage information such as the presence of external references and whether they have been resolved. For more information see MICR79, "Linker Conventions and Implementation". The entry for a null segment is zero.

4) The next 144 words are, according to [MICR79], reserved for future use. Most, but not all, of this area contains zeros. The meaning and function of the nonzero entries is unknown, and altering them seems to have no effect on the execution of the file and segment.

Each segment (figure 7) in the file may extend over any number of blocks and consists of a header word, a variable number of routines, a procedure dictionary, a final word of segment information, and possibly some padding. The header word contains the address (measured in words, relative to the start of the segment-- word zero) of the last meaningful word in the segment (the "final word of segment information" just mentioned).

Each routine (figure 7) is made up of a (possibly empty) constant pool and two words of run-time information, followed by the machine instructions themselves. The constant pool is aligned on a word

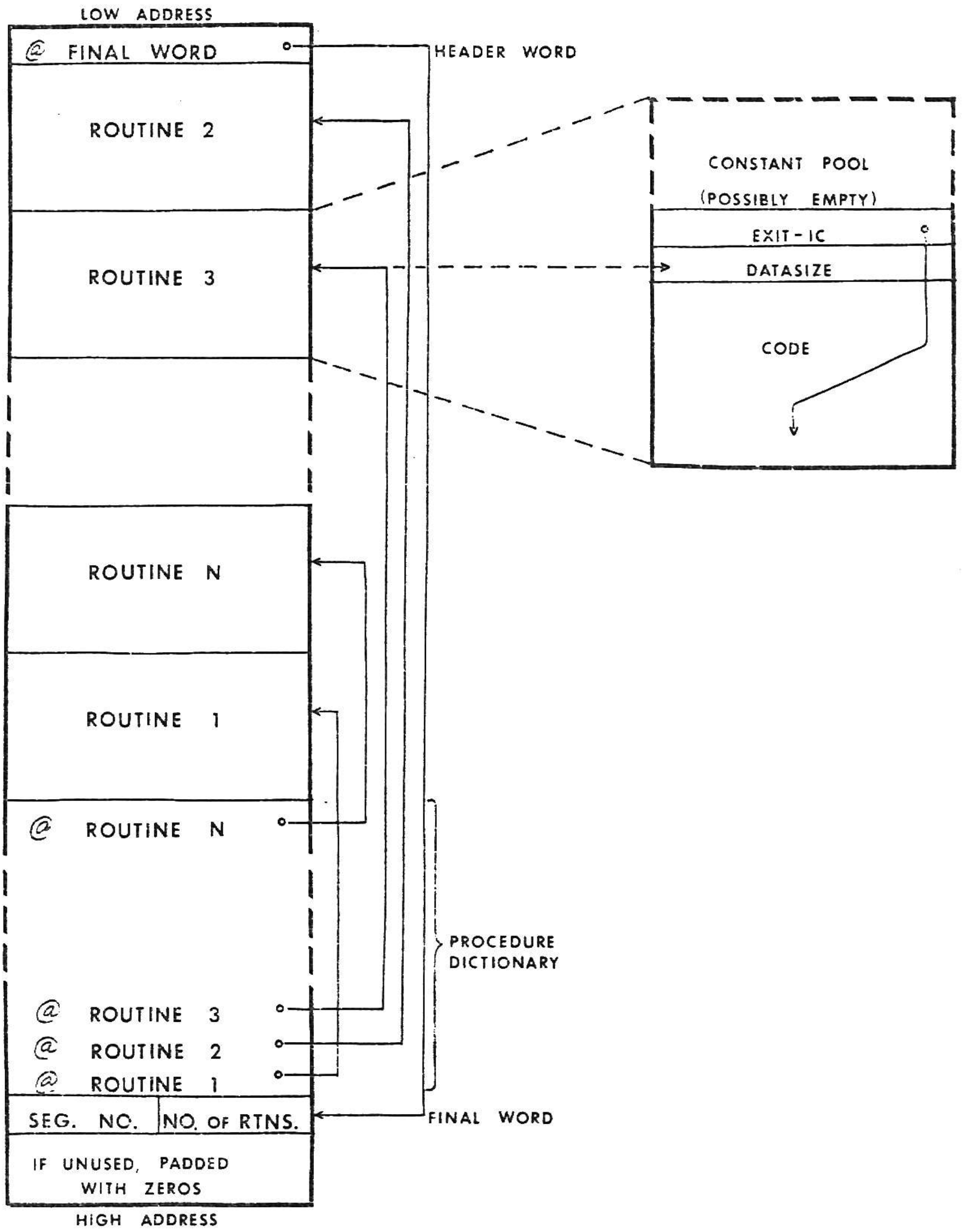


FIGURE 7. Microengine code segment layout (left) and enlargement of a typical routine (right).

boundary, and is similar to the constant pool of the Concurrent Pascal virtual machine. The first information field is the EXIT-IC, a word-aligned pointer (byte measure, relative to the start of the segment-- byte zero), the target of which is the first machine instruction of the routine's epilogue. Usually, the epilogue consists of only an RPU (Return from Procedure-- User) instruction. Presumably, either the hardware or operating system stores this value into the IPC register when a fatal run-time error occurs so that the stack gets cleaned up before returning to the operating system. The second information field, DATASIZE, is the number of words of local variable space required by the routine. The routine's code appears last, and because of prior alignments, it necessarily starts on a word boundary.

The segment's procedure dictionary (of variable length) follows the code of the last procedure and contains a pointer for every routine in the segment (see figure 7). The target of each pointer is the DATASIZE field of the corresponding routine. The pointers (word measure, relative to the start of the segment-- word zero) are arranged in what one would normally consider reverse order. That is, if there are 50 routines in the segment, the pointer for routine 50 comes first. and that for routine 1 comes last. Note that although the routine pointers are in reverse routine-number order, the routines themselves will not be in the same order since the initial statement of a routine is assigned its number before any enclosed routines are assigned theirs. The numbering scheme is further disrupted with each nesting level.

The last meaningful word in the segment contains two pieces of information. The even-address byte contains the segment's identifying number (zero through fifteen) within the code file, and the odd-address byte contains the number of routines in the segment. If this word is not the last one in the block, the rest of the block is padded with zeros. It must be noted that the segments are the unit of execution, not the code file. During any execution, only the segments which are specifically invoked actually enter main memory to have the processor applied to them.

3.2.3 Routine Invocation and Stack Organization

Routine invocations involve a good deal of cooperation between hardware and software. If the called routine is a function, software in the calling routine pushes enough empty space onto the stack to hold the function value when it is returned. If parameters are required, the caller also pushes them onto the stack. Assume for the moment that the calling and called routines are part of the same code segment. The machine instructions which make such intrasegment calls are CPL (Call Procedure-- Local), CPG (Call Procedure-- Global), and CPI (Call Procedure-- Intermediate). During the execution of any of these instructions the hardware performs a number of operations. It uses the procedure number (fetched as an instruction operand) of the called routine as a backward index into the segment's procedure dictionary, fetches the pointer to the routine's DATASIZE field, and pushes onto the stack that number of words for use as the local variable space. It then builds a four-word markstack.

The first markstack word to be pushed on the stack contains two one-byte fields. One is the number of the code segment containing the calling routine's code (or zero, if the calling and called routine are in the same segment). The other has been reserved for future use. The return address in the caller's code (contents of the IPC register) is stored in the next word. The third word is the dynamic link (pointer to the caller's local variables), which is a copy of the caller's MP register. Since UCSD Pascal allows nested routine definitions in the source code, the fourth word is the static link-- a pointer to an enclosing routine's variables. The value used for the static link depends on the nesting level (lexical level) of the called routine, and the particular instruction used. See CPL, CPG, and CPI instructions in [MICR79]. Figure 8 depicts the general notion of static and dynamic links.

After construction of the markstack, the hardware copies the SP register into MP, making the called routine's local variables addressable. Hardware also calculates the IPC value for the first instruction of the called routine and execution continues at that point. The configuration of the stack after a function begins execution is shown in figure 9.

If the called routine is not in the same segment as the caller (that is, external) the invoking instruction is either CXL (Call eXternal Local procedure), CXG (Call eXternal Global procedure), or CXI (Call eXternal Intermediate procedure). Before the UCSD Pascal compiler generates a call to an external user routine, it generates an external call to a well-known procedure in an operating system segment which is always core-resident at run time. That routine fetches the required segment from the disk and pushes it onto the stack. Thus, the compiler and operating system work together to ensure that external procedure-call instructions will never be forced to deal with the invocation of a routine which is not in main memory. The run-time operation of external call instructions is the same as the other call

```
PROGRAM P;
"program variables"

PROCEDURE R1;
"R1 variables"

     PROCEDURE R2;
     "R2 variables"
     BEGIN "code" END;

     PROCEDURE R3;
     "R3 variables"
     BEGIN R2 END;

BEGIN R3 END;

PROCEDURE R4;
"R4 variables"
BEGIN R1 END;

BEGIN R4 END.
```

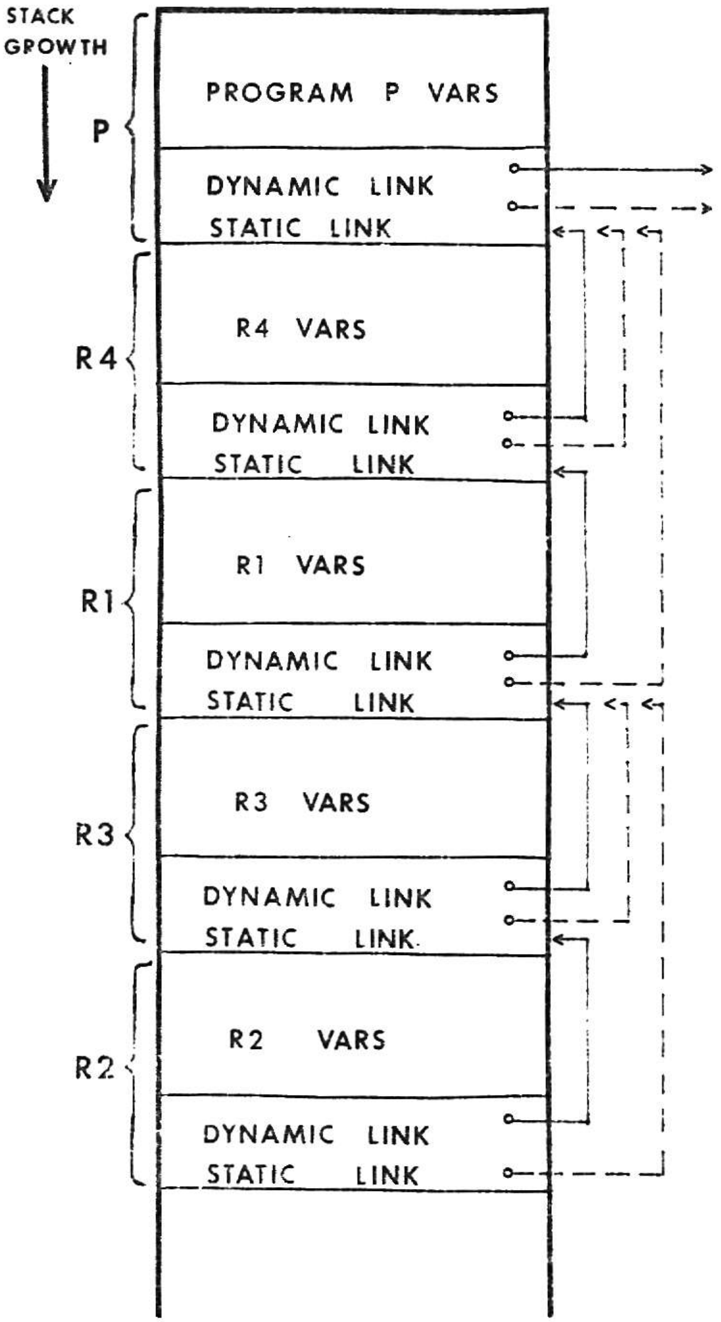


FIGURE 8. The UCSD Pascal source code on the left produces a run-time stack configuration similar to that shown on the right when procedure R2 executes. Dynamic links (solid arrows) always point to the caller. Static links (broken arrows) point to the enclosing routine activation. Since R1 and R4 are enclosed by the same routine (P) their static links point to the same markstack. Likewise, for R2 and R3 whose parent is R1. The markstacks shown here are not complete.

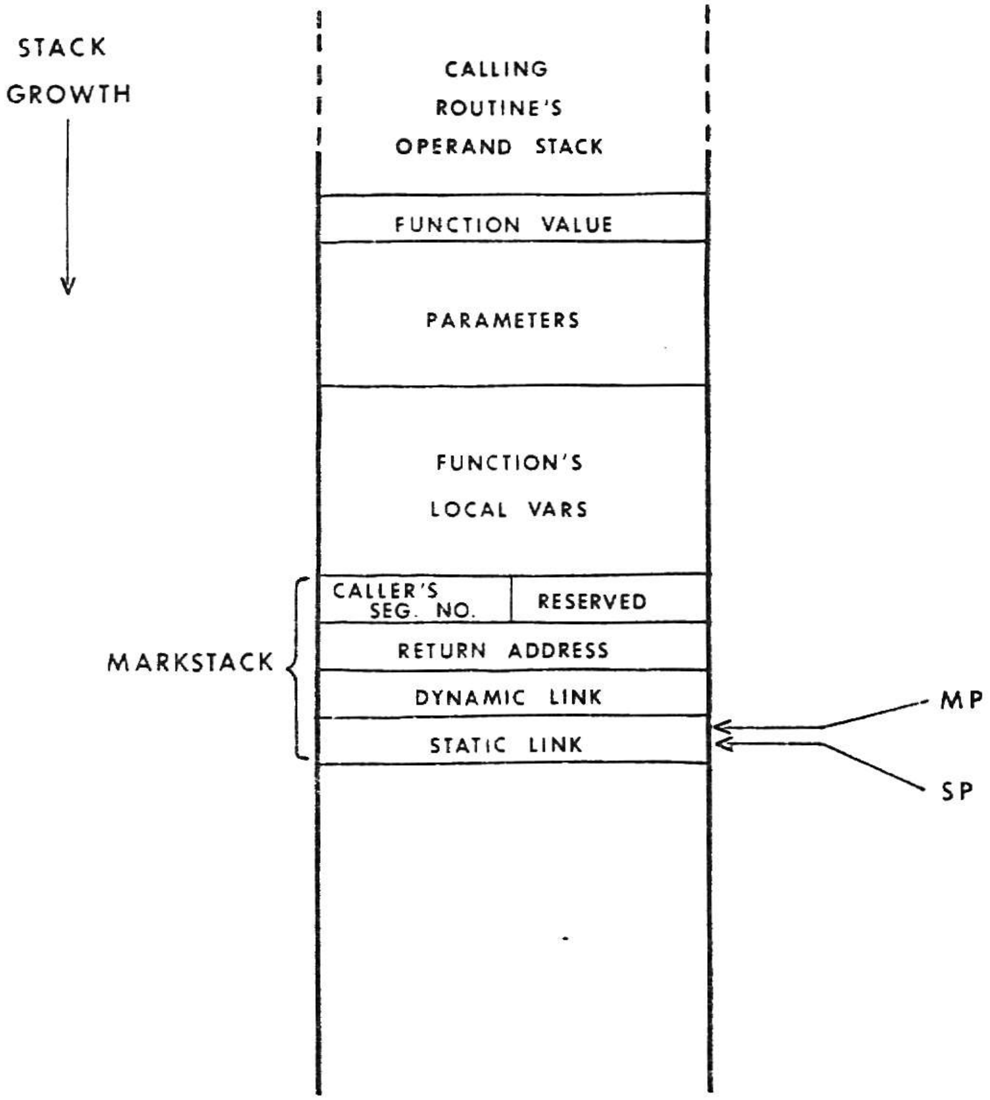


FIGURE 9. Snapshot of a generalized Microengine function call after the function has started executing. Compare the position of the markstack relative to parameters and local variables with that in figure 2.

instructions except that the segment number operand is used to find the address of the code segment containing the called procedure before indexing into the procedure dictionary.

3.3 SUPPORT FOR CONCURRENT PROCESSES

The Microengine has several design features which can facilitate the execution of concurrent processes, although not all of them are supported by the hardware at this time.

The embodiment of a process is the Task Information Block (TIB) which is comparable to a process control block and contains the fields WAITQ, PRIOR, FLAGS, SPLOW, SPUPR, SP, MP, BP, IPC, SEGB, HANGP, XXX, SIBS, MAINTASK, and STARTMSCW. The first ten are the nonnegative logical registers described in section 3.2.1. If the task gets placed in a semaphore wait queue, HANGP can be used to hold a pointer to the semaphore data structure (described below) on which it is waiting. XXX is an unused integer field. SIBS points to an array of segment dope vectors (segment information blocks). The Western Digital documentation [MICR79] states that MAINTASK is of type BOOLEAN and that STARTMSCW is a pointer to a markstack, but does not explain their functions. The reader is left to make his own assumptions.

The target of the pointer SIBS is a vector of Segment Information Blocks (SIB). A SIB contains the fields SEGBASE, a pointer to the core-resident code segment; SEGLEN, the number of words in the segment; SEGREFS, the number of routine calls currently active in the segment; SEGADDR, the absolute physical disk address where the segment resides on secondary storage; and SEGUNIT, the number of the disk drive where the segment resides.

Figure 10 shows several tasks in, say the ready list, and how

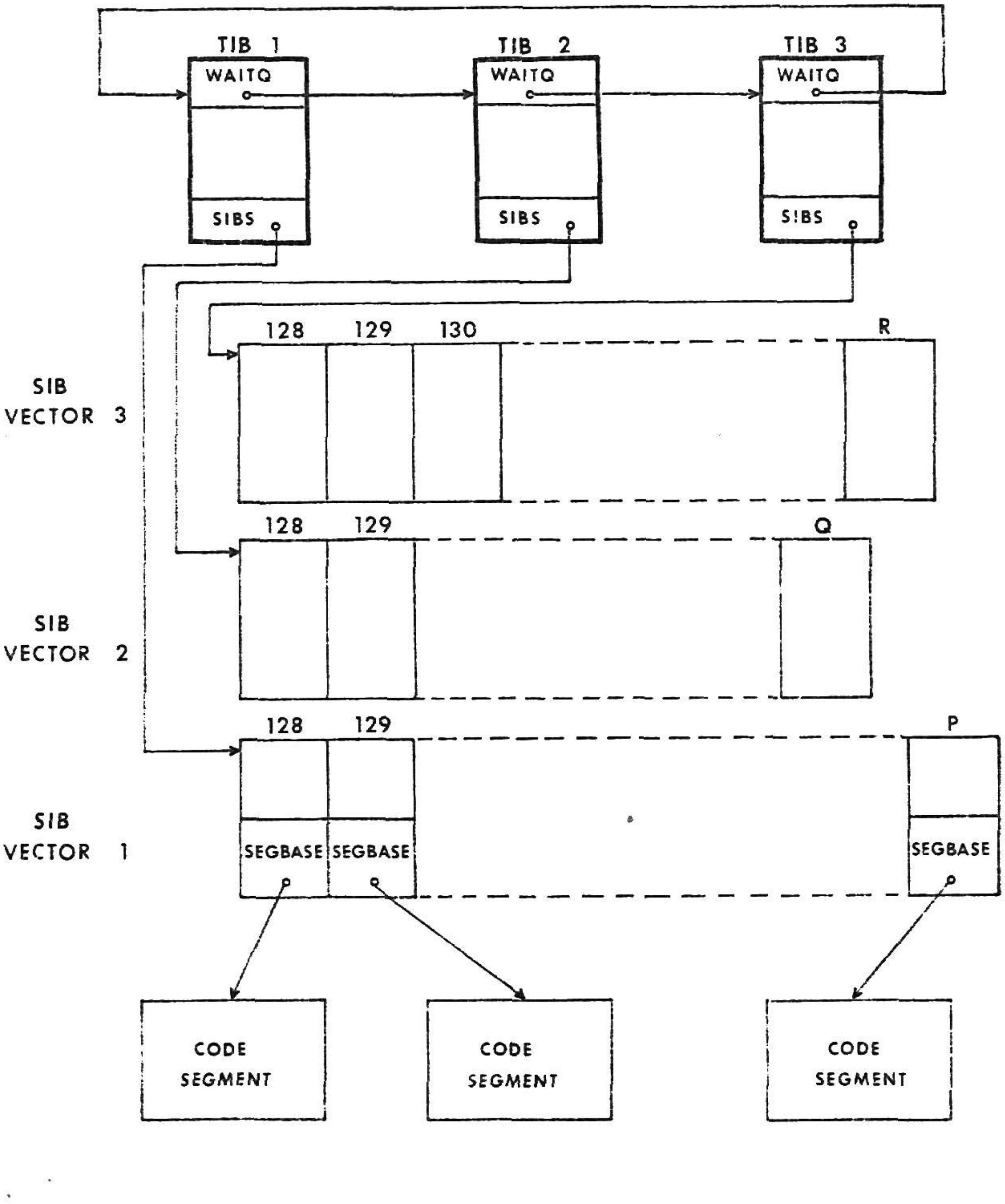


FIGURE 10. Linkage between tasks and private code segments in the Microengine. TIBs and SIBs contain more information than shown here. Each SIB (numbered rectangles in the SIB vectors) has a SEGBASE field, although they are not all shown here.

each one has some private code segments bound to it through the SIBS field and SIB vector. The segments are "private" to each task since it would be difficult (but not impossible) for a casual programmer to execute code which traversed all the links in order to tamper with the code of some other task. In external routine call instructions, the segment number operand refers to a private segment if its value is 128 or greater.

Other segments can be easily called by any task in the system. The SIBs for these "shared" segments are not gathered together in a single data structure like a task's private segments. Instead, pointers to them reside in an array, the segment dictionary, which in turn is pointed to by the SDP register (see figure 11). In external routine call instructions, the segment number operand refers to a private segment if its value is in the range 0 through 127. However, there is an implementation restriction which limits shared segment numbers to the range 0 through 15.

Figure 12 shows how neatly the two segment schemes mesh. The three negative-numbered registers (RQP, SDP, and CTP) point to the ready queue, shared segment dictionary, and currently executing TIB respectively. When a process comes up for execution its private code segments become available through the CTP register, but it does not have easy access to the code of other processes. However, all processes have easy access to the shared code via the SDP register. This means, for example, that operating system processes can keep sensitive code in the relatively secure private segments, while making available that code which really must be public. Notice that context switching only involves resetting the CTP register and advancing the ready queue pointer since process state information is always in the

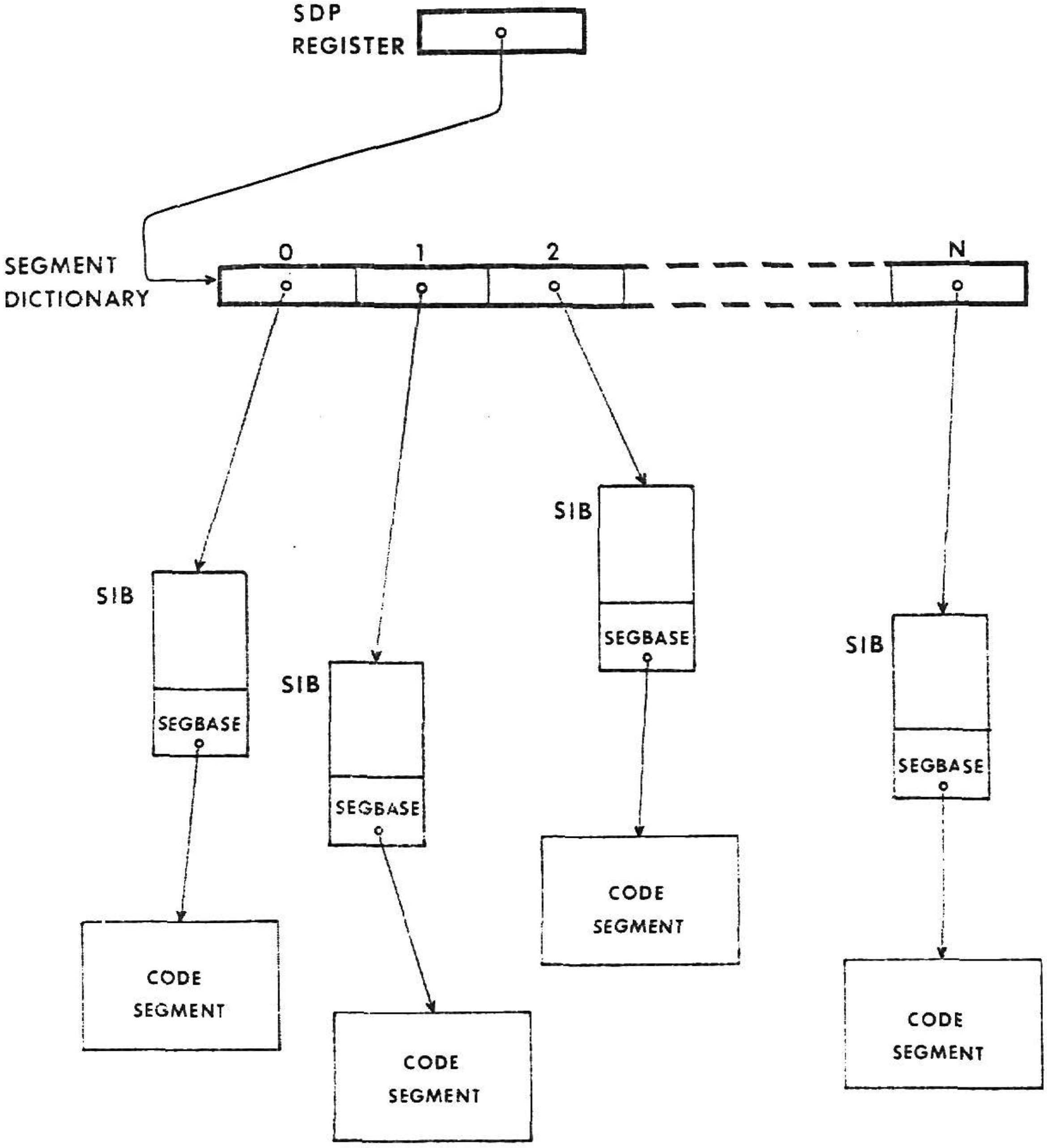


FIGURE 11. Linkage between the Segment Dictionary Pointer register and code segments which can be shared by several tasks. SIBs contain more information than shown here.

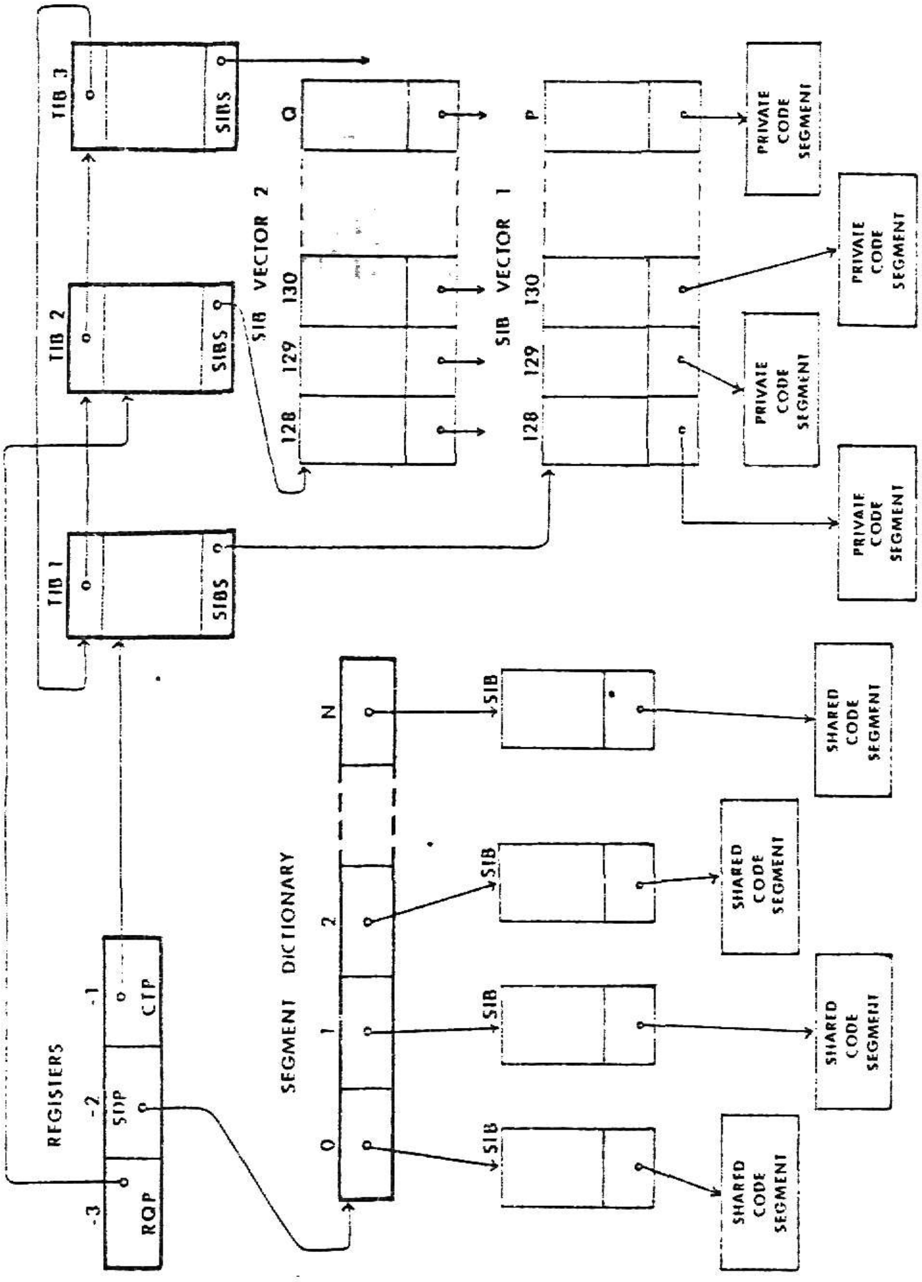


FIGURE 12. Run-time code segment configuration.

TIB.

Machine instructions WAIT and SIGNAL are provided for process synchronization. The semaphore data structure on which these act consists of a COUNT field and a pointer to the semaphore's wait queue. COUNT holds the number of outstanding SIGNALs which have been issued on the semaphore. Presumably, the two instructions behave in a straightforward manner, similar to Dijkstra's P and V operators.

At the present time the hardware does not support interrupts and hardware context switching. According to [MICR79], however, the machine is designed to provide a vectored, fixed-priority interrupt system. There will be eight two-word entries in the vector. One entry for DMA completion, six for serial and parallel port I/O completion, and one for exceptions on either of the serial ports.

3.4 P-MACHINE SEMANTICS FOR CONCURRENT PASCAL

The purpose of a translator for any high level language is to map the source code to the architecture of the target machine. In the case of HCPASCAL, the language existed first, and a virtual machine was implemented to accommodate it. The machine was built according to the needs of the language. For MEPASCAL, the source language and target machine came into existence independently of each other, so the compiler must somehow make use of the existing architectural facilities of the Microengine to carry out the semantic actions of Concurrent Pascal.

3.4.1 Code Segments

In the virtual machine there are four code spaces: kernel. interpreter, concurrent program, and sequential programs (which exist as permanent variables in concurrent processes). These can be accommodated on the Microengine in the following ways.

The interpreter is trivial since its function is performed by the compiler and hardware.

The kernel code should be independent of the concurrent program since it is designed as a relatively stable piece of software to provide support for _any_ Concurrent Pascal program. Therefore, it makes sense to put it by itself in a Microengine shared code segment. If the kernel is always in the same segment (say, segment zero) it is very easy to implement the kernel-call mechanism as a call to a well-known routine within that segment. The kernel operator and operands would have to be pushed onto the stack as parameters by compiler-generated machine code. If it becomes necessary to modify the kernel relatively frequently, a second approach to kernel-call implementation might be more useful. Since the number by which the routine is known to callers can be affected by changing the arrangement or number of routines in the kernel segment, it would be advantageous to isolate the kernel-call handler in a second (well-known) kernel segment where routine number one merely examines the kernel operator and passes the operands to the appropriate routine in the segment. This can be done via routine calls embedded in a CASE statement. See figure 13 for a pseudo UCSD Pascal outline of such a segment. In this report we will assume the entire kernel is in only one segment.

The classes, monitors, and processes of the concurrent program

```
TYPE
KERNOPTR = (INITGATE, ENTERGATE, LEAVEGATE, ENDPROCESS,
            INITPROCESS, REALTIME, DELAYGATE, CONTGATE,
            STOPJOB, WAIT, SYSERROR, IO);


SEGMENT PROCEDURE KCALLHDLR (OPTR: KERNOPTR;
                             OPND1, OPND2, OPND3, OPND4: INTEGER);
CONST
IGATERTN =                   |
EGATERTN =                   |
LGATERTN =                   |
EPROCRTN =                   |
IPROCRTN =                   |
RTIMERTN =          routine numbers in
DGATERTN =            kernel segment
CGATERTN =                   |
STJOBRTN =                   |
WAITRTN  =                   |
SYSERRTN =                   |
IORTN    =                   |

KERNSEG = 0;     "SEGMENT NUMBER OF KERNEL PROPER"

BEGIN
CASE OPTR OF
   INITGATE:    PUSH OPERANDS; EXTERNAL CALL TO KERNSEG, IGATERTN;
   ENTERGATE:   PUSH OPERANDS; EXTERNAL CALL TO KERNSEG, EGATERTN;
   LEAVEGATE:   PUSH OPERANDS; EXTERNAL CALL TO KERNSEG, LGATERTN;
   ENDPROCESS:  PUSH OPERANDS; EXTERNAL CALL TO KENRSEG, EPROCRTN;
   INITPROCESS: PUSH OPERANDS; EXTERNAL CALL TO KERNSEG, IPROCRTN;
   REALTIME:    PUSH OPERANDS; EXTERNAL CALL TO KERNSEG, RTIMERTN;
   DELAYGATE:   PUSH OPERANDS; EXTERNAL CALL TO KERNSEG, DGATERTN;
   CONTGATE:    PUSH OPERANDS; EXTERNAL CALL TO KERNSEG, CGATERTN;
   STOPJOB:     PUSH OPERANDS; EXTERNAL CALL TO KERNSEG, STJOBRTN;
   WAIT:        PUSH OPERANDS; EXTERNAL CALL TO KERNSEG, WAITRTN;
   SYSERROR:    PUSH OPERANDS; EXTERNAL CALL TO KERNSEG, SYSERRTN;
   IO:          PUSH OPERANDS; EXTERNAL CALL TO KERNSEG, IORTN
ENDCASE
END;
```

FIGURE 13. Pseudo UCSD Pascal outline of specialized segment to handle kernel calls. Pushing the operands and calling the external routine are separate machine operations. In high level code this would appear as a normal routine call with parameters.

can be placed in another shared segment where each initial block, routine, and entry routine of the concurrent program is one routine in the concurrent code segment. The segment must be sharable since monitors and classes can be entered by any process which has the appropriate access right. The concurrent segment will always be segment number one. Always placing the concurrent program in segment number one makes it well-known in the system and will aid the implementation of the interface mechanism (explained below).

Sequential programs and their interface with host concurrent processes present special problems. When calling a sequential program, the interpreter for the Concurrent Pascal virtual machine often makes use of a property not found on the Microengine; namely, that real memory is a single, linear address space where all addresses are alike, and that once an address has been obtained, the processor can easily be forced to jump to that location. Sequential program invocation on the virtual Pascal machine is fairly straightforward. The concurrent program must somehow load sequential code into a variable (a large array, for example), and the compiler must generate code to push the variable's address onto the stack. CALLPROG is the virtual instruction which actually starts the program. During its interpretation, the return address is saved on the stack and the virtual program counter (Q) is loaded with the address of the first sequential program instruction which is located a fixed distance from the start of the program variable (whose address is on the stack). On the Microengine, the address of the program variable (in the real memory address space) can be pushed onto the stack by any of several machine instructions. However, it cannot be placed directly into the IPC register to cause a jump to the sequential program since the hardware expects the address in the IPC to be an offset from the beginning of a code segment. A possible solution to this problem is depicted in figure 14 and runs as follows. Assume that when the kernel creates a TIB for a process it also allocates space for two SIBs for segments private to the newly created task. The segment numbers of these can always be 128 and 129, where segment 129 is the sequential program segment. (The purpose of segment 128, the interface segment, will be described below.) As part of the program invocation code, the compiler generates instructions which follow pointers from the CTP register, through the current task's TIB to its SIB vector, and pop the address of the code variable into the SEGBASE field in SIB 129. The next instruction generated is an external call to routine number one in segment 129. The effect of all this at run time is to make the code variable look like a code segment to the hardware, and start the sequential program by means of a normal external routine invocation.

Operating system services are provided to the sequential program through an interface which names the process ENTRY routines to which the sequential program has access. The operation of the interface mechanism is best demonstrated by example. The (meaningless) concurrent program shown in figure 15 contains an interface definition (line 7) for the sequential program J. The program is actually invoked in line 27. The MCPASCAL compiler produces code which performs the following functions in order to get J started:

1) Push onto the stack the address of the first instruction of each ENTRY routine named in line 7, in reverse order of appearance in the source code. That is, push the address of PE3, then that of PE2, then that of PE1;
2) Push the parameters (the values 1 and 2, in the example);

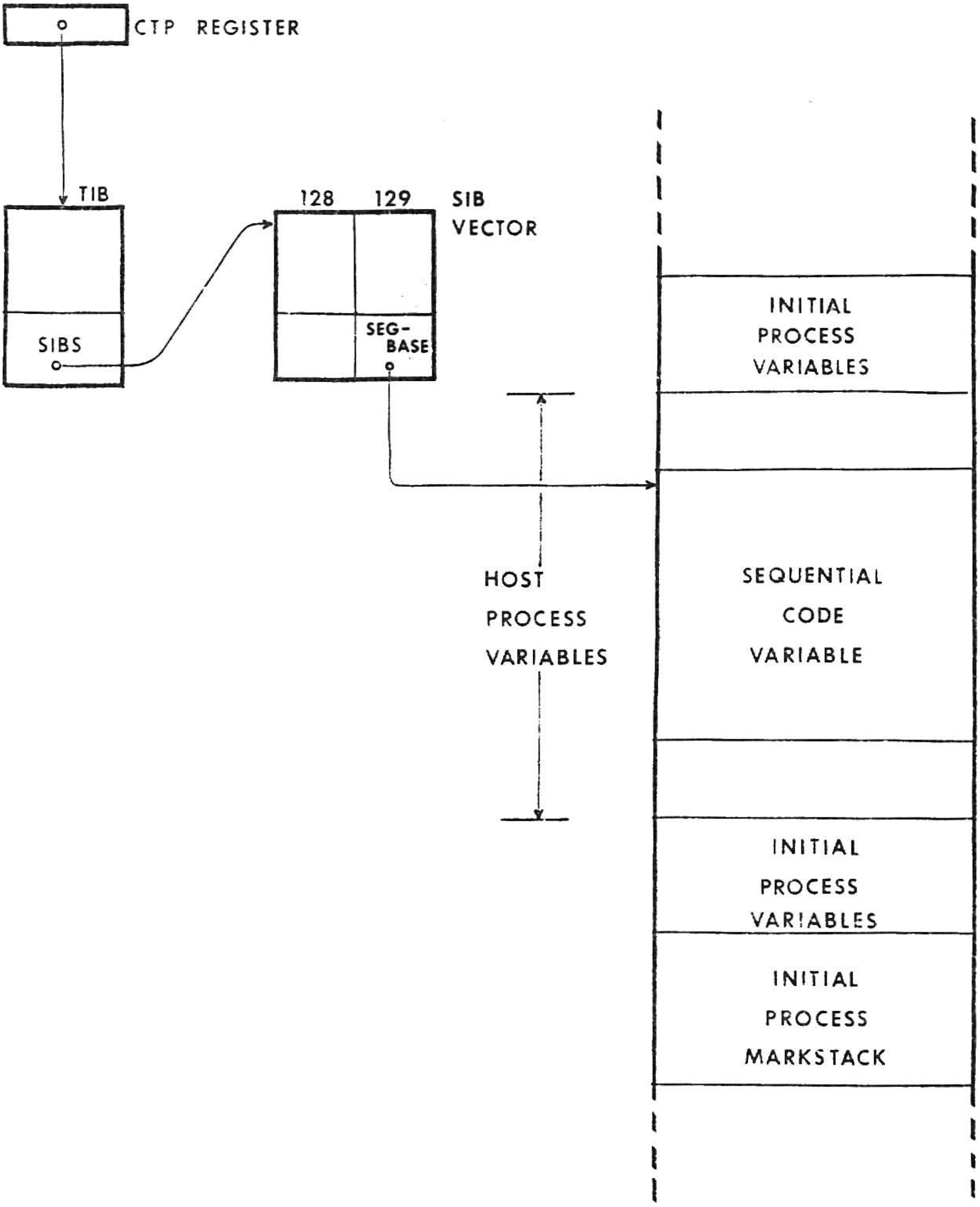


FIGURE 14. Configuration of pointers just before calling a sequential program.

```
1     TYPE
2          PRC = PROCESS;  +1000
3          TYPE    CODE = ARRAY[1..1000] OF INTEGER;
4          VAR     CODEVAR: CODE;
5
6          PROGRAM J (A, B: INTEGER; C: CODE);
7          ENTRY PE1, PE2, PE3;
8
9          PROCEDURE ENTRY PE1 (PARAM1: INTEGER;
10                             VAR PARAM2: INTEGER);
11         BEGIN
12         PARAM2 := PARAM1 + 100
13         END;
14
15         PROCEDURE ENTRY PE2 (PARAM1: INTEGER;
16                             VAR PARAM2: INTEGER);
17         BEGIN
18         PARAM2 := PARAM1 + 200
19         END;
20
21         FUNCTION ENTRY PE3 (PARAM: INTEGER): INTEGER;
22         BEGIN
23         PE3 := PARAM + 300
24         END;
25
26         BEGIN
27         J (1, 2, CODEVAR)
28         END;
29
30
31    VAR    PRCV: PRC;
32
33
34    BEGIN
35    INIT PRCV
36    END.
```

FIGURE 15. Sample Concurrent Pascal program containing an interface definition (line 7) for the sequential program defined in line 6 and invoked in line 27.

3) Push the address of the variable containing the sequential program code;

4) Execute the CALLPROG instruction as described above.

A snapshot of the run-time stack at this point is shown in figure 16. Notice that the addresses of the process ENTRY routines (interface routines) in the stack constitute a jump table, built at run time. The sequential program calls interface routines by executing the virtual instruction CALLSYS(PREFIX_INDEX). During the interpretation of CALLSYS, the value of PREFIX_INDEX is used as an index into the jump table on the stack and the processor branches to the corresponding ENTRY routine in the concurrent process. This mechanism allows a great deal of independence between the concurrent and sequential programs. The only point on which they must agree is the order in which the interface routines are to appear.

The design of an interface mechanism for the Microengine proved to be a significant challenge. The problems stem from two Microengine architectural features: routines are known and called by their number, and the number of the called routine must be supplied to the invoking instruction as an immediate operand, not a stack operand. The first item precludes the construction of a run-time jump table, but if the sequential program is allowed to call process ENTRY routines directly, two major points of independence are lost. First, the sequential program would need to know, at compile time, the identifying numbers (not just the order) of the entry routines it is allowed to use. Second, the number and arrangement of routines in the concurrent program could be changed, independent of the sequential program, only if it could be <u>guaranteed</u> that the routine numbers of the process entries would be unaffected-- an extremely difficult, if not

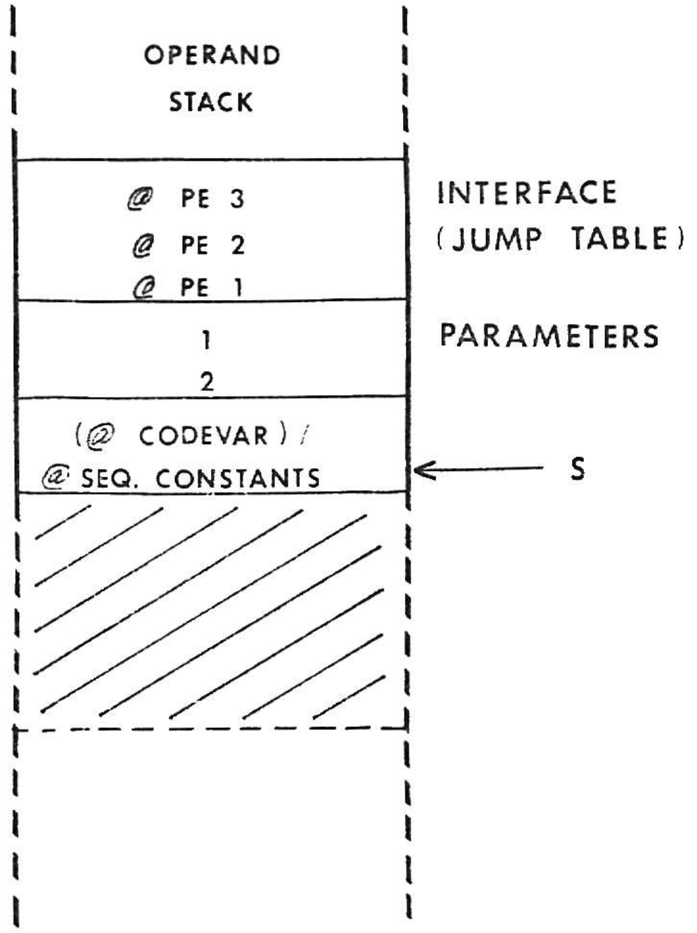


FIGURE 16. Run-time stack after execution of the virtual instruction CALLPROG which invokes a sequential program. CALLPROG uses the sequential program variable address which it finds on top of the stack and replaces it with the address of the sequential program's constant pool. The sequential program's markstack will be be built later in the lined area.

impossible, task. If routine call instructions took their operands from the stack, then the numbers of the interface routines could be placed on the stack as parameters to the sequential program. Invocation of an interface routine would only require pushing the appropriate parameter onto the top of the stack and initiating a routine call. However, as mentioned above, the architecture prevents such a maneuver.

The required Microengine mechanism must be a run-time mapping of prefix indices to interface routine numbers in the concurrent segment. The map can be realized in an interface segment (see figure 17) built by the compiler whenever it encounters the invocation of a sequential program which requires an interface. Some details of the scheme, such as the time at which the interface segment should be loaded into main storage, and the accommodation of more than one interface in a single process have not yet been resolved. The interface segment (always segment 128) contains only one routine which itself consists largely of a CASE statement. The case labels are the prefix indices which are known to the sequential program. The code of each case is an external call to a process ENTRY routine in the concurrent segment. If the routine numbers change because of some structural change in the concurrent program, the operands to the routine calls will be adjusted accordingly when it is recompiled.

Sequential program invocation can be the same as described above for the virtual machine, except that it is no longer necessary to push the jump table onto the stack. The invocation of a process ENTRY routine will be quite different. The sequential program will push onto the stack the prefix index of the ENTRY routine as a parameter. It then will call procedure one in segment 128, the interface mapping

```
SEGMENT PROCEDURE INTERFACE (PREFIX_INDEX: INTEGER);
BEGIN

RANGECHECK (PREFIX_INDEX);
CASE PREFIX_INDEX OF
   1: EXTERNAL CALL TO CONCURRENT SEGMENT, ROUTINE W;
   2: EXTERNAL CALL TO CONCURRENT SEGMENT, ROUTINE X;
   3: EXTERNAL CALL TO CONCURRENT SEGMENT, ROUTINE Y;
   .                            .
   .                            .
   .                            .
   .                            .
   .                            .
   N: EXTERNAL CALL TO CONCURRENT SEGMENT, ROUTINE Z
ENDCASE

END;
```

FIGURE 17. Pseudo UCSD Pascal description of concurrent/sequential interface segment. PREFIX_INDEX is the index number of the prefix routine the sequential program is calling. W, X, Y, and Z are the numerical identifiers which the routine-calling instructions use to invoke the corresponding process ENTRY routine. RANGECHECK causes a run-time error if PREFIX_INDEX is not in the range 1..N.

routine. The mapping routine will use the parameter as the case selector, and call the coresponding process ENTRY routine in the concurrent segment.

Figure 18 shows the proposed arrangement of segments required for a Concurrent Pascal program to run on the Microengine. Two segments (kernel and concurrent program) are shared by all tasks, and two (interface and sequential program) are privately associated with their controlling task.

3.4.2 Data Spaces

Just as code areas must be mapped to the Microengine architecture, so must Concurrent Pascal data spaces. They will be handled in the manner described here. Figure 19 shows the general state of affairs in the stack after the initial process has started executing its initial routine. This state can be reached through the following sequence of events. At IPL time the system is powered up and a human operator presses the RESET button on the rear of the processor cabinet. This causes the hardware to read into main memory a fixed area of data from a disk unit which is well-known to it. The hardware assumes that it has just read in the first part of a bootstrap loader and proceeds to load the kernel by executing that information. Once the kernel code is in place it can begin execution and fetch the concurrent program code from a disk unit. At this point the stack consists of the kernel code segment, space for its variables, and a markstack. As concurrent object code is read into the processor, it is pushed onto the kernel's operand stack. Once the segment has been loaded, the kernel sets the segment dictionary pointer, builds a SIB, and puts the address of the segment into it. Routines in the

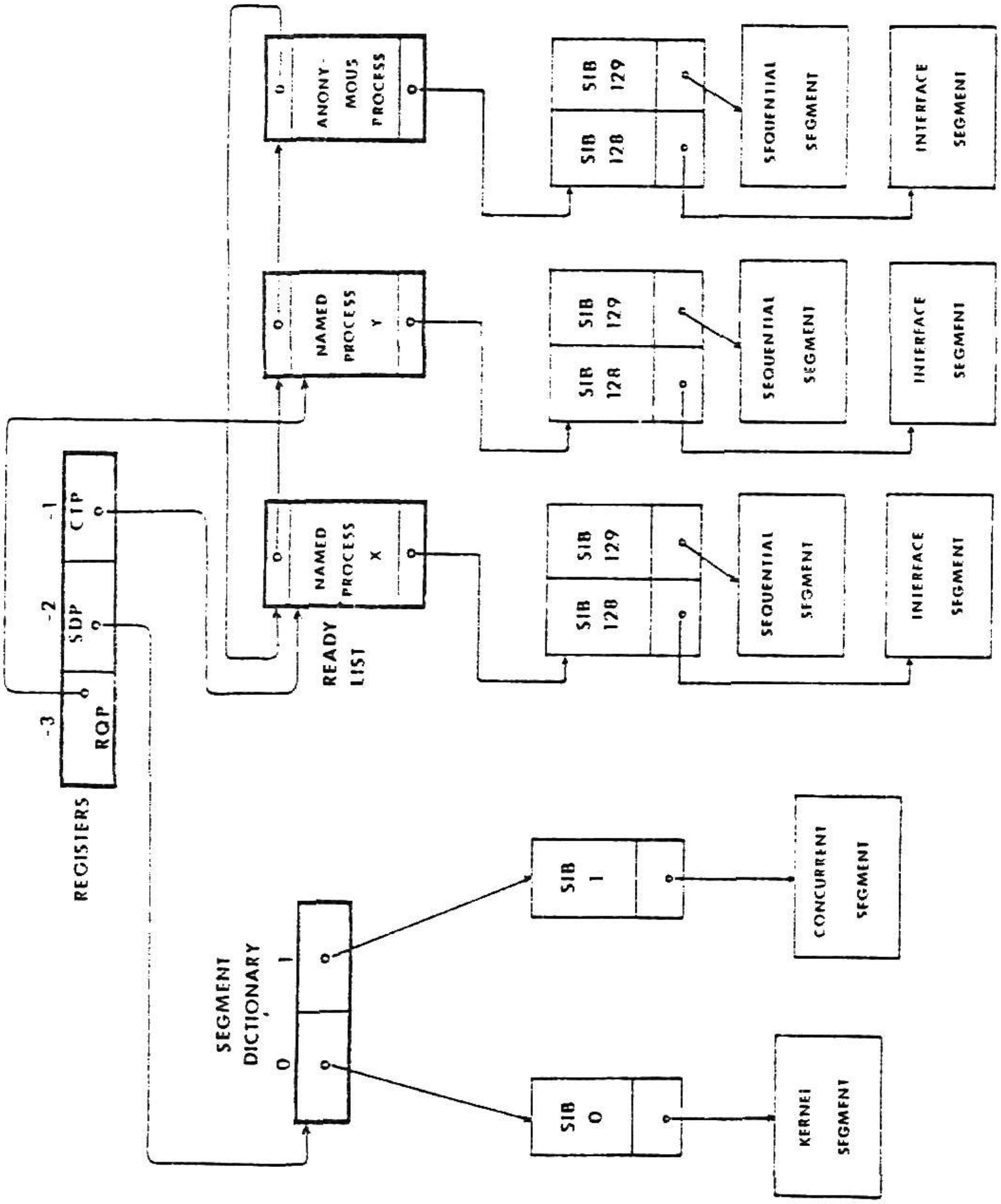


FIGURE 18. Concurrent Pascal program on the Pascal Microengine.

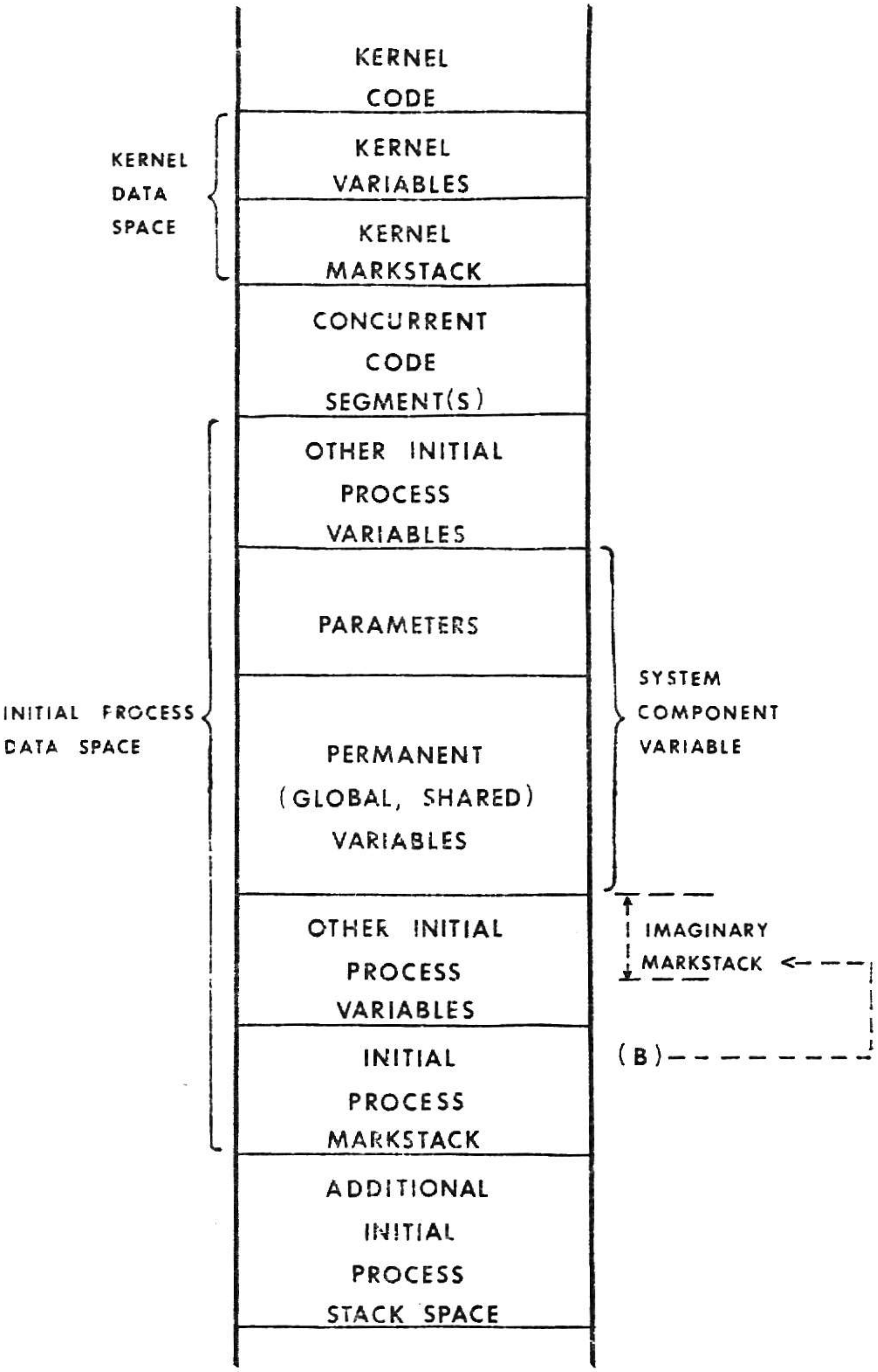

KERNEL CODE
KERNEL VARIABLES
KERNEL MARKSTACK
KERNEL DATA SPACE
CONCURRENT CODE SEGMENT(S)
OTHER INITIAL PROCESS VARIABLES
PARAMETERS
SYSTEM COMPONENT VARIABLE
INITIAL PROCESS DATA SPACE
PERMANENT (GLOBAL, SHARED) VARIABLES
OTHER INITIAL PROCESS VARIABLES
IMAGINARY MARKSTACK
(B)
INITIAL PROCESS MARKSTACK
ADDITIONAL INITIAL PROCESS STACK SPACE

FIGURE 19. General layout of kernel data space and initial process stack space with a system component. While any part of the system component's code is executing, the global base register (B) points to the component record. The concurrent code segment is shown grossly out of scale for the sake of brevity.

concurrent segment can now be called. The concurrent program becomes active when the kernel calls its initial routine, routine number one.

Invocation of the initial routine causes the hardware to allocate stack space for its variables and build a markstack (see section 3.2.3). The amount of variable space to be allocated was determined by the Concurrent Pascal compiler (MEPASCAL). If any of the initial process' variables are system components, they will appear here, as shown in figure 19. The embodiment of a system component variable (instance of a system component type) is a record in the variable space of the initial process. The record has the same format as the top region of the stack, just after a routine call instruction has been completed. For example, suppose that a class is defined as

```
TYPE     CL = CLASS (P1, P2, P3: INTEGER);
         VAR   V1, V2: REAL;

                    PROCEDURE ENTRY X;
                    BEGIN
                    "code"
                    END;

                    PROCEDURE ENTRY Y;
                    BEGIN
                    "code"
                    END;

                    BEGIN
                    "initial code"
                    END;
```

and that the concurrent program variables declarations include

```
VAR     CL1, CL2, CL3, CL4: CL;  .
```

Then in the initial process variable space, along with any other concurrent program variables, there would be four records (CL1, CL2, CL3, and CL4), each containing five fields. Three integer fields at the high-address end of the record are the parameters P1, P2, and P3. Two real fields at the low-address end of each record are the permanent variables of the class, V1 and V2. Any time the class instance CL2, for example, starts executing, the global base register (B) will be forced by compiler-generated software to point to an imaginary markstack adjacent to the high-address end of the record for CL2, as shown in figure 19. Unlike the local variable areas of procedures, functions, and ENTRY routines, which appear upon activation and disappear upon return, the initial process variables disappear only if the whole concurrent program returns to the kernel. Hence, the permanence of component permanent variables is realized. Before initializing a component variable, software pushes its parameters onto the stack. The actual INIT code sequence then pops them into the component variable's record. (For a process initialization, the parameter movement is handled by the kernel.) In this way access rights are "remembered" after the initial routine returns to its caller.

As figure 20 shows, process stack space is allocated contiguously in the order of process creation. The amount of space to be allocated to a process is determined at compile time. Once the space is allocated, it exists forever during execution. Within that space the stack for the process will rise and fall as code is executed and routines are activated. Since each process has its own stack operating concurrently with those of other processes, there will be several stacks and stack pointers existent in the machine at the same time. However, there is no ambiguity for the hardware regarding which stack pointer to use, nor is there any need for software to reset SP during context switches, since every process has its own SP in its TIB. When a TIB gets switched onto the processor (when the TIB becomes the target of the CTP register) the correct SP comes with it.

The amount of stack space needed by a process can be supplied to

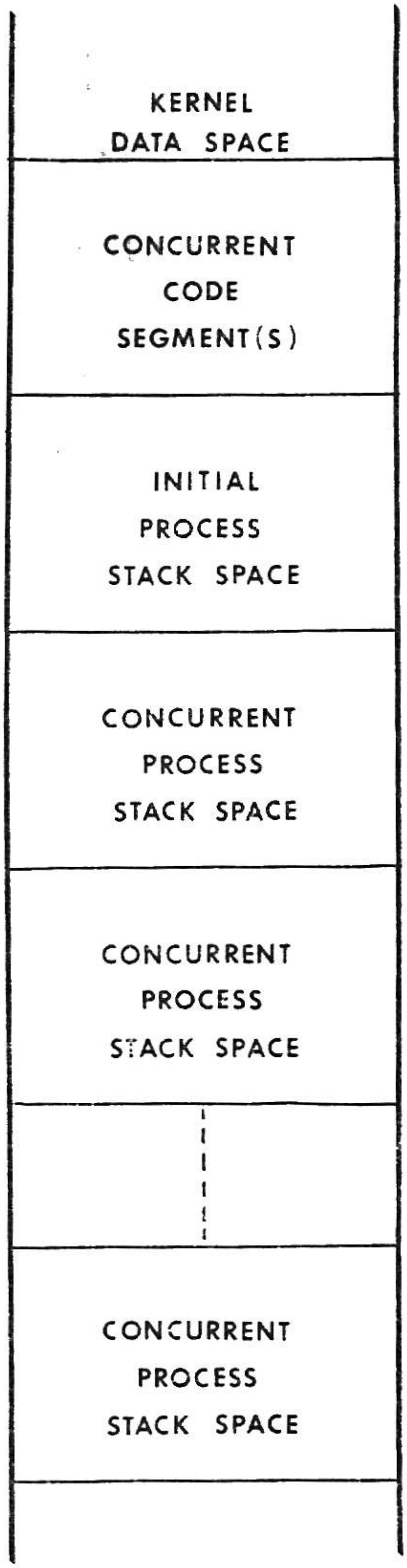


FIGURE 20. General scheme of stack space allocation.

the kernel as a parameter to the kernel routine which creates processes. Even though the space information is provided at process-creation time, it will not be used until the kernel is called upon to create the next process. Figure 21 shows how stack space for a new process (process 2) must be allocated in relationship to the most recently allocated space (process 1). During execution of routine call instructions the hardware apparently makes use of the SP register to determine the starting address of the routine's local variable area and, by implication, the location of the new markstack. This means that when process 2 is created, the kernel must set the SP register in process 2's TIB so that it points to the last allocated word in process 1's stack space. When the new process comes up for its first execution time slice it will execute a call to its own initial routine, and the process markstack will automatically be built in the correct location. The placement of the initial routine's markstack is somewhat critical. If it spills into the previously allocated area, it could be destroyed, but placing it too far away wastes memory. The new process' global (permanent) variables have been allocated in the variable space of the anonymous initial process, so the new variable space is null. It seems possible to allocate process global variables here since they would be deallocated only if the process initial routine tried to return to its caller, but placing them in the initial process area maintains a consistency with the allocation scheme for class and monitor permanent variables.

If the method just described is to ever work successfully, a mechanism is required which will allow one process to enter the kernel routine which creates processes, and allow two to exit it safely. (The second process must not be allowed to cut back a nonexistent

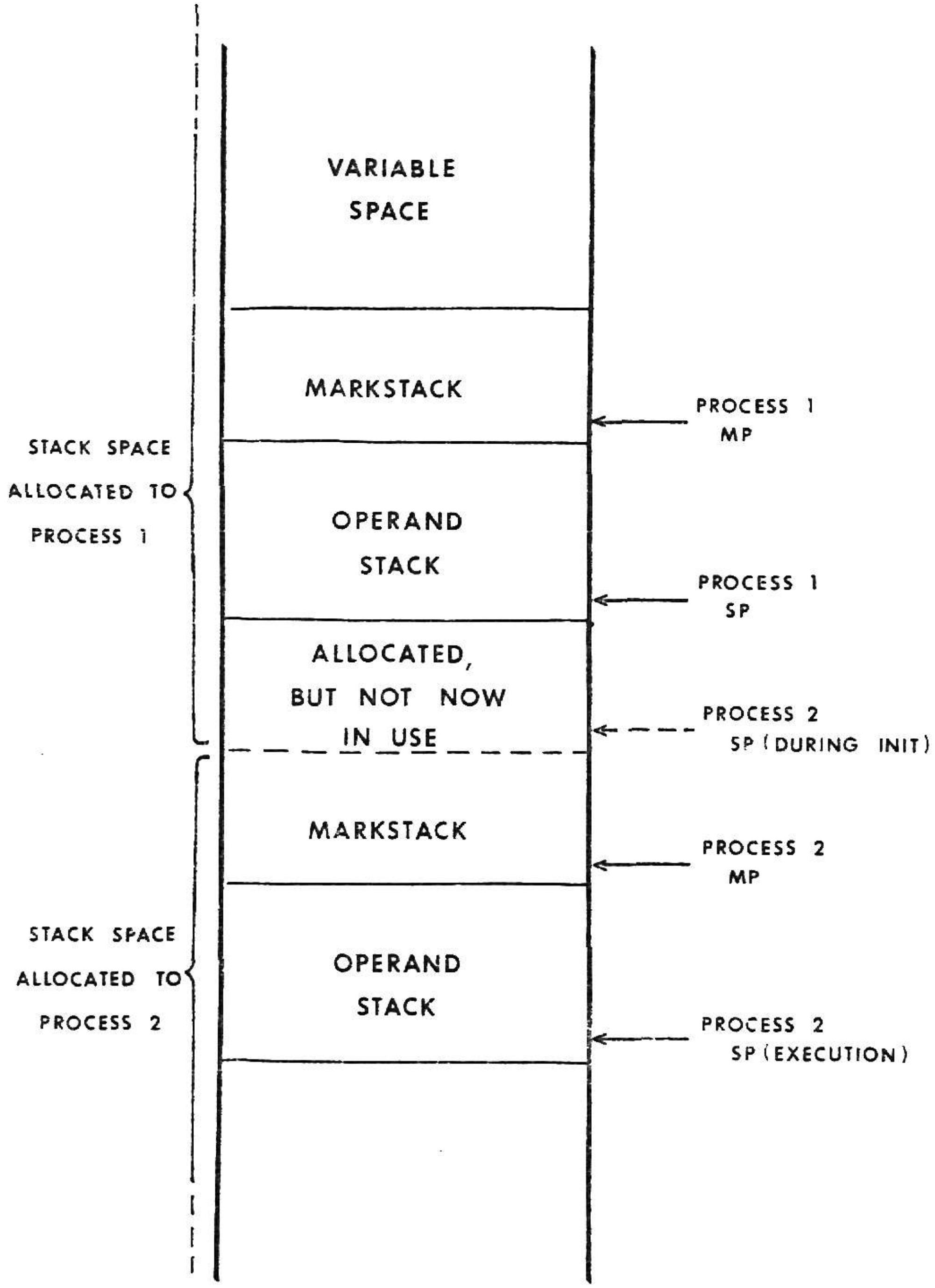

VARIABLE SPACE
MARKSTACK
PROCESS 1 MP
STACK SPACE ALLOCATED TO PROCESS 1
OPERAND STACK
PROCESS 1 SP
ALLOCATED, BUT NOT NOW IN USE
PROCESS 2 SP (DURING INIT)
MARKSTACK
PROCESS 2 MP
STACK SPACE ALLOCATED TO PROCESS 2
OPERAND STACK
PROCESS 2 SP (EXECUTION)

FIGURE 21. Boundary between the stack spaces allocated to two concurrent processes. When the kernel creates process 2, SP must point to the location indicated by the broken arrow. Notice that there are no variables here for the process 2 initial routine-- they are the permanent variables which have been allocated space in the data area of the anonymous initial process.

stack.) And yet, the hardware should do as much of the work as possible. This turns out to be no small task. Assume for the moment that the anonymous initial process is to start a process called PRC. The proposed mechanism calls for the compiler to generate code in the initial process which (in pseudocode) looks like:

```
CALL KERNEL TO CREATE PROCESS
SKIP NEXT INSTRUCTION
CALL PRC'S INITIAL ROUTINE
OTHER CODE,
```

and for the kernel's process-creation routine to be similar to:

```
BUILD TIB FOR PROCESS PRC
BUILD OTHER PROCESS STRUCTURES (SIB, for example)
FETCH DYNAMIC LINK (points to initial process SKIP)
ADD 2 BYTES TO DYNAMIC LINK VALUE
PUT (DYNAMIC LINK + 2) INTO PRC'S IPC
SET SP REGISTER IN PRC'S TIB
PUT PRC'S TIB IN READY LIST
RETURN TO CALLER.
```

Execution proceeds in the following manner. The anonymous process executes a normal call to the process-creation routine in the kernel. That routine builds a TIB and related data structures for PRC (see figure 18). Next, the IPC address of the SKIP instruction in the initial process is fetched. This is possible by means of the following steps:

1) Load the contents of the MP register onto the stack. This puts the memory address of the process-creation routine's markstack onto the stack.

2) Add one word to the top-of-stack value to yield the address of the dynamic link field in the markstack.

3) Push onto the stack the word pointed to by the top-of-stack word. That is, push indirectly the dynamic link.

The dynamic link is the IPC value of the SKIP instruction in the initial process. Two (the skip instruction is two bytes long) is added to it to yield the IPC value of the CALL PRC instruction. That value (dynamic link + 2) is put into the IPC register field of PRC's TIB so that PRC will start execution there when it gets its first slice of processor time. PRC's SP register is initialized as discussed above, and the TIB is placed in the ready list. The anonymous process executes a normal return from the kernel and next executes the SKIP in the initial process. It then continues on to its other code. If the CALL were not skipped, the initial process would end up in the code of PRC, not its own. On the other hand, when PRC comes up for its first execution, it must execute the CALL so that it ends up in its own initial code. If PRC attempts to return from its initial routine, it will be prevented from entering the code of the initial process, from which it was called, by compiler-generated code which calls a kernel routine before the RETURN instruction can be executed. That routine (ENDPROCESS) simply removes PRC's TIB from the ready list.

STRUCTURE OF THE CONCURRENT PASCAL COMPILER

4.1 OVERVIEW AND SUMMARY OF PASSES ONE THROUGH FIVE

MCPASCAL is a seven-pass, recursive descent compiler written in Pascal32, a variant of sequential Pascal which generates Interdata 8/32 object code. Figure 22 shows the various parts of the compiler on Kansas State University's Interdata 8/32. MCPASCAL.CSS is an operating system command file which makes logical device assignments, allocates temporary intermediate code files and a permanent object code file, and initiates the sequential Pascal program MCPASCAL. MCPASCAL is the compiler's driving program and performs the following functions:

1) Scans the string of driver options requested by the user and saves them for future reference;

2) Invokes the passes of the compiler (MCPASSx-- Managers, Concurrent, PASS number x) in the proper sequence;

2) Invokes the program MNEM (explained below) as indicated in the driver options specified by the user;

4) Monitors the compilation and reports its progress (for example, passes completed and presence/absence of compilation errors) to the user's console.

The driver communicates with the programs it calls through variables of type ARGLIST and PROGRESULT which are defined in the prefix of each program.

The compiler passes use four disk files as shown in figure 23. The two temporary files contain the intermediate code produced by each pass. When an intermediate code file is no longer needed, it is overwritten by subsequent passes. For example, MCPASS3 writes over the output of MCPASS1 when it produces its own output.

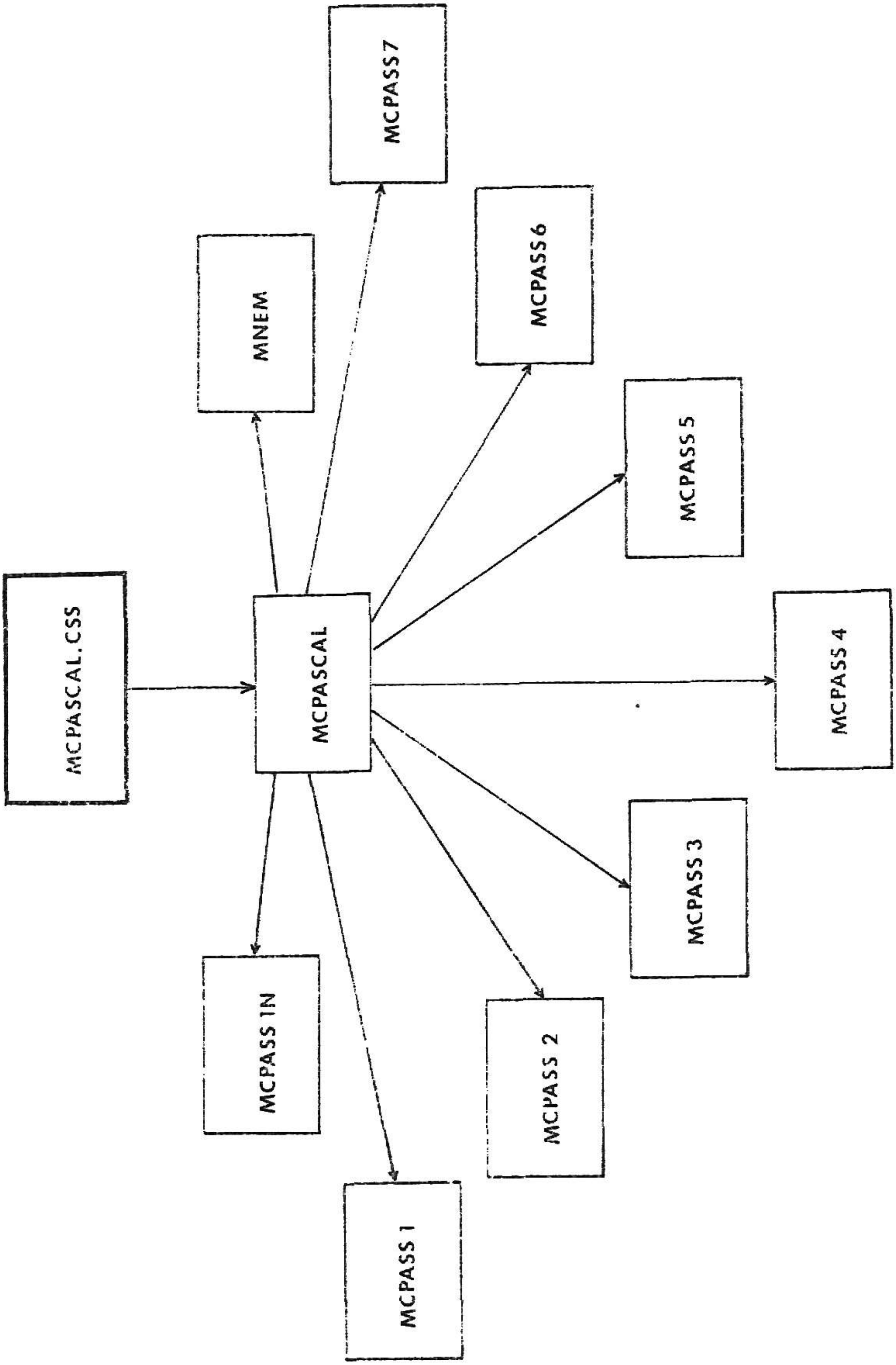


FIGURE 22. Control structure of the MCPASCAL compiler. .

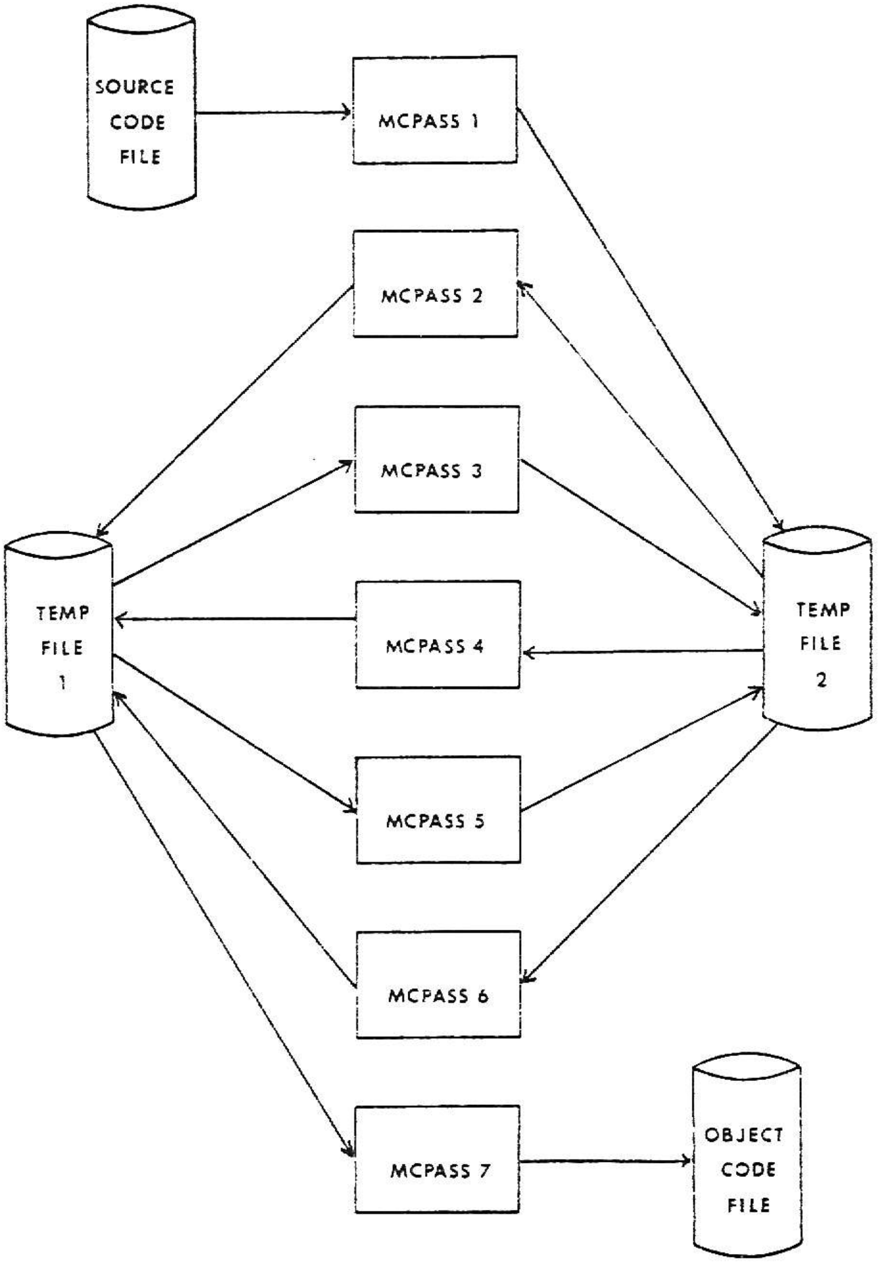


FIGURE 23. Flow of intermediate code in the Mcpascal compiler system.

MNEM (MNEMonics) is a sequential program which accepts the number of a compiler pass as a parameter. It reads the intermediate code output of that pass and prints the contents on the same device (or file) as the source code listing generated by the first pass. The output is formatted, and operators appear as mnemonics instead of integers. Operands appear as integers and are enclosed by parentheses. MNEM is described more fully in Appendix A. This mechanism replaces the test mechanism built into the compiler and invoked as a compiler option. The test mechanism prints the intermediate code as integers. Operators are identified by preceding them with the letter "C".

The first pass of the compiler (MCPASS1) is the lexical scanner. In general terms, it translates the source program into a sequence of 16-bit integers which represent the program tokens, and produces a listing of the source file. It also performs three rather specific functions unrelated to lexical analysis. First, it generates the null set long constant. Second, it allocates heap space for the PASSLINK, a record data structure which remains in the heap between passes, and through which the passes communicate with each other. The third specific function is the analysis and initialization of compiler options which may appear at the beginning of the source program.

MCPASS1N (MCPASS1 No source) is the same as MCPASS1 except that it does not produce a source listing. Its existence allows the source/no-source option to be specified as a driver option when the compiler is invoked, rather than as a compiler option embedded in the source code. If "NS" is in the string of driver options, MCPASS1N is invoked as the first compiler pass instead of MCPASS1.

MCPASS2 performs recursive descent syntax analysis on the token stream produced by the previous pass. At the end of the pass it saves the number of jump labels it created in the PASSLINK record.

Scope (name) analysis of program identifiers is performed by pass three. It enforces rules which, for example, demand that within a single block or record, identifiers have only one meaning, and forbid a nested component definition from referring to the parameters or variables of an enclosing component. In addition, the calculated length of the constant pool is saved in the PASSLINK, even though the constants themselves remain sprinkled through the intermediate code stream.

MCPASS4 performs the semantic processing of the declaration portions of the program. It analyzes types, assigns addresses to variables and parameters, and assigns block labels to routines. Semantic rules are also enforced here. For example, strings must be of even length; variables of type QUEUE must be variables of monitors only. Pass four consumes declarations, encodes their information, and distributes it wherever needed in the routine bodies. At the completion of the pass the intermediate code is merely a sequence of routine bodies. The number of routines in the program is saved in the PASSLINK.

Analysis of routine bodies is performed by pass five and consists of ensuring the compatability of operands with each other, and of operands with operators. For example, only an integer may be added to another integer, and the addition operator must be an integer ADD, not real. The pass also generates addressing commands.

4.2 PASS SIX

The final two passes constitute a two-pass assembler. The first of these, MCPASS6, selects the final code, converts routine and jump labels to addresses, determines the stack space required for each routine and system component, and constructs the constant pool.

The main portion of the pass is a loop around a CASE statement, where the case labels are the operators of the input language. The loop consists of reading an operator (variable OP, in the MCPASS6 source code) and using it as the case selector value to perform the actions appropriate to that operator. If the operator is one which has operands in the code stream, they are read by one of the READxARG routines, where "x" is the number of operands (1-5) to be read. The global variables ARG1, ARG2, ARG3, ARG4, and ARG5 contain the operands after the read operation. The loop terminates when the EOM operator is encountered.

Some inputs translate on a simple one-for-one basis to the output language. For example, whenever PUSHCONST1 is the input operator its operand (the value to be pushed onto the stack at run time) is read into ARG1, and the output PUSHCONST2, followed by ARG1, is always emitted. Other translations are a bit more complex. For example, the intermediate code generated from PUSHVAR1 and its three operands (variable type, addressing mode, and displacement) depends on whether the variable to be pushed is of word type, whether it is a variable in a system component, and whether the displacement is positive. Based on the operand values, one or more instructions will be generated, and the displacement value may or may not be adjusted. Detailed descriptions of the translations are presented in sections 5.2.3 and 5.2.4.

In the pass six input language, the operands of jump instructions (JUMP1, FALSEJUMP1, and CASEJUMP1) are label numbers which have been generated during syntax analysis. A jump's destination is marked by a DEFLABEL1 instruction whose operand is a label number which is the same as that of the jump operand. Since jumps in the final code are in terms of displacements relative to the jump instructions themselves, the labels must be converted to displacements within the virtual code address space. As the first step in the conversion, this pass builds a table (JUMPTABLE) in the heap which will be used by the next pass. Label numbers are used as the table index to insert table entries which are object code location counter values. The global variable LOCATION serves as the location counter of the final code to be produced by pass seven, and contains the address of the next instruction to be generated. When a DEFLABEL1(<label>) instruction appears in the code stream, the current value of LOCATION is placed in the label-th position of JUMPTABLE. Label numbers are unique so there is no danger of overwriting previous entries. If the LINENUMBER compiler option is in effect a NEWLINE2 instruction is generated, otherwise DEFLABEL1 produces no code. When a jump instruction appears, a corresponding output instruction is generated and the location counter updated. The current value of LOCATION is then emitted, but LOCATION is not updated since the emitted value will be removed from the code stream by pass seven. Pass seven performs the rest of the label-to-displacement conversion.

Routine label numbers are converted to virtual addresses in a similar manner using BLOCKTABLE. CALL1 instructions correspond to jump instructions and cause the emission of the current LOCATION value as well as the output language instruction. Pass seven will use and remove the LOCATION value placed in the code stream. ENTER1 is analogous to DEFLABEL1 and results in the insertion of the LOCATION value into the BLOCKTABLE entry indexed by the routine label. Final conversion to displacements is done by the next pass.

Pass six computes the maximum amount of run-time stack space required by each routine. This is possible since Concurrent Pascal does not allow recursive routine calls. The space requirement is the sum of the length of the routine's local variable area, the maximum size of the operand stack, the markstack size, and any additional reserved space (in the case of a process which is host to a recursive sequential program). The local variable length and size of additional reserved space are taken from the input code stream as operands to ENTER1 and are kept in the global variables VARLENGTH and STACKLENGTH, respectively. The markstack size is always five words. The maximum amount of space needed for the operand stack is determined as the pass scans the intermediate code. For each routine the variable TEMP simulates the rise and fall of the run-time operand stack. At the start of the routine (ENTER1 instruction) TEMP is set to zero. As the pass generates instructions which will push or consume stack operands at run time, TEMP is incremented or decremented by the length of the object pushed or consumed. MAXTEMP is also set to zero at the start of each routine. Each time TEMP is <u>increased</u> its new value is compared to the current value of MAXTEMP. If TEMP is greater, its value is placed in MAXTEMP, thus recording the "high-water" mark of the operand stack up to that point in the routine. The operand stack space is also affected by routine calls. When a routine call instruction (CALL1) is encountered, the stack requirement of the called routine (obtained from STACKTABLE-- described below) is added to TEMP to simulate the space it uses during its activation. The run-time return from the called routine, and resultant release of stack space is simulated by immediately decrementing TEMP. When the compiler encounters the end of the calling routine (RETURN1 instruction) its total stack requirement is entered into a table (STACKTABLE), the index of which is the routine label, making this routine's stack requirement available to subsequent routines which might call it. Since Concurrent Pascal allows calls only to routines which have been defined earlier in the source code (and a previous pass enforces this restriction) the stack requirement values will always be in the table when needed at CALL instructions. Pass seven also uses STACKTABLE.

Long constants (compiler-generated null set, real. and string constants) appear in the input code as a CONSTANT1 operator, followed by the byte length of the constant, followed by the constant itself. The constant pool at this point is in the form of a table (CONSTTABLE), the entries of which contain one word of some constant. The global variable CONSTANTS counts the number of words of constants currently in the pool and is the index into CONSTTABLE. When a constant is found, CONSTANTS is incremented by one for every word of the constant which is read from the code stream and placed in CONSTTABLE. No output is emitted.

Logically JUMPTABLE, BLOCKTABLE, STACKTABLE, and CONSTTABLE are simple arrays of integers, but physically they appear as shown in figure 24. The LABELS, BLOCKS, and CONSTANTS values from the PASSLINK record indicate the number of entries required for each table. The tables are allocated space from the heap in pieces consisting of 100 table entries and space for a pointer to the next piece. So, for example. if the value in LABELS is 175, JUMPTABLE will consist of 200

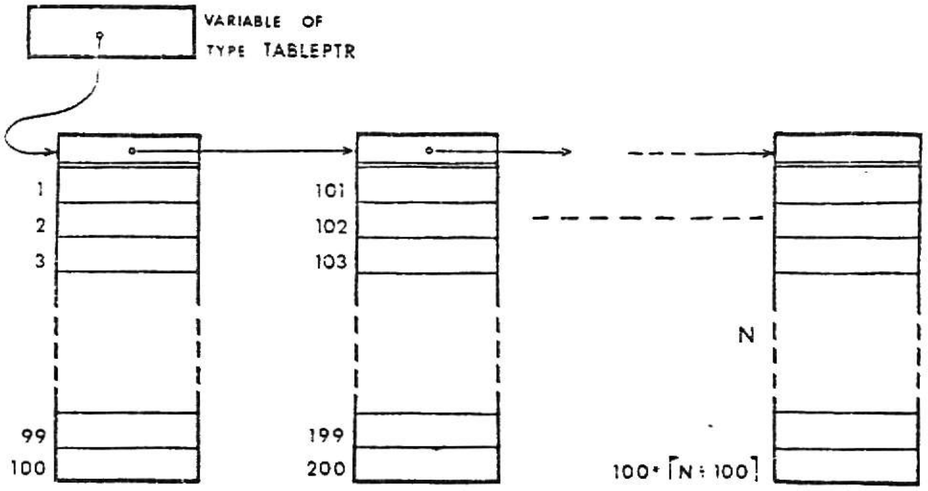


FIGURE 24. Heap table layout. N is the number of entries actually needed.

table entries. In general. tables have $100*\lceil N/100 \rceil$ entries allocated, where N is the number actually needed. The tables are known by their pointer (of type TABLEPTR) to the first piece, such as the JUMPTABLE variable.

Three routines manage the heap tables. ALLOCATE gets the required space from the heap, sets the pointers between table pieces, and returns to the caller a pointer to the first table piece. ENTER uses as input a pointer to a table's first piece, an index, and an integer value. It places the integer in the table named by the pointer, in the position specified by the index. The *logical* equivalent is TABLENAME[INDEX]:=VALUE. Table entries are retrieved by the ENTR function which uses a table pointer and an index as input. It is *logically* equivalent to ENTR:=TABLENAME[INDEX]. All four tables built by MCPASS6 are left in the heap for MCPASS7 to use. They are passed by putting the pointer to each table's first piece in the PASSLINK record. The address of the PASSLINK itself is sent to the next pass as PARAM[2].PTR. a program parameter.

The source listing of MCPASS6 contains several sections which require some comment. In the constant definitions, "VIRTUAL DATA TYPES" refers not to Pascal TYPEs but to the length attribute of an object-- objects have the same length as a byte, word, real value, or set variable. The virtual addressing constants have the following meanings:

MODE0=0; Constant-- the object is to be addressed relative to the start of the constant pool;

MODE1=1; Procedure-- the object is a parameter or local variable of a non-entry routine;

MODE2=2; Program-- the object is a parameter or variable of a sequential program;

MODE3=3; Process entry-- the object is a parameter or local variable of an entry routine in a process type; that is, of an interface routine;

MODE4=4; Class entry-- the object is a parameter or local variable of an entry routine in a class type;

MODE5=5; Monitor entry-- the object is a parameter or local variable of an entry routine in a monitor type;

MODE6=6; Process-- the object is a parameter or permanent (global) variable of a process type;

MODE7=7; Class-- the object is a parameter or permanent (global) variable of a class type;

MODE8=8; Monitor-- the object is a parameter or permanent (global, shared) variable in a monitor type;

MODE9=9; Standard-- standard routine;

MODE10=10; Undefined-- used for error recovery;

MODE11=12; Manager-- not relevant to project;

MODE12=13; Manager entry-- not relevant to project.

Notice that the values of MODE11 and MODE12 do _not_ correspond to their names.

The section of the source code marked "COMMON TEST OUTPUT MECHANISM" contains the routines which produce an unformatted listing of the intermediate output code, perform simple pass initialization and termination functions, and handle page-buffer I/O with the intermediate code files. The "INPUT PROCEDURES" routines read from the code stream the number of instruction operands specified in the routine's name, as mentioned above. The WRITEx routines put into the output code stream the number of integers specified in the routine name, where the first one is an operator and the others are operands. WRITEARG emits just one instruction operand (no operator). Both WRITEx and WRITEARG increment the location counter, LOCATION. Routines in "STACK PROCEDURES" are used to simulate the growth and reduction of the run-time stack. "BLOCK PROCEDURES" are called only when the compiler starts scanning a new routine or finishes the current one (that is, when ENTER1 and RETURN1 instructions, respectively, are encountered). The heap table management routines have been described above. BEGINPASS and ENDPASS take care of initialization and termination functions which are peculiar to pass six, such as allocating heap tables, starting the location counter at zero, and saving data in the PASSLINK record. In ENDPASS, PROGLENGTH is the length of the entire object file (header words, virtual code, and constants); CODELENGTH is the length of the virtual code proper; STACKLENGTH is the stack requirement for the anonymous initial process, hence the entire program; and VARLENGTH is the amount of variable space required by the initial process (that is, the length of the permanent variable space). The rest of MCPASS6 will be discussed in section 5.2.

4.3 PASS SEVEN

MCPASS7 is the code assembly phase of the compiler and is structurally much simpler than the other passes. In general terms, it performs the following functions:

1) Put the five words of header information into the object file;
2) Convert intermediate code operators to the even-number integer encoding which is intelligible to the interpreter;
3) Finish converting jump labels and routine labels to

displacements;

4) Insert stack requirement values into routine entry and process initialization instructions;

5) Remove error messages from the code stream and put them at the end of the source listing;

6) Put long constants into the object file immediately after the virtual code.

All five values for the header words are available from the PASSLINK. The translation of intermediate code operators is done on a simple one-for-one basis and operands are, for the most part, simply copied from the input file to the object file.

To convert jump and routine lables to displacements, the labels in the instructions are used as indices into JUMPTABLE and BLOCKTABLE (which were passed from MCPASS6 through the heap) to retrieve the location counter value associated with the label. The difference between that value and the current location counter value (which was emitted along with the instruction by pass six) is written out as the instruction operand.

The stack space required for each routine is in STACKTABLE. The interpreter needs the information contained there to perform run-time checks for stack overflow when routines are activated. When a routine entry or process initialization instruction is encountered, the routine's label is used as an index into STACKTABLE to fetch its stack requirement value. That value then becomes an instruction operand in the output stream.

If the source program contained any errors they were marked in previous passes by a MESSAGE(PASS, ERROR, LINE) instruction in the intermediate code. Pass seven removes MESSAGE instructions from the code and prints the text for the error which is encoded in the ERROR operand. PASS is the number of the compiler pass which detected the error, and LINE is where it was found in the source code.

Once the virtual code proper has been generated, the constant pool is built by copying the contents of CONSTTABLE to the output file.

COMPILER MODIFICATIONS

Structurally, the MEPASCAL (MicroEngine PASCAL) compiler system of programs is directly analogous to the MCPASCAL system. The following correlation holds between the programs of the two compilers:

	MCPASCAL COMPILER	MEPASCAL COMPILER
Command File:	MCPASCAL.CSS	MEPASCAL.CSS
Driver:	MCPASCAL	MEPASCAL
Compiler Passes:	MCPASSx	MEPASSx
Pass 1, no source:	MCPASS1N	MEPASS1N
Code Mnemonics:	MNEM	MEMNEM.

Each program of MEPASCAL performs the same general function as its analog. In some cases (passes one and two, for example) the code is virtually identical.

Theoretically, passes one through five are independent of the compiler's target machine. Passes six and seven translate the machine-independent output language of pass five to the language of some particular machine. In view of this, it was originally thought that MEPASCAL could be built by rewriting just the last two passes of MCPASCAL so that they would produce Microengine machine code instead of virtual P-code. It seemed that localizing all the modifications would provide several advantages. It would eliminate the need for detailed knowledge about the workings of the other passes, design and programming errors introduced during the modification process would be confined to a small, familiar portion of the compiler, and changing just two passes in a straightforward manner seemed to offer fewer opportunities to make mistakes in the first place. The idea is simple, and it seemed workable. As it turns out, the first five passes can be slightly affected by the requirements of the target language, so they could not be used without change. However, modifications were introduced only when it was felt that they were absolutely necessary, and so very little code was actually changed. Some changes could have been made either in pass six or one of the prior passes. As a rule, the choice was made to change pass six, even though it might have been "optimal" in some sense to do otherwise. This chapter describes the minor changes made to the first five compiler passes, and the major rewrite of pass six.

5.1 CHANGES TO PASSES ONE THROUGH FIVE

MEPASS1 (MEPASS1N) is the same as MCPASS1 (MCPASS1N) except for two changes in the PASSLINK record which is defined in the program prefix. Because of the way the Microengine XJP (case JumP) operator works, the number of words which case jumps will add to the constant pool must be exchanged between passes five and six. The field XJP_OFFSETS will hold that value. Pass five will also create and pass on a record of information concerning the concurrent/sequential program interfaces. INTERFACE will hold a pointer to that record. These two additions were also made to MCPASS2 to yield MEPASS2.

MCPASS3 incorporates the same PASSLINK changes made to the first two passes. The pass also uses a new output operator, DUPTOS2, (DUPlicate Top-Of-Stack word) when handling the input operator INIT_NAME1. The new operator is required because the Microengine code generated after INIT_NAME will, at run-time, pops the top-of-stack word, although that word will be needed later. DUPTOS2 will preserve the word for future use. See section 5.2.4.4 for more information.

The PASSLINK changes described above are included in pass four as well as the new input and output operators DUPTOS1 and DUPTOS2. DUPTOS1 appears as part of the CASE statement in the main loop of the program. MEPASS4 also contains a modification to the BODY procedure. Before the change, the output generated in response to the BODY1 input operator varied, depending on whether the body (routine) being entered by the compiler was the initial routine of a system component. If it was _not_ an initial routine, the output was the BODY2 operator and five operands, including the length of the routine's local variable area and the amount of stack space occupied by its parameters. For initial routines, the output was the same, except that the variable area and parameter length operands were always zero. Space for variables and parameters had already been allocated as a record in the data space of the anonymous initial process. For the Microengine, the two sizes need to be known in pass six for proper address displacement calculation. Since this is needed regardless of whether the routine is an initial one or not, BODY was changed so that the actual sizes are always included as BODY2 operands. More information is included in sections 5.2.4.1, 5.2.4.5, and 5.2.4.6.

Compared to the previous passes, MEPASS5 incorporates a large number of code modifications, including the new operators DUPTOS1 and DUPTOS2. In the other passes PASSLINK.INTERFACE is of type POINTER merely to reserve a fullword (32 bits) of storage, the amount of space required for _any_ pointer. In pass five PASSLINK.INTERFACE is actually _used_, and so it must be declared as a pointer to some specific object type. Since it will point to an IFINFO (InterFace INFOrmation) record, it is declared to be of type IFPTR (InterFace PoinTeR). The IFINFO structure records the number of interfaces contained in the concurrent program (field INTERFACES) and the number of accessible process ENTRY routines in each one (field INTERFACESIZES). INTFACEPTR and INTFACE are instances of IFPTR and IFINFO, respectively. Figure 27 shows how the objects and pointers of the PASSLINK relate to each other.

The instance of PASSLINK in pass five is the target of the pointer INTER_PASS_PTR. The procedure INITIALIZE handles initialization functions which are peculiar to this pass, before any intermediate code is read. In that procedure, INTER_PASS_PTR@.XJP_OFFSETS is set to zero to indicate that no case jumps (hence, no case jump offsets) have yet been found in the code. Space for the interface information record (type IFINFO) is allocated from the heap. Since no interfaces have been seen, the INTERFACES field in that record is initialized to zero. As the pass executes, it uses some space in the heap for temporary workspace. When the pass is finished, the workspace can be returned to the heap, but the interface information record must be retained for use by pass six. After allocating space for the IFINFO record, the extent of the heap is MARKed into INTER_PASS_PTR@.RESETPOINT. Temporary space is allocated beyond that point as needed, and at the end of the pass (in procedure EOM) that workspace (and only that workspace) is RELEASEd, leaving the IFINFO record intact. In preparation for pass six (procedure NEXT_PASS) the pointer to the IFINFO record (INTFACEPTR) is stored into one of the PASSLINK fields, making the record accessible to the next pass. During its execution, MEPASS5 recognizes the existence of an interface for a sequential program when it encounters a PROG_CALL1 operator with a nonzero operand. The operand is the length (in bytes) of the run-time jump table to be built before invoking the sequential program, and is a direct indicator of the number of process ENTRY routines to which the program will have access. The procedure PROG_CALL was modified to count the number of interfaces encountered, and the number of ENTRY routines made accessible in each one. That information is saved in the IFINFO record and left in the heap for pass six. Refer to sections 5.2.4.4 and 5.2.4.8 for additional information.

The procedure CASE_LIST was modified to start the process of converting the virtual Concurrent Pascal CASE statement construct to a form suitable for the Microengine. The modification merely tallies the number of bytes by which the constant pool must be enlarged in order to accommodate the current CASE statement's list of jump offsets. The total amount by which all CASE statements enlarge the constant pool is saved in the PASSLINK record (XJP_OFFSETS) for use by pass six. More information on the Microengine CASE jump operator (XJP) and the way it is handled by MEPASCAL can be found in [MICR79, REGE79] and section 5.2.4.3.

Procedure ADDRESS contains a deletion. On the virtual Pascal machine, when a component variable is INITed, or passed as a parameter (access right) to another system component, its address is pushed onto the stack. That address is then incremented (FIELD operator) by the length of the permanent variables in the component. As a result, the top-of-stack word points to a word in the "middle" of the component variable's record, such that all the parameters (access rights) lie above it, and all the permanent variables below (see figure 25). This address will be loaded into the global base register (G) whenever a process executes the code of the component type. Since the Microengine hardware expects parameters <u>and</u> variables to lie above the global base (see figure 19), the FIELD operation for system components which is generated in the ADDRESS procedure must be dispensed with. Sections 5.2.4.4, 5.2.4.5, 5.2.4.6 also contain discussions related to this topic. At the time of writing it was noticed that a similar line

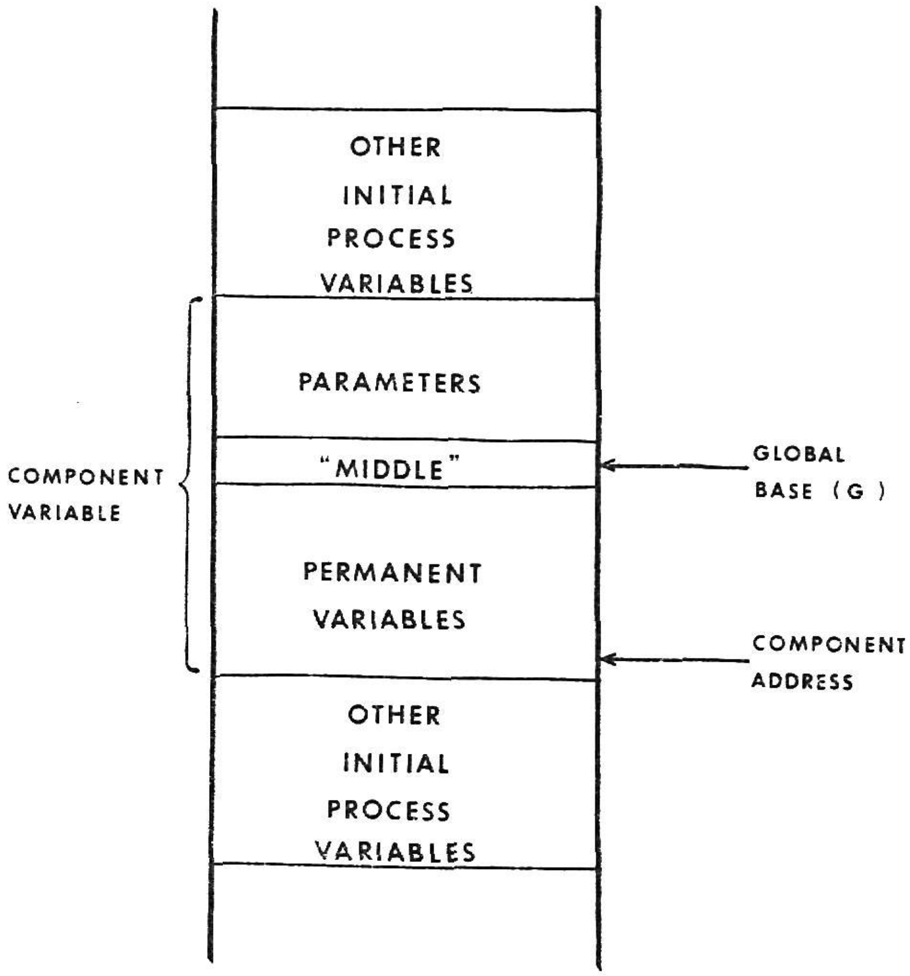


FIGURE 25. Relationship of global base to component variable address in the virtual Pascal machine. Compare with figure 19 which shows the analogous Microengine configuration.

of code appears at the end of the procedure SUB, and may also require deletion.

The last change to pass five involves subscript handling for strings. The virtual Pascal machine tacitly assumes that the underlying real hardware is byte addressable. That means that the address of a byte occupies the same amount of space (one word) as the address of a word. On the Microengine, not all addresses (pointers) have the same format. In fact, there are three different pointer formats [MICR79, REGE78, REGE79], although only two are of importance here; namely, word pointers (which occupy one word of memory) and byte pointers (which occupy two words). The only addresses generated by MEPASCAL which are _not_ word pointers are pointers to elements of arrays of characters (strings). When it produces the address of an array element, pass six must know the virtual data type (BYTETYPE or not) of the pointer's target object. This information is available in MCPASS5 but not MCPASS6, so in MEPASS5 the object type (field KIND) was added as a fourth operand to the INDEX2 operator. MEPASS6 generates the correct address format based on the value of that operand. Section 5.2.4.2 contains a description of how that is accomplished.

All of the code changes made in the first five passes are shown in Appendix D.

5.2 CHANGES TO PASS SIX

Pass six contains more modifications than all of the previous passes combined. Not only were the changes numerous, but some were quite extensive. Besides changing the instructions to be generated, the format of the pass and its output structures were also changed.

5.2.1 Pass Output

As shown in figure 26, MEPASS6 takes its input from the heap and intermediate code file, and produces additional heap information and up to two intermediate code files.

The structures which pass six leaves in the heap are shown in figure 27. Space for the PASSLINK and IFINFO records was allocated in passes one and five, respectively. The fields in the PASSLINK record, which are not pointers, have the following meanings:

OPTIONS-- the set of compiler options in effect, as determined by pass one

LABELS-- the number of jump labels generated in pass two

BLOCKS-- the number of routine labels generated in pass four

CONSTANTS-- the calculated amount of constant pool space (in bytes) needed for long constants, as determined by pass three

XJP_OFFSETS-- the calculated amount of consant pool space (in bytes) needed to accommodate CASE jump offsets, as determined by pass five.

The meanings of the pointer fields is obvious from figure 27. In general terms, the IFINFO record contains information regarding the (possibly null) interface segment(s) in temporary file number four (figure 26). In particular, the INTERFACES field contains the number of interfaces found in the intermediate code, thus the number of interface segments generated. INTERFACESIZES is an array which contains the number of accessible process ENTRY routines for each interface. The interfaces are implicitly numbered in the order of their appearance in the intermediate code, and those numbers are used for the array index. In the TABLEPART record, SEGDISTANCE contains the

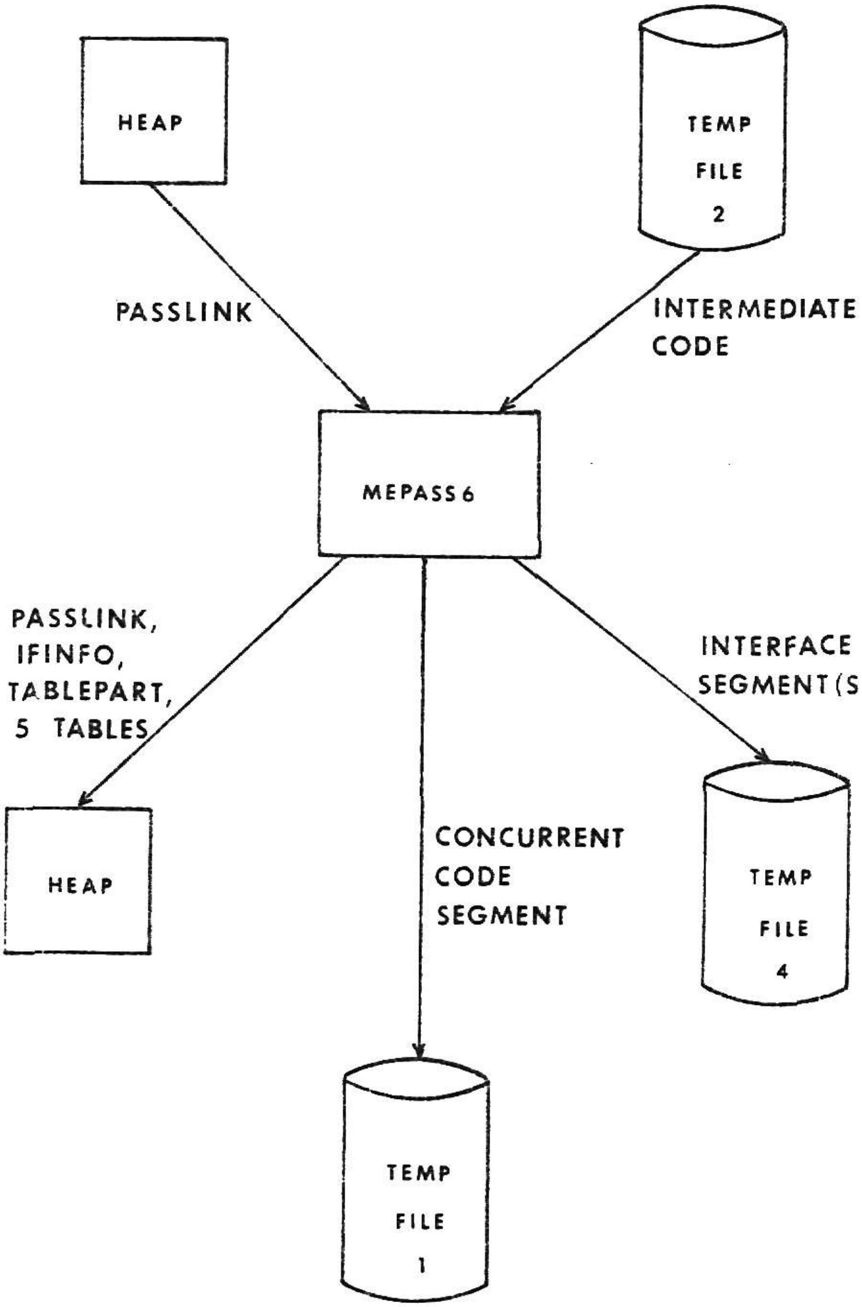


FIGURE 26. Input and output data for MEPASS6.

* The number of interfaces can be zero, leaving TEMP FILE 4 empty.

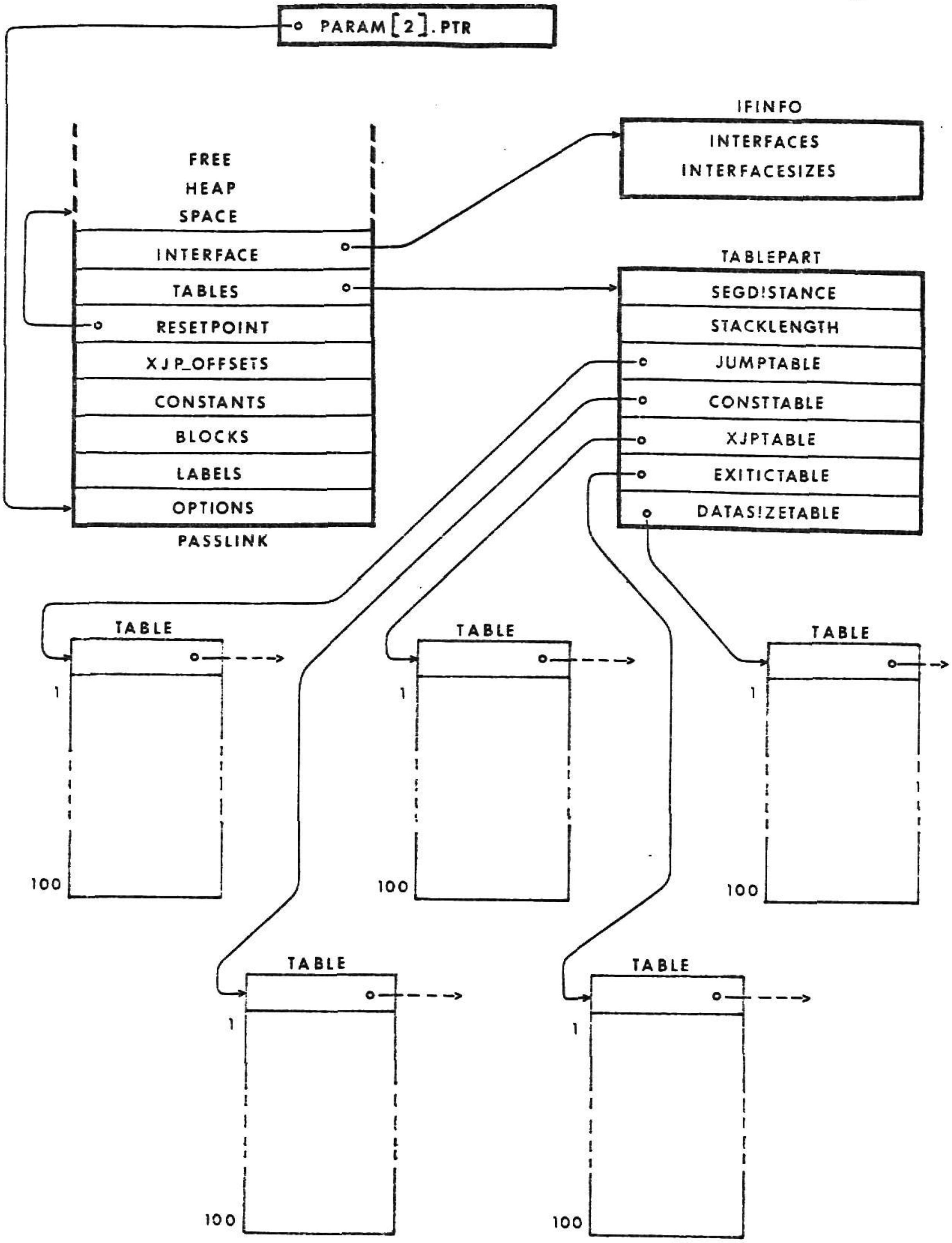


FIGURE 27. Structures left in the heap by MEPASS6. Althcugh they are shown separately here, IFINFO, TABLEPART, and all of the TABLEs actually reside in the area marked "FREE HEAP SPACE". RESET points to the word just above IFINFO, since its value remains unchanged from pass five. TABLE layout is shown in detail in figure 24.

byte-measure equivalent of the header word in the final concurrent code segment. See section 3.2.2 for a description of the header word. STACKLENGTH is the stack requirement for the concurrent program. It is a leftover from MCPASS6 and could have been eliminated. The meaning of the other fields is obvious from figure 27.

MEPASS6 leaves five tables in the heap: JUMPTABLE, CONSTTABLE, XJPTABLE, EXITICTABLE, and DATASIZETABLE. The first two tables are used in exactly the same way as in MCPASS6 (section 4.2). XJPTABLE contains case jump offsets. In the Pascal virtual machine these appear in the object code proper, but the Microengine requires them to be in the constant pool. Pass six removes the offsets from the code stream and places them in XJPTABLE in the same manner that constants are put into CONSTTABLE. Pass seven will combine the contents of the two tables to form a single constant pool for the concurrent segment. EXITICTABLE uses as many entries as there are routines in the concurrent segment. Routine numbers are used to index into the table, and the entries are the final-code EXIT-IC values for the routines. The meaning of the EXIT-IC field in Microengine code files is given in section 3.2.2. Routine numbers are also used as the index into DATASIZETABLE, and the entries are the DATASIZE values for the routines (see section 3.2.2).

In the conversion of MCPASS6 to MEPASS6 heap objects were added, deleted, and retained. An explanation of the reason each object was added or deleted will be helpful. The MCPASCAL compiler calculated PROGLENGTH, CODELENGTH, STACKLENGTH, and VARLENGTH in pass six so that pass seven could use those values as part of the five-word header for the object code file (see figure 3). Microengine object files (figures 6 and 7) do not require that header, so those fields could be deleted from TABLEPART. (STACKLENGTH still remains through oversight.) In the same record, BLOCKTABLE was used to convert routine labels to the virtual addresses required by final-code CALL instructions. Microengine routine-invocation instructions use the routine labels themselves as operands, so the table was removed since the conversion became superfluous. The routine stack requirement values held in STACKTABLE are needed during MCPASS6 to help calculate the stack requirements of routines which call other routines (see section 4.2). MCPASS7 retrieves those values and inserts them into the code stream as operands to routine ENTER operators. Since Microengine object code does not require that information, STACKTABLE is not left in the heap by MEPASS6 and, as a consequence, does not appear in TABLEPART. It is used <u>during</u> pass six, as in MCPASS6, and then discarded.

The additional heap objects are INTERFACE and XJP_OFFSETS in PASSLINK, IFINFO, SEGDISTANCE in TABLEPART, XJPTABLE, EXITICTABLE, and DATASIZETABLE. Since all interface segments have the same structure (section 3.4.1) which is known to the compiler (section 5.2.4.8), the number of process ENTRY routines in an interface completely defines it. The number of interfaces and the size of each one is determined by MEPASS5 and left in the heap in the IFINFO record which is the target of the pointer INTERFACE in PASSLINK. MEPASS6 generates interface segments which are almost in final object-code form; <u>much</u> closer to that form than the concurrent segment. IFINFO is left in the heap for MEPASS7 so that when it scans the file of interface segments it will know the layout of each segment. Otherwise, there would be no way to determine, for example, the end of the constant pool and the location of the EXIT-IC field without backtracking in the file.

XJP_OFFSETS was calculated in MEPASS5 (section 5.1) and its function is analogous to that of CONSTANTS-- it tells MEPASS6 the number of table entries needed in XJPTABLE to accommodate case jump offsets which will be removed from the code. XJPTABLE is, essentially, an extension of CONSTTABLE, but case jump offsets cannot be intermixed with the long constants. The separation is required because an earlier pass calculated constant-mode displacements under the virtual-Pascal-machine assumption that case offsets would reside in the code portion of the object file. Willy-nilly inclusion of the offsets in the constant pool would invalidate those displacements in all but the most extraordinary circumstances. Pass seven will combine the contents of the two tables so that in the final code long constants will reside in the constant pool <u>ahead of</u> all the case offsets. This ensures that constant-mode displacements determined previously will still be valid, and yet allows pass six to easily compute the operand to the Microengine XJP (case jump) operator (see [MICR79] and section 5.2.4.3).

EXITICTABLE is passed from MEPASS6 to MEPASS7 since pass six cannot possibly know a routine's exit address before it has even started scanning its code. This situation arises because, as figure 7 indicates, the EXIT-IC field precedes the code of its routine in the object file. In MEPASCAL, pass six determines the EXIT-IC value as the routine is scanned, and pass seven places it in the proper position.

Although there is no usage-before-availability problem with object-code DATASIZE values (they are available as the VARLENGTH operand in ENTER1 instructions), emitting them directly into the code stream is undesirable. Pass seven is incapable of handling the unpredictable appearance of values which are neither operators nor operands, without extensive modification. It can, however, easily pull the values out of the heap table and put them in place when the final code file is built.

MEPASS6 always generates a file (temporary file 2) of Microengine machine code (with a sprinkling of virtual operators) for the concurrent segment. Physically, it is a sequence of 16-bit integers, and consists of only the code for the routines which make up the concurrent program. The constant pool, procedure dictionary, and other non-code items do not appear in the file, although pass six takes into account their existence in the final object code. Since the hardware requires routines (more specifically, the EXIT-IC fields of routines) to begin on word boundaries, NOP (No OPeration) instructions may appear between adjacent routines for alignment purposes. Routines always end with an RPU instruction. There are three virtual (non-Microengine) instructions which can appear in the file in order to communicate information used by other parts of the compiler. MESSAGE_2 passes encoded error message data so that character error messages can be printed by pass seven. EOM_2 tells pass seven when it has reached the end of the file. NEWLIN_2 is a crutch used by MEMNEM to determine when the machine code corresponding to a new line of source code has been reached (Appendix A). MEPASS6 also inserts location counter values into the code stream (just as MCPASS6 does-- see section 4.2) as part of the mechanism for resolving jump displacements. All four of these items, which are extraneous to the final object code, will be removed in the next pass.

MEPASS6 generates an interface segment for every interface it finds in the intermediate code. All of the segments are placed end-to-end in a temporary file (file number four) as a sequence of 16-bit integers. Pass seven will concatenate the interface segments to the concurrent code segment, pack operators and operands into words, and provide sufficient padding to ensure that each one begins on a disk block boundary, thereby constructing a single, complete object code file ready for execution on the Microengine. Section 5.2.4.8 contains an in-depth discussion of the mechanism in pass six which generates the segments.

5.2.2 Pass Structure

The structure of MEPASS6 is similar to that of MCPASS6 (section 4.2), although there are some differences. As mentioned in the preceding section, TABLEPART and PASSLINK were changed, and IFINFO was added to the package of objects to be left in the heap for pass seven. The input operators are the same except that DUPTOS1 has been added. The output operators for the Pascal virtual machine have all been replaced by Microengine operation codes. All Microengine codes are included, even though some will not be used, for the sake of completeness. Since they are defined in the CONST section of the program, the unreferenced opcodes do not add to the size of the program's object code. Three non-Microengine output operators (described in the preceding section) are also defined for handling error messages, marking the end of the concurrent code segment, and marking the start of each source line.

Several changes were made in the "COMMON TEST OUTPUT MECHANISM" portion of the pass because of the interface mechanism. Three variables (IFPAGE_OUT, IFPAGES_OUT, IFWORDS_OUT) were added to handle page-buffer output to the interface file. That output mechanism is exactly like the one used for intermediate concurrent code output, except for the file designation. The file identifiers are OUTFILE (value=1) for the concurrent code file, and INTERFACEFILE (value=4) for the interface segment file. File 3 is the final object code file to be generated by pass seven. File initialization (INIT_PASS routine) and termination (NEXT_PASS routine) functions for all files used by pass six are performed at the same time in identical fashion.

Procedure WRITE IFL has been expanded so that output can be easily directed to either the concurrent file or interface file as necessary. When the pass starts (procedure BEGINPASS), a switch (GENNINGINTFAC) is set FALSE to indicate that an interface segment is not currently being generated (concurrent code is being generated) and that intermediate code output should be directed to the concurent file. When an interface is encountered, the switch is set TRUE (in procedure GEN_INTERFACE) to indicate the opposite state of affairs. After the interface segment has been completely generated, the switch is returned to the FALSE setting. All output to files occurs through procedure WRITE_IFL, and on every invocation it tests GENNINGINTFAC to determine the file to which the output should be directed. Duplicate sections of code handle buffer management and output to each file. The duplicate code is a violation of good programming practice, and could be easily eliminated by the use of array variables. However, at the time this mechanism was designed, the duplicate code arrangement was an easier concept to deal with, and the amount of overhead is not alarming.

The "OUTPUT PROCEDURES" section of code contains a number of modifications which are due to two causes. First, unlike the Concurrent Pascal virtual machine. Microengine operators use operands of various lengths. In fact, for some instructions, a single operand can have either of two lengths, depending on its value. The

Microengine instruction set is described in appendix B.5 of [MICR79]. Heterogeneous lengths do not affect the intermediate code produced by MEPASS6 since it generates operators and operands which are all 16-bit integers, but updating the final-code location counter becomes more complicated. The new mechanism uses a new global type (TYPEOFCODE), the values of which have the following meanings:

OPTR-- The output data item is a Microengine instruction <u>OPeraToR</u>. Length is always one byte;

UB-- An <u>Unsigned Byte</u> operand. Length is always one byte;

SB-- A <u>Signed Byte</u> operand. Length is always one byte;

DB-- A "<u>Don't</u> care" <u>Byte</u> operand. Length is always one byte;

B-- A "<u>Big</u>" operand. Length is one or two bytes, depending on the value;

W-- A <u>Word</u> operand. Length is always two bytes;

NOTME--The output data item is <u>NOT</u> a <u>MicroEngine</u> code item and will be removed by pass seven, so length is zero.

The new procedure UPDLOC (UPDate LOCation counter) ensures that the location counter (variable LOCATION) gets incremented correctly for each type of intermediate code item. Every time a WRITEx or the WRITEARG routine emits an item of output, a call is made to UPDLOC. The numerical value of the item and its TYPEOFCODE value are provided as parameters so that UPDLOC can determine the item's size, and increment LOCATION accordingly. The WRITEx and WRITEARG parameter lists were modified to accept operand TYPEOFCODE values so they can be passed on to UPDLOC. Notice that the NOTME value provides a way to use the standard output procedures for generating intermediate code instructions which will not appear in the final code, without also causing an increment of the location counter.

The second cause for modification concerns the generation of interface segments. For the sake of consistency, the interface-generation routine emits code by invoking the same output procedures as the rest of the program. Without modification of the procedures, generation of an _interface_ would cause the _concurrent_ segment's location counter to be incremented. This is an undesired side effect since locations within a segment must be relative to the beginning of _that_ segment. The remedy for that side effect is to test the GENNINGINTFAC switch before calling UPDLOC. If the item just emitted was sent to the interface file the switch will be TRUE, and LOCATION will not be affected. Otherwise, it will be updated as described above.

The function DISPL (DISPLacement) is designed to return the displacement of a variable from some base address. It had to be entirely rewritten because of addressing differences between the two target machines. In the virtual Pascal machine, displacements are positive (for routine parameters) or negative (for local variables) even-byte displacements from a data-space base register (either G or B-- see figure 28). Constants are addressed relative to the starting address of the constant pool which is calculated when the concurrent program starts execution. The Microengine addresses both parameters and variables with positive word offsets from registers B or MP, and constants are addressed by word offset from the start of the segment. The virtual machine can refer to the fields of the markstack by addressing the words with offsets 0, 2, 4, 6, and 8 from, for example, the local base. The first variable word after the markstack has displacement 10, and the first parameter has displacement -2. In contrast, the Microengine hardware does _not_ allow access to the

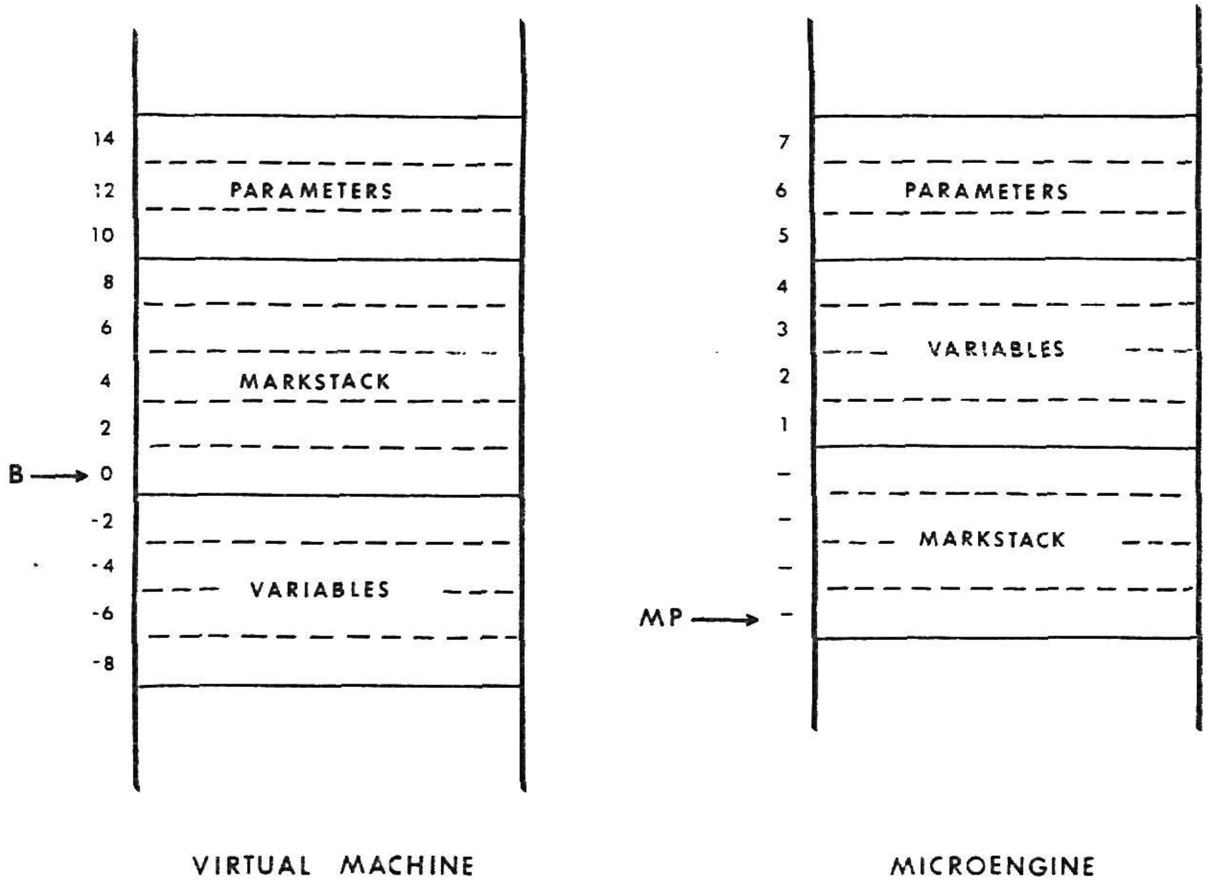


FIGURE 28. Variable and parameter addressing mechanisms of the two target machines. The local stack areas are shown for a routine which has three words of parameters and four words of local variables. Addressing displacements appear to the left of each stack. In general, the Microengine markstack fields are not accessible by using the displacement mechanism.

markstack, except by the LSL (Load Static Link) instruction [MICR79]. The hardware automatically compensates for the markstack which is between the base address and the first variable, so that the first variable word has offset 1. The offset of the first parameter depends on the number of variables, since the variables separate the parameters from the base address. For example, if a routine has four words of local variables and three words of parameters, the parameter words have offsets 5, 6, and 7. The MEPASS6 version of DISPL calculates displacements in the following way. The constant block will immediately follow the segment header word in the object file, so a constant's displacement is its word displacement within the constant block plus one word. In "procedure", "program", and "entry" addressing modes (section 4.2) the displacement calculation takes into account the following factors:

1) The displacements for variables must be positive, not negative;
2) Parameters must be addressed as if they are an extension of the variables area;
3) The first two words of variables are reserved for the source line number and old global base register values, since the Microengine has no markstack fields for them specifically;
4) The Microengine uses word, not byte, displacements.

In "process", "class", and "monitor" modes, the calculation is the same, except that space for the two reserved words is of no concern since no space for them exists in the component's permanent variable record.

The displacement calculation for a "process entry" reference is no more complex than the others, but the run-time structure of the concurrent/sequential interface, on which it is based, is somewhat complicated. Figure 29 shows the arrangement of the stack after a sequential program has invoked a process ENTRY routine and it has begun executing. That stack state was reached in the following way. The sequential program code pushed enough "blank" space onto its operand stack to hold the value which the ENTRY function will eventually return. It also pushed the parameters which the ENTRY routine requires, followed by the index of the prefix routine which the program sought to invoke. The index is itself a parameter, and it is critical that it always be pushed last onto the stack. The program then executed an external routine call instruction, causing the interface markstack to be constructed by the hardware. The interface routine pushed its local variable with offset 1 (the prefix index parameter) onto the stack for use as the selector value for its CASE statement (see figure 17). Execution of the case jump (XJP instruction) consumed the top-of-stack operand, and execution of the case code invoked the process ENTRY routine, causing the allocation of its local variable space on the stack and the construction of its markstack. As in other routines, local words one and two are reserved for the source line number and old global base. The displacement calculation for process ENTRY routine <u>variables</u> is the same as for class and monitor ENTRY routines. <u>Parameters</u>, however, are different since, as the reader can see in figure 29, the interface markstack sits between the local variables and parameters. Conceivably, there should never be anything <u>but</u> the stack marker in that position, and parameter displacements could be calculated as

(BYTE_OFFSET + TWOWORDS + FOURWORDS + ONEWORD) DIV WORDLENGTH

where TWOWORDS accounts for the two reserved local words, FOURWORDS is

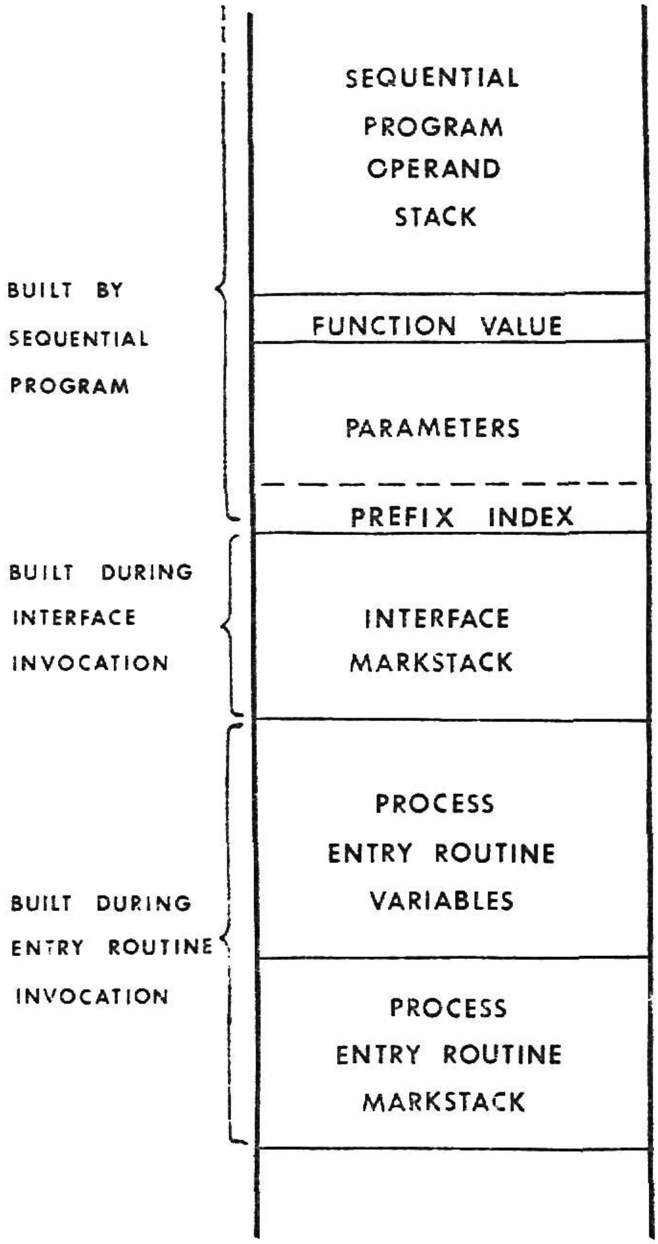

SEQUENTIAL PROGRAM OPERAND STACK
BUILT BY SEQUENTIAL PROGRAM
FUNCTION VALUE
PARAMETERS
PREFIX INDEX
BUILT DURING INTERFACE INVOCATION
INTERFACE MARKSTACK
PROCESS ENTRY ROUTINE VARIABLES
BUILT DURING ENTRY ROUTINE INVOCATION
PROCESS ENTRY ROUTINE MARKSTACK

FIGURE 29. Configuration of the run-time stack after the execution of a process ENTRY routine (a function, in this case) has begun. The interface routine has no local variables.

the size of the interface markstack, and ONEWORD accounts for the word occupied by the prefix index which is an "extra" parameter. That calculation was not used, however, since it seems to depend too heavily on a favorable arrangement of the stack. Also, the machine instruction set includes a number of operators which can access the data spaces of any markstack in the chain of static links. When one of these "intermediate" instructions is used, variables are addressed relative to the markstack to which they are normally considered local. The interface routine invokes the process ENTRY routine with a CXL instruction which makes the static link field in the ENTRY routine's markstack point to the interface's markstack. By directing the "intermediate" addressing operators to traverse just one static link (operand value of 1) the interface parameters can be accessed as if they are part of the ENTRY routine's local variable space. The significance of all this is that the displacement calculation has to only take into account the single word occupied by the prefix index. Section 5.2.4.8 contains more information on the operation of the interface mechanism.

The procedures BEGINPASS and ENDPASS carry out initialization and termination functions specific to this pass of the compiler. As part of the pass setup, the sizes of all the interfaces are copied from the PASSLINK record to an array which is easier to access. Heap space is allocated for the five tables which will be left for the next pass. The current extent of the heap is MARKed, and then a temporary table (STACKTABLE) is given space. Indices into the constant and case jump offset tables are initialized to indicate empty tables. The total size of the constant pool is calculated as the sum of the constants and case jump table sizes since they will be combined in the object code file. The location counter is initialized so that it "points" to the first byte of code proper to be generated. That byte follows the segment header word and the constant pool. ENDPASS RELEASEs the heap space which was used by STACKTABLE since that table will not be used by pass seven. Heap space is then allocated for a TABLEPART record and the table-identifying pointers are stored into it. The address of the last meaningful word in the concurrent segment can be calculated from the current location counter value and the number of routines in the segment. At the end of the pass, LOCATION is the segment-relative address of the word after the last *instruction* in the segment. As figure 7 shows, only the procedure dictionary comes between the last routine (routine one) and the final word. There are as many entries in the procedure dictionary as there are routines in the segment, and each entry takes one word (two bytes), so the segment-relative address of the last word is

LOCATION + (2 * ROUTINES).

Four other routines were added to pass six. PICK_PUSHCONST chooses one of the three Microengine instructions which push integer constants onto the stack, given the value of the constant to be pushed. The difference between the instructions is their length, and it seemed worthwhile to make the small effort required to make optimal use of them to help reduce the size of the object code. GEN_SAVELINE_SAVEGBASE generates the prologue for routines. It generates code which, at run time, will save the source code line number and the contents of the global base register in the local variable space of the new routine. In every routine local word one is reserved for the line number, and local word two is for the old global base register. GEN_EXIT generates epilogue code which is the same for several routine modes-- restoration of the old global base register and the actual return to the calling routine. GEN_INTERFACE generates interface segments, and is presented in detail in section 5.2.4.8.

Six procedures were deleted and rewritten as in-line code. Each of them was relatively small and was called from only one point in the program, so in the interest of comprehensibility they were moved to the places where they more logically belong. ENTERBLOCK and EXITBLOCK were moved into the ENTER1 and RETURN1 (respectively) cases of the SCAN routine. The comparison procedures COMPAREWORD, COMPAREREAL, and COMPARESET were rewritten in the COMPARE1 case of SCAN, and COMPARESTRUCT was inserted into the COMPSTRUC1 case.

5.2.3 Straightforward Instruction Conversions

Simply stated, the major concern of this project is to change the representation of Concurrent Pascal programs without changing their semantics. In order to do that, each virtual instruction which can be output by MCPASS7 was analyzed to determine what changes it causes in the state of the virtual machine. For each one, a sequence of Microengine instructions which would mimic those changes was chosen for generation in place of the virtual instruction. The definitive characterization of virtual machine state changes is the PDP-11/45 interpreter which realizes them. A source listing of the interpreter, written in PDP-11 Assembly code, was available [ZEPK74]. The listing is commented with the same code written in a psuedo-Pascal/assembler "language" which provides a certain (limited) amount of insight into the function of each virtual instruction. Since, at least superficially, both machines have a stack architecture, some instructions could be substituted very simply. Those virtual instructions and their substitutes are presented here.

The instructions which operate on integer (or word-length) and real operands were among the simplest to convert since they perform virtually identical actions on both machines. Figures 30 and 31 list the MCPASS7 output instructions, their actions, and the equivalent Microengine code sequence for word and real operators. Both machines use 16-bit integers, where the low-address byte contains the low-order bits-- PDP-11 format. Real numbers are handled differently on each machine. The virtual machine uses an eight-byte format, whereas the Microengine uses the PDP-11 single precision floating point format which takes only four bytes [DIGI76].

Figure 32 is a list of the instructions for set and structure operations. Sets for the virtual machine have a fixed length of 16 bytes (128 bits). Set members are numbered from 0 to 127, and membership is represented by turning the corresponding bit "on". The Microengine hardware supports sets of variable length up to 255 bytes (4080 bits), but fixed-length sets of 128 elements are used in this project for the sake of simplicity. On the stack, Microengine sets consist of two parts. The top-of-stack word is an integer which is the number of words in the set proper, which is next on the stack. When it is on the stack, the combination of length word and membership bits is referred to as the "set". When they are not on the stack, sets consist of only the membership bits. The Microengine has no hardware instructions specifically for pushing (popping) sets and automatically appending (discarding) the length word. The instructions LDC (LoaD multiword Constant), LDM (LoaD Multiple words), and STM (STore Multiple words) can be used to push and pop the membership bits, but the length word must be handled by additional instructions. The length

VIRTUAL		MICROENGINE
OPERATOR	RESULT OR ACTION	INSTRUCTION(S)
ABSWORD	absolute value of TOS	ABI
ADDWORD	sum of TOS, TOS-1	ADI
ANDWORD	logical AND of TOS, TOS-1	LAND
CONVWORD	real-value equivalent of TOS	FLT
COPYWORD	store TOS indirect through TOS-1 *	STO
DECRWORD	decrement word indirect through TOS-1 *	DUP1 LDM (1) SLDC01 SBI STO
DIVWORD	integer quotient of TOS-1/TOS	DVI
EQWORD	boolean value of (TOS-1 = TOS)	EQUI
GRWORD	boolean value of (TOS-1 > TOS)	LEQI LNOT
INCRWORD	increment word indirect through TOS-1 *	DUP1 LDM (1) SLDC01 ADI STO

FIGURE 30. Concurrent Pascal word (integer) virtual instructions and their Microengine equivalents, where TOS is an integer which is the top-of-stack item and TOS-1 is the integer which was pushed just before TOS. Execution of an instruction causes the stack operands to be popped from the stack. The result (if any) is then pushed onto it. Instructions marked '*' do not push a result.

VIRTUAL OPERATOR	RESULT OR ACTION	MICROENGINE INSTRUCTION(S)
LSWORD	boolean value of (TOS-1 < TOS)	GEQI LNOT
MODWORD	TOS-1 MOD TOS	MODI
MULWORD	product of TOS, TOS-1	MPI
NEGWORD	2's complement of TOS	NGI
NEWORD	boolean value of (TOS-1 <> TOS)	NEQI
NGWORD	boolean value of (TOS-1 <= TOS)	LEQI
NLWORD	boolean value of (TOS-1 >= TOS)	GEQI
ORWORD	logical OR of TOS, TOS-1	LOR
PREDWORD	TOS minus 1	SLDC01 SBI
SUBWORD	TOS-1 minus TOS	SBI
SUCCWORD	TOS plus 1	SLDC01 ADI

FIGURE 30. (continued from previous page)

VIRTUAL OPERATOR	RESULT OR ACTION	MICROENGINE INSTRUCTION(S)
ABSREAL	absolute value of TOS	ABR
ADDREAL	sum of TOS, TOS-1	ADR
COPYREAL	store TOS indirect through TOS-1 *	STM (2)
DIVREAL	quotient of TOS-1/TOS	DVR
EQREAL	boolean of (TOS-1 = TOS)	EQUREAL
GRREAL	boolean of (TOS-1 > TOS)	LEQREAL LNOT
LSREAL	boolean of (TOS-1 < TOS)	GEQREAL LNOT
MULREAL	product of TOS-1, TOS	MPR
NEGREAL	negation of TOS	NGR
NEREAL	boolean of (TOS-1 <> TOS)	EQUREAL LNOT
NGREAL	boolean of (TOS-1 <= TOS)	LEQREAL
NLREAL	boolean of (TOS-1 >= TOS)	GEQREAL
SUBREAL	TOS-1 minus TOS	SBR
TRUNCREAL	TOS, truncated and converted to integer	TNC

FIGURE 31. Concurrent Pascal virtual instructions for real values and their Microengine equivalents. TOS and TOS-1 have the same meanings as in figure 30, but are real values instead of integers. Instruction marked '*' does not push a result onto the stack.

VIRTUAL OPERATOR	RESULT OR ACTION	MICROENGINE INSTRUCTION(S)
	--- SET INSTRUCTIONS ---	
ANDSET	intersection of TOS, TOS-1	INT
BUILDSET	TOS set with a new member; New member number is given by integer TOS.	SLDC00 LDCB (127) CHK DUP1 SRS UNI
COPYSET	store TOS set indirect through TOS-1 *	FJP (0) STM (8)
EQSET	boolean value of set compare (TOS-1 = TOS)	EQUPWR
INSET	boolean value of test for inclusion of TOS-1 member number (integer) in TOS set	INN
NESET	boolean value of set comparison (TOS-1 <> TOS)	EQUPWR LNOT
NGSET	boolean value of subset test (TOS-1 <= TOS)	LEQPWR
NLSET	boolean value of superset test (TOS-1 >= TOS)	GEQPWR
ORSET	union of TOS, TOS-1 sets	UNI
SUBSET	TOS-1 set less members of TOS-1 set	DIF
	--- STRUCTURE INSTRUCTIONS ---	
COPYSTRUC	move TOS@ to TOS-1@ *	MOV (size)
EQSTRUC	boolean value of (TOS-1@ = TOS@)	EQUBYT (size)
GRSTRUC	boolean value of (TOS-1@ > TOS@)	LEQBYT (size) LNOT
LSSTRUC	boolean value of (TOS-1@ < TOS@)	GEQBYT (size) LNOT
NESTRUC	boolean value of (TOS-1@ <> TOS@)	EQUBYT (size) LNOT
NGSTRUC	boolean value of (TOS-1@ <= TOS@)	LEQBYT (size)
NLSTRUC	boolean value of (TOS-1@ >= TOS@)	GEQBYT (size)

FIGURE 32. Set and structure instructions. "TOS@" means "the object to which the top-of-stack word points". Similarly for "TOS-1@". "*" means nothing is left on the stack.

word is discarded during the execution of the ADJ (ADJust set length) instruction which forces the set to a length determined at compile time. However, in this project an FJP (False JumP) to the next instruction is used instead (see figure 32, COPYSET). The lengths of sets never really need to be adjusted since they will be fixed, and the FJP is presumed faster. Notice that structure operators use pointers as stack operands rather than the structures themselves.

The virtual machine has several instructions for process control. They are ATTRIBUTE, CONTINUE, DELAY, EMPTY, START, STOP, and WAIT [BRIN75, BRIN77, ZEPK74]. Except for EMPTY, these all interact with the kernel, either by issuing a call to a kernel routine, or by accessing a fixed location known to the kernel and interpreter. Since the kernel for the Microengine has not yet been designed, these instructions have been "commented out" of the code and must be modified at some later time. EMPTY returns a boolean value indicating whether a QUEUE variable is empty, without any kernel interaction. The code sequence SLDC00 EQUI emulates this instruction on the Microengine.

The three heap control instructions SETHEAP, NEWINIT, and NEW have not yet been changed, but rather, "commented out". SETHEAP initializes the heaptop pointer so that it points to the bottom of the heap space, thus making the heap "empty". The equivalent Microengine code is:

```
SLDC02      push heap pointer register (SPLOW) number
SWAP        reverse TOS and TOS-1 words
SPR         put TOS value in heap pointer register.
```

NEW allocates heap space for a new object and returns a pointer to it. Figure 33 shows Microengine code which performs a similar function. There are two unresolved issues here. The obvious one concerns the

```
    SLDC04              stack pointer register number (SP register)
    LPR                 push stack pointer
    LDCB (100)          keep 100 bytes between heap and stack
    SBI                 stack grows toward low addresses
    SLDC02              heap pointer register number (SPLOW register)
    LPR                 push heap pointer
    push object's length   (choose one of 3 possible instructions)
    ADI                 calculate heap pointer after allocation
    LEQI                test (SP - 100) <= (SPLOW + object length)
    FJP (ok)
    heap limit error code

ok: SLDC02
    LPR                 push heap pointer
    STO                 store (indirect) pointer to object into
                              pointer variable

    SLDC02              SPLOW register number, used with SPR below
    SLDC02
    LPR                 push heap pointer
    push object's length
    ADI                 calculate new heaptop
    SPR                 set heap pointer register to new heaptop
```

FIGURE 33. Microengine code to emulate the NEW Pascal virtual instruction. This code performs three general functions: 1) check for collision of heap and stack, 2) store the pointer to the new object in the object's pointer variable, and 3) reset the heap pointer to the next word of free space in the heap.

nature of the "heap limit error code". Whether it should be a call on a well-known kernel routine or in-line code has not been determined. The second issue was discovered at the time of writing, and is centered on the problem of ensuring, at run time, that the stack and heap will not collide. The virtual instruction NEW has an operand which includes the stack requirement of the routine in which it appears. During execution, it compares the extent of the heap after the contemplated allocation, and the maximum extent of the stack during the routine in order to determine if a collision is possible. The MCPASCAL compiler generates the stacklength operand in pass seven by referring to the routine's entry in STACKTABLE. In MEPASCAL, however, STACKTABLE is removed by pass six, so pass seven does not have routine stack requirements available to it. The code in figure 33 assumes (rather naively) that the routine's stack will not grow by more than 100 bytes from its current position. This is a rather arbitrary "safety factor" and not nearly as desirable as the original mechanism. If possible, STACKTABLE should be passed on to MEPASS7 in order to restore that mechanism. NEWINIT is the same as NEW, except that the newly-allocated heap space is initialized to zero. Its Microengine equivalent is shown in figure 34. Besides the two issues mentioned above, an additional concern is the large amount of space consumed by the initialization code.

Sixteen other virtual instructions and their transformations will be described here. They are:

COPYTAG EOM FALSEJUMP FIELD
FUNCVALUE INITVAR IO JUMP
MESSAGE NEWLINE NOT POINTER
POP RANGE REALTIME VARIANT.

There are two Microengine instructions which are equivalent to FALSEJUMP-- FJP and FJPL. The difference is that FJP uses a signed

```
      SLDC04
      LPR            push stack pointer
      LDCB (100)     stack-heap separation
      SBI
      SLDC02
      LPR            push heap pointer
      push object's length
      ADI
      LEQI           test (SP - 100) <= (SPLOW + length)
      FJP (ok)
      heap error code

ok:   SLDC02
      LPR            push heap pointer
      STO            store pointer to new object in pointer variable

      SLDC02
      LPR            push object's base address; will be used
                          later to scan through new object
      DUP1           get copy of pointer to new object
      SLDC02         SPLOW register, used with SPR below
      SLDC02
      LPR            push heap pointer
      push object's length
      ADI            calculate new heaptop
      SPR            set heap pointer to new heaptop

next: SLDC02
      LPR            push new heaptop
      GEQI           object scan pointer >= new heaptop?
      LNOT
      FJP (exit)
      SLDC02
      ADI            point to next word in object
      DUP1           copy object scan pointer for use
                          by GEQI instruction in next iteration
      DUP1           push another copy for STOring the initial value
      SLDC00         initial value to be inserted in object
      STO            set a word in the object to zero
      UJP (next)     prepare to initialize next word in object

exit: FJP (0)        pop object scan pointer
```

FIGURE 34. Microengine code to emulate the Pascal virtual instruction NEWINIT, which allocates heap space for a new object and initializes the object to zero. Compare with figure 33. After the space has been allocated and the heap pointer updated, the heap pointer points to the word which immediately follows the new object.

byte operand whereas FJPL (False JumP Long) uses a signed word operand. While FJP takes less space, its range is also shorter. The longer instruction is used for all false jumps, even though it might be suboptimal, in order to keep code assembly simple. If both instructions were used, code addresses could not be known in pass six as they are now. JUMP also has two equivalents (UJP-- Unconditional JumP, and UJPL-- Unconditional JumP Long), and the longer instruction is always used, for the reasons just mentioned. FIELD'S equivalent is INC (INCrement field pointer), except that the offset operand must be converted to word measure. LNOT (Logical NOT) performs the same function (boolean negation) on the Microengine as the virtual machine instruction NOT.

The Microengine has no direct equivalent of POP, but FJP and NFJ (Not equal False Jump; that is, jump if equal) can be combined into a suitable substitute. POP has an operand which is the number of bytes to be discarded from the stack. An odd number of bytes is never popped since the stack consists only of whole words. Since no Microengine instruction merely discards a variable portion of the stack without side effects, the compiler generates a variable number of NFJ(0) instructions and possibly one FJP(0) instruction. NFJ compares the two top-of-stack words as integers and the jump is made if they are equal. In either outcome the two words are discarded from the stack. FJP tests the top-of-stack word, discards it, and jumps if it is FALSE. Both instructions use a signed distance operand to calculate the jump address. If the distance is zero, the next sequential instruction will be executed, regardless of the test results, and the stack operand(s) will be popped from the stack. Thus, NFJ(0) is equivalent to POP(4), and FJP(0) to POP(2). Presumably, the number of bytes to be popped at any given time will be relatively small, so the compiler generates enough NFJ(0) instructions such that after their execution either no words or one word remains to be popped. If one word remains, an FJP(0) is generated, otherwise not.

RANGE checks the top-of-stack integer against its operands to ensure that the integer is within the range which they specify. If it is not within the range, a run-time error occurs. The Microengine operator CHK (CHecK) does the same thing (including triggering a run-time error) except that the range-specifying operands are taken from the stack, and the integer to be checked is the TOS-2 item. IO, POINTER, and REALTIME interact with the kernel, so their equivalent code has not been finalized. IO and REALTIME must call the kernel in all cases. POINTER ensures that pointer variables contain some nonzero value. A run-time error results if the value is zero, and the Microengine equivalent of the error code is undetermined at this time.

VARIANT also uses error code which is as yet undetermined, and will also require changes to one of the earlier passes in MEPASCAL. Figure 35 shows how the instruction appears in the final code. VARIANT assumes that the address of the variant record is the top-of-stack word. Its first operand is the field offset of the tag field, and the second is the set of tags (<u>always</u> one word long) for which access is being requested. The tag field contains the <u>bit number</u> of the tag which is currently "legal". During instruction execution, the bit in the request set whose number is given in the tag is examined. If that bit is not "on", a run-time error occurs, otherwise execution proceeds. Notice that VARIANT <u>does not alter the stack</u>, leaving the record address in place. Use of the bit <u>number</u> for the

```
0001 TYPE
0002 IDENTIFIER = ARRAY[1..12] OF CHAR;
0003             "0       1        2       3"
0004 ARGTAG = (NILTYP, BOOLTYP, INTTYP, IDTYP,
0005             "4       5        6       7"
0006           PTRTYP, XTYP,    YTYP,   ZTYP);
0007 RECTYPE = RECORD
0008           FIRST: INTEGER;
0009           CASE VRNTTAG: ARGTAG OF
0010               "0       1"
0011             NILTYP, BOOLTYP:          (BOOL: BOOLEAN);
0012               "2      5     6     7"
0013             INTTYP, XTYP, YTYP, ZTYP: (INT:  INTEGER);
0014              "3"
0015             IDTYP:                    (ID:   IDENTIFIER);
0016              "4"
0017             PTRTYP:                   (PTR:  REAL)
0018           END;
0019
0020 VAR   VREC: RECTYPE;
0021
0022 BEGIN
0023 VREC.VRNTTAG := INTTYP;     "TAG IS 2"
0024 VREC.INT := 100;           "USING BITS 2, 5, 6, 7"
0025
0026 VREC.VRNTTAG := YTYP;      "TAG IS 6"
0027 VREC.INT := 200;           "USING BITS 2, 5, 6, 7"
0028
0029 VREC.VRNTTAG := PTRTYP;    "TAG IS 4"
0030 VREC.PTR := 300.0          "USING BIT 4"
0031 END.
```

```
7777777777777777777777777777777777777777777777777777777777777777777777
7                MCPASCAL INTERMEDIATE CODE PASS 7                   7
7777777777777777777777777777777777777777777777777777777777777777777777
JUMP(2)
           BEGNPRCS(22)
LINE   23   GLOBLADD(-16)   FIELD(2)   PUSHCONS(2)   COPYWORD
LINE   24   GLOBLADD(-16)   VARIANT(2,9984)   FIELD(4)
           PUSHCONS(100)   COPYWORD
LINE   25
LINE   26   GLOBLADD(-16)   FIELD(2)   PUSHCONS(6)   COPYWORD
LINE   27   GLOBLADD(-16)   VARIANT(2,9984)   FIELD(4)
           PUSHCONS(200)   COPYWORD
LINE   28
LINE   29   GLOBLADD(-16)   FIELD(2)   PUSHCONS(4)   COPYWORD
LINE   30   GLOBLADD(-16)   VARIANT(2,2048)   FIELD(4)
LINE   31   CONSTADD(16)   PUSHREAL   COPYREAL   ENDPRCS
```

FIGURE 35. Example of the VARIANT virtual machine instruction, the source code which produces it, and the final code which surrounds it. Tag bit numbers are shown above the identifiers. Some of pass seven's output is not shown here since it is irrelevant.

current tag value is apparently desirable when running on a PDP-11 since hardware shift instructions can then be used advantageously [DIGI76, ZEPK74]. For the Microengine, however, it will be necessary for the tag field to hold the value of the current tag bit, not its number. That change has not been made, and the compiler pass which must be modified has not yet been identified. Once that change has been made, the equivalent Microengine code would be:

```
     DUP1              save copy of record address
     IND (word displacement)   push current tag value
     LDCI (tag set)  push tags for which access is requested
     LAND              logical AND the two words together
     SLDC00            push word with all bits "off"
     EFJ (ok)          if request ok, TOS-1 will have an "on" bit
     variant error code
 ok: next instruction.
```

If the NUMBER compiler option is specified, MCPASCAL generates NEWLINE instructions at points in the object code which correspond to the beginning of source lines. The NEWLINE instruction stores the line number operand into a word of the local markstack. However, the Microengine has no markstack space for the line number, and in addition, its markstack fields are not generally accessable (section 5.2.2). The line numbering mechanism has been implemented by specifically reserving the first word of local variable space in every routine for the current line number. The Microengine code equivaluent to NEWLINE consists of two instructions-- one of the "push constant" instructions, and STL(1). The routine DISPL in MEPASS6 adjusts all references to program variables to take into account the reserved word. In addition to the push and store instructions, pass six also puts the NEWLIN_2 virtual instruction into the code stream. This is a crutch for the intermediate code mnemonics program MEMNEM. NEWLIN_2 makes it possible for MEMNEM to identify the beginning of each source line without resorting to a lookahead technique (see Appendix A).

No Microengine code is generated when MESSAGE1 and EOM1 are encountered in pass six. They are merely inserted into the intermediate code as virtual instructions for use by pass seven.

In general terms, the virtual instruction FUNCVALUE pushes onto the stack enough space to hold the value which will be returned by a function which is about to be called. It pushes either one or four words, depending on the type of the return value. However, there is a twist. Class and monitor entry routines expect the address of the component's permanent variables to be the first "parameter" pushed on the stack. But the code generated by the compiler carries out the actions:

1) push the address of the permanent variables record;

2) push space to receive the function value;

3) push parameters;

4) call the entry routine.

If the function-value space is placed directly on the top of the stack, the permanent variables record address will not be in the proper position. The function-value space should be placed between the two top-of-stack words. FUNCVALUE performs this stack-space insertion for class and monitor function entries. There are four equivalent Microengine code sequences. As FUNCVALUE comes into MEPASS6, it has a mode operand and a type operand. If the mode is "process entry" or "procedure", one or two SLDC00 instructions are generated to push space at run time for word-type or real-type function values. If the mode is "class entry" or "monitor entry", the code sequences are:

```
WORD-TYPE         REAL-TYPE
SLDC00            SLDC00
SWAP              SWAP
                  SLDC00
                  SWAP.
```

At run time, these instructions will "insert" one or two words of zero between the permanent variables record address and the next word on the stack.

Finally, INITVAR and COPYTAG were commented out of pass six since they are never produced by pass five of the concurrent compiler. They are presumably produced by the sequential compiler. Apparently, passes six and seven of the two compilers were so similar that they were combined, and are used by both translation systems.

5.2.4 "Difficult" Instruction Modifications

The virtual instruction transformations described in the preceding section were relatively easy to make since they are, for the most part, independent of the underlying hardware architecture. The transformations described here were more difficult since these instructions make use of some architectural assumptions.

5.2.4.1 Value and Address PUSH Instructions

Pass five generates four instructions for pushing values and addresses onto the operand stack. They are PUSHCONST1 (PUSH the value of a CONSTant), PUSHVAR1 (PUSH the value of a VARiable), PUSHIND1 (PUSH a value, INDirect), and PUSHADDR1 (PUSH an ADDRess). Pass six analyzes these operators and their operands in order to select the appropriate final-code instructions from nearly a dozen possibilities.

The operand for PUSHCONST1 is the 16-bit value of the constant to be pushed onto the stack. For the virtual machine there is just one instruction for pushing short (immediate) constants, but there are three for the Microengine. Based on the value of the constant, MEPASS6 selects the hardware instruction which will take the least possible amount of code space.

PUSHVAR1 has three operands: the virtual data type of the variable, the addressing mode, and its displacement from either the local or global base register of the virtual machine. If the variable is a word or less, its type will be WORDTYPE and it can be pushed directly. Otherwise, it must be pushed indirectly. The choice of the direct-push instruction to be generated depends on the variable's mode, and is made by the MEPASS6 routine PUSHVALUE, discussed below. The Microengine code sequence for an indirect push consists of two instructions; one to place the address of the variable on the stack, and another to actually use that address and replace it with that location's content. The procedure PUSHADDRESS chooses the first of these, from among several possibilities, based on the addressing mode, and PUSHINDIRECT picks the second, based on the virtual data type.

When the PUSHADDR1 and PUSHIND1 intermediate-code instructions are encountered by pass six, their operands are passed directly to the routines PUSHADDRESS and PUSHINDIRECT, respectively. Those two routines (described below) determine and generate the corresponding final code.

The procedure PUSHVALUE uses a variable's address mode to choose among three Microengine instructions which push onto the stack the value of a word in some data space. If the mode is "procedure", "class entry", or "monitor entry" then the word will be (at run time) in the data space which is local to the routine active at that moment. The Microengine instruction LDL (LoaD Local word) is generated for those modes. The operand for LDL is the variable's displacement within the local data space, and is calculated by the routine DISPL, described in section 5.2.2. "Process", "class", and "monitor" modes indicate that the variable is a permanent variable of a system component, and so it will be addressable by a displacement from the global base register. (At run time the global base register will have been set so that it points to the *correct* permanent-variables record in the initial-process data space-- see sections 5.2.4.5 and 5.2.4.6.) So, PUSHVALUE generates an LDO (LoaD glObal word) operator and the appropriate displacement. "Program" mode variables will be in the global data space of a sequential program, so they will be addressable from the global base register, and an LDO instruction is generated. Variables whose mode is "process entry" require special handling. If the virtual displacement (the displacement as it comes from pass five) is negative, the object being referenced is a *variable* in the entry routine's local data space, and an LDL instruction will serve to push it onto the stack during execution. If the displacement is not negative, the object is a *parameter* (or function value) of the routine. It cannot be addressed relative to the local base register since it is local, not to the entry routine, but to the interface routine which called it (see section 5.2.2 and figure 29). The object *can* be addressed as an "intermediate" (neither local nor global-- lexically somewhere in between) word; that is, relative to *its own* local base and the distance (in static links) from the current local base to the required one. To push the variable at run time, PUSHVALUE generates an LOD (LOaD intermediate word) instruction which has two operands. One is the variable's displacement from its local markstack, calculated by DISPL. The other is the number of static links which must be traversed to reach the markstack to which the variable is local. The instruction used to invoke the process entry routine (CXL) makes the entry routine's static link field point to the interface routine's markstack, so only one static link needs to be traversed.

The PUSHADDRESS routine selects Microengine code to push onto the stack the address of some data object. The strategy used to make the selection is exactly the same as that used by PUSHVALUE. The instructions which can be generated are LLA (Load Local Address), LAO (Load Address, glObal), and LDA (LoaD intermediate Address). LLA is generated for "procedure", "class entry", and "monitor entry" modes. LAO is generated for "process", "class", "monitor", and "program" modes. For "process entry" mode LLA is generated if the object is a local variable. Otherwise, LDA is generated with a static-link-distance operand of one in order to access the object relative to the markstack of the interface procedure. Variables are never located in the constant pool, so PUSHVALUE never has to deal with constant-mode references. There are cases, however, where the address of a long constant must be pushed onto the stack. For constant-mode references PUSHADDRESS generates an LCA (Load Constant Address) instruction. The operand for LCA is the offset (in words) of the constant, relative to the start of the segment.

PUSHINDIRECT generates code which will push onto the stack the data item which is the target of the top-of-stack pointer. For objects of type WORDTYPE, REALTYPE, and SETTYPE that pointer occupies _one_ stack word. It is _two_ words for BYTETYPE objects, however. The routine generates an SINDO (Short INDex (0) and load word) for pushing word-length objects. During its execution, the instruction adds the index (0, in this case) to the pointer and pushes the word which the new pointer indicates. For reals, an LDM (LoaD Multiple words) instruction is generated with an operand value equal to the word length of Microengine real values-- two words. An LDM is also generated to push sets indirectly. Its operand is the length of the set to be pushed. The hardware actually supports sets of varying length, but since the virtual machine only supports eight-word sets, MEPASCAL only generates fixed-length sets, and so the operand to the LDM is fixed at eight. Since the hardware expects to find the length of the set as the top-of-stack word (see section 5.2.3), the routine also generates an instruction to push the set's (fixed) length value. For byte objects an LDB (LoaD Byte) instruction is generated. At run time, however, this instruction expects the TOS-1 word to be the address of the first word of a byte array (two meaningful bytes per word), and the TOS word to be the offset (in bytes) from that address to the target byte. During execution, LDB consumes both stack words and leaves in their place a word which contains the target byte in the low-order position, and zero in the high-order position. This mechanism is quite different from that of the Concurrent Pascal virtual machine. The virtual instruction PUSHBYTE assumes that the TOS word by itself points directly to the target byte. Thus, there is an implicit assumption that the underlying hardware is byte addressable. Refer to the next section for information on how the Microengine byte pointer is built.

5.2.4.2 INDEX and BYTE Instructions

The virtual machine instructions PUSHBYTE and COPYBYTE have equivalent Microengine instructions in LDB (LoaD Byte) and STB (STore Byte). PUSHBYTE and LDB push the byte to which the top-of-stack item points. COPYBYTE and STB write the low-order byte of the top-of-stack word into the location where the TOS-1 item points. The actions of the two pairs of instructions are identical; however, the pointers required for the Microengine instructions are not the same as those for the virtual instructions. Pointers on the virtual machine are all alike-- one word long and measured in bytes. The Microengine uses *three* pointer formats, from one to three words long. *Packed field* pointers are of no concern here. *Word pointers* are one word long, apparently use byte measure, and point to the low-address byte (low-order bits, even address) of a word. Word pointers are generated by instructions such as LLA (Load Local Address). *Byte pointers* consist of two words (see figure 36). The TOS-1 word is a word pointer-- call the byte to which it points the "base". The TOS word contains the offset (byte measure) from the base to the byte which is the ultimate target [REGE78]. If MEPASCAL were to simply generate LDB and STB in place of PUSHBYTE and COPYBYTE, at run time the Microengine instructions will attempt to use the two top-of-stack words as a byte pointer. In fact, the TOS word will be a pointer in *virtual machine* format and the TOS-1 word will be other irrelevant data which must remain untouched.

The solution to this problem with pointer formats is in the code which *generates* byte pointers. Byte pointers are generated only for references to Concurrent Pascal strings (ARRAY[N..M] OF CHAR). Notice that in Concurrent Pascal *and* UCSD Pascal, CHAR variables (and CHAR

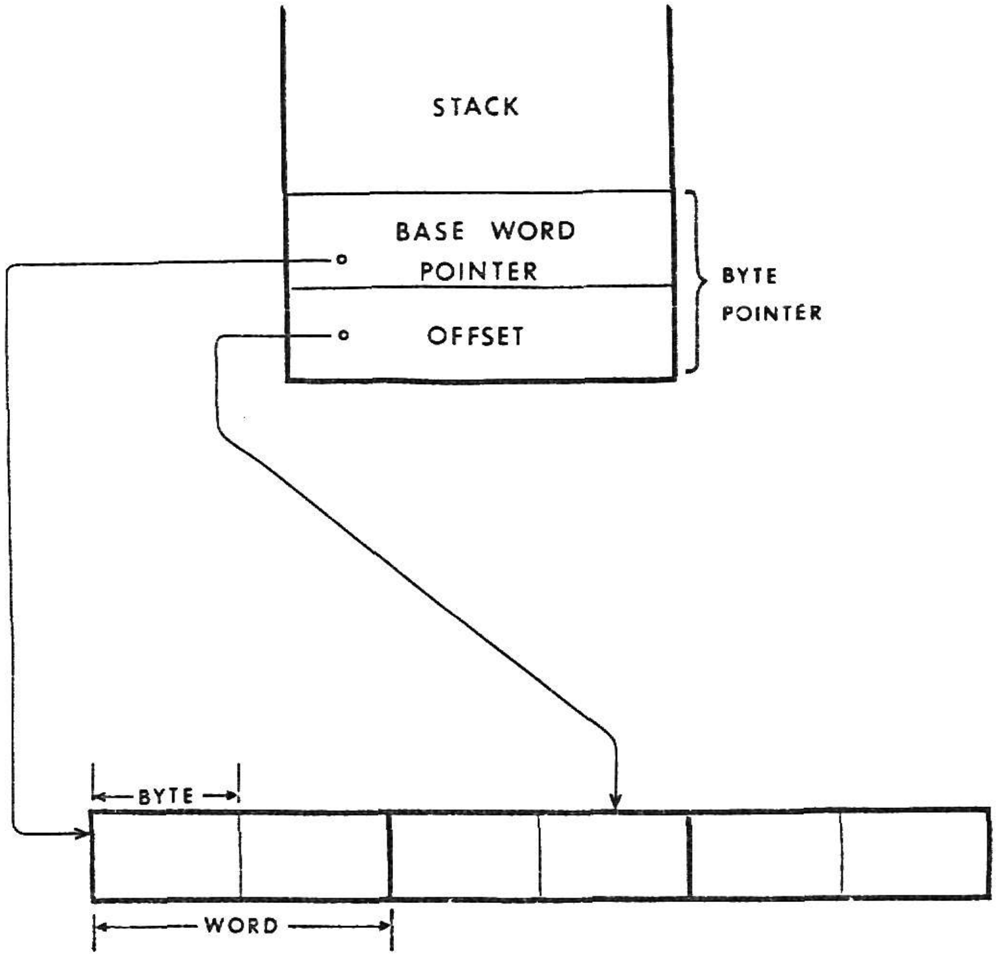


FIGURE 36. Microengine byte pointer to the fourth byte (offset = 3) in a string.

constants when they are on the stack) occupy only one word in which the high-order byte is zero. Concurrent Pascal arrays of characters are identical to UCSD Pascal packed arrays of characters (two characters in a word). UCSD arrays of characters have one character per word, an arrangement which is not supported by Concurrent Pascal. In the final analysis, byte pointers are generated only when a byte is referenced as an element of an array. The virtual instruction INDEX is responsible for generating the final address of an array (any array--CHAR or not) element just before being pushed on the stack or stored, so it is the instruction which was modified to handle the Microengine byte pointer format.

INDEX assumes that the TOS-1 word is a pointer to the base of an array, and that the top-of-stack word is an integer index into the array. During its execution, the instruction compares the index value to the bounds of the array being referenced. If it falls outside, a run-time error occurs, otherwise the index (adjusted to a zero base) is multiplied by the length of an array element to yield the target element's offset (in bytes) from the base of the array. The base and offset are then added, leaving the address of the target element on top of the stack. A direct conversion to a Microengine code sequence would work, except in the case where the array elements are bytes. For byte arrays, the final arithmetic addition of base address and offset must be skipped in order to leave a Microengine byte pointer on the stack. MEPASS5 incorporates a change to the INDEX2 instruction which it emits (section 5.1). In MCPASS5, INDEX2 has three operands. In the MEPASCAL version of pass five, the instruction has a fourth operand which is the virtual data type of the array elements. When INDEX1 is encountered by MEPASS6, the type operand is used to determine if an instruction should be generated to add the base address and element offset. In retrospect, it seems that the LENGTH operand conveys the same information as the TYPE operand-- the element is either BYTETYPE or not (element lengh is either one or not). If the element length is one byte. then it must need a byte pointer. If the length is used as the discriminant, the type operand and the change to pass five become unnecessary. The Microengine code which is equivalent to INDEX is:

```
    push min.      push minimum array bound
    push max.      push maximum array bound
    CHK            range check
    push min.      adjust index to zero base
    SBI
   {IXA (size)     add base address and offset,
                        yielding element address},
```

where the IXA (IndeX Array) is <u>not</u> generated for a reference to a BYTETYPE element.

5.2.4.3 CASEJUMP

There is a radical difference in the way CASE statements are handled on each target machine. An example of a virtual machine CASE statement is shown in figure 37, and the syntax is given in [HART76]. Notice that the CASEJUMP instruction <u>follows</u> the code for the cases. This makes for a good deal of wild branching during execution, but it also <u>guarantees</u> that pass <u>six</u> of the compiler will be able to resolve the displacements which are operands of the CASEJUMP operator. The displacements are the distances to the code for each case. During execution, the instruction checks the top-of-stack selector value against the legal range defined by the first two operands. If the selector value is outside the range, a run-time error is triggered. If it is in bounds, the selector is used to find the appropriate displacement operand, which is then algebraically added to the virtual

```
0001 VAR
0002 A, B, C, D, E, F, SELCTR: INTEGER;
0003
0004 BEGIN
0005 A := 100;
0006 CASE SELCTR OF
0007    3:     B := 2;
0008    8, 11: C := 3;
0009    9:     D := 4;
0010    12:    BEGIN
0011           E := 5;
0012           F := 5;
0013           END;
0014    6:     F := 6
0015 END;
0016 F := 200;
0017 END.
```

```
777777777777777777777777777777777777777777777777777777777777777777777777
7                MCPASCAL INTERMEDIATE CODE PASS 7                      7
777777777777777777777777777777777777777777777777777777777777777777777777

JUMP(2)
           BEGNPRCS(4)
LINE    5   GLOBLADD(-2)    PUSHCONS(100)    COPYWORD
LINE    6   PUSHGLBL(-14)    JUMP(138)
LINE    7
LINE    7   GLOBLADD(-4)    PUSHCONS(2)    COPYWORD    JUMP(146)
LINE    8
LINE    8   GLOBLADD(-6)    PUSHCONS(3)    COPYWORD    JUMP(124)
LINE    9
LINE    9   GLOBLADD(-8)    PUSHCONS(4)    COPYWORD    JUMP(102)
LINE   10
LINE   10
LINE   11   GLOBLADD(-10)    PUSHCONS(5)    COPYWORD
LINE   12   GLOBLADD(-12)    PUSHCONS(5)    COPYWORD
LINE   13   JUMP(58)
LINE   14
LINE   14   GLOBLADD(-12)
LINE   15   PUSHCONS(6)   COPYWORD   JUMP(32)
LINE   15
CASEJUMP(3,9,-142,18,16,-38,12,-130,-110,6,-136,-94)LINE   15
LINE   16   GLOBLADD(-12)   PUSHCONS(200)   COPYWORD
LINE   17   ENDPRCS
```

FIGURE 37. MCPASCAL CASE statement.

program counter to realize the case jump. Since there is not a fixed number of cases in a CASE statement, the operator has a variable number of operands.

The Microengine case jump operator (XJP), on the other hand, has only one immediate operand, and it points to the place in the constant pool where all the other necessary information can be found. The constant pool information occupies as many words as there are cases, plus two words. The first word is the minimum legal value for the selector, and the second word is the maximum value. The next words contain the displacements to the code for each case, relative to the byte following the XJP instruction. As mentioned in section 5.2.1, the XJP selector bounds and case offsets cannot be carelessly tossed into the constant pool. Long constants are sprinkled throughout the intermediate code stream until pass six removes and collects them in the heap table CONSTTABLE. Earlier passes have counted the number and relative positions of the long constants, and generated addressing commands based on the assumption all case jump information would be in the code stream. All case information which belongs in the constant pool is removed from the code stream and placed in XJPTABLE when MEPASS6 encounters CASEJUMP1. When MEPASS7 generates the final object code file it will emit one word which points to the end of the segment (see figure 7) and then build the constant pool by dumping the contents of CONSTTABLE and XJPTABLE, in that order. Addressing commands for constants which are not for case jumps will be valid because those constants have not changed position relative to the start of the constant pool. (The routine which calculates displacements for addressing commands-- DISPL in MEPASS6-- compensates for the final-word pointer.) The value of the XJP operand (segment-relative offset of the selector bounds and case offsets) is the offset of the required information in XJPTABLE (variable XJPOFFSETPTR in MEPASS6) plus the size of CONSTTABLE (known from data in the PASSLINK record) plus one word (the pointer to the segment's final word). When CASEJUMP comes into pass six, the case offsets are represented by case labels. The labels are guaranteed to be resolvable into offsets since the location-counter value of each case label was inserted into JUMPTABLE as it was encountered, and all the cases appear in the intermediate code <u>before</u> the CASEJUMP1 operator. The labels in the CASEJUMP1 instruction are used as indices into JUMPTABLE to retrieve those location-counter values, and the differences between them and the current value of the location counter are the offsets to be inserted in XJPTABLE. The correct placement of the case offsets complicates an otherwise trivial translation of CASEJUMP to the Microengine code sequence:

```
push min.           push minimun CASE index
push max.           push maximum CASE index
CHK                 check:   min. <= TOS-2 <= max
XJP (displacement)  casejump using constant-pool CASE table.
```

5.2.4.4 Routine Invocation Instructions

The six virtual instructions CALL, INITCLASS, INITMON, INITPROC, CALLPROG, and CALLSYS all invoke routines, but the circumstances which surround the use of each, and the side effects produced by each one vary.

CALL is the simplest of the six. In the final code, its operand is a signed value which is the distance to be jumped in order to reach the code of the called routine. During execution, the address of the instruction after CALL is pushed onto the stack as the return address and the operand is algebraically added to the virtual program counter so that the next instruction to be executed will be the first one in the called routine. By pushing the return address onto the stack, CALL begins the construction of the markstack (see figure 2) for the called routine. The first instruction of the called routine will finish it (see section 5.2.4.6). By the time CALL executes, other instructions have pushed onto the stack space for the function value and parameters, if required by the called routine. Only non-ENTRY, monitor ENTRY, and class ENTRY routines are invoked by CALL, so in execution sequence, CALL is always followed by either ENTER, ENTERMON, or ENTERCLASS. Along with other actions, these instructions finish building the markstack. In MEPASS6, the Microengine instruction CPL (Call Procedure Local) has been substituted for CALL, although it is not quite equivalent. The operand to CPL is the called routine's number. This happens to be the compiler-generated block label which is an operand of CALL1. During its execution, CPL not only causes a jump to the called routine, but also completely builds the markstack and sets aside local variable space for the routine.

INITCLASS is used to invoke the initial routine (BEGIN..END. block) of a class. Just before INITCLASS executes, other instuctions have pushed onto the stack the address of the "middle" of the record containing the class's permanent variables (see figure 25), and the parameters (access rights) required by the class. When the instruction executes, it pops the parameters from the stack into the component variable record, puts the return address on the stack, and jumps to the class initial code.

In order to move this instruction to the Microengine, pass five and the input syntax for pass six had to be changed. Even though INITCLASS uses the address of the class variable in copying the parameters, it *leaves* it on top of the stack for later use in setting the global base. For the Microengine, an STM instruction is used to pop the parameters and, as a side effect, it consumes the class variable address. The syntax change involved adding an instruction to duplicate the class address in order to leave a copy of it on top of the stack for later use. The original pass six input syntax for an *init stat* [HART76] was

```
---> varaddr ---> FIELD(disp) ---> arg list ---
                                              |
          -------------------------------------
          |
          ---> INIT(mode, label, parm length, var length) --->  .
```

In order to preserve the class address, DUPTOS was added, giving the syntax

```
---> varaddr ---> FIELD(disp) ---> DUPTOS ---> arg list ---
                                                          |
          -------------------------------------------------
          |
          ---> INIT(mode, label, parm length, var length) --->  ,
```

where DUPTOS translates to the Microengine hardware instruction DUP1. Once the access rights have been put in the class variable, the initial code for the class is invoked by a CPL instruction, so the Microengine code for INITCLASS is

```
          STM (paramsize)     pop "paramsize" words into class variable
          CPL (routine no.)   invoke initial routine of class.
```

The operation of, and equivalent code for, INITMON is the same as for INITCLASS. BEGINCLASS and BEGINMON instructions follow INITCLASS and INITMON *in execution*. They complete construction of the markstack and set the global base for the class or monitor.

INITPROCESS calls on the kernel to create a new process and get it started in its own initial routine. The kernel pops the parameters into the process variable and leaves its address on top of the stack. After the kernel call, the original process merely discards the address. The equivalent Microengine code for the kernel call has not yet been determined, and the instruction which pops the top-of-stack word is FJP(0). The execution sequence is somewhat unclear since two processes are executing concurrently after the kernel creates the new process. During the execution of INITPROCESS, the original process enters and returns from the kernel. The new process starts out in kernel code and the first virtual instruction it executes is BEGINPROC.

CALLPROG pushes the return address onto the stack and starts the execution of a sequential program stored in a variable whose address is on top of the stack. Because of the Microengine architecture, the mechanism for starting a sequential program is entirely different, as described in section 3.4. The code for invocation of a program is given below, and assumes that the code variable address is on top of the stack. In short, the code puts the address in the sequential program's segment information block, and then calls the initial routine in that segment. Figure 14 will be helpful for following all the pointers which must be chased. The Microengine code is

```
SLDC01        push CTP register number (-1)
NGI
LPR           push pointer to TIB (CTP register content)
IND (11)      push pointer to SIB vector (11th word in TIB)
INC (5)       increment by 5 words to yield pointer to SEGBASE field
                 in SIB no. 129. (SIB 129 is second record in
                 SIB vector, each SIB is 5 words long, and SEGBASE
                 is first field in each SIB.)
SWAP
STO           pop pointer to program variable into
                 SEGBASE field of SIB 129
CXL (129, 1)  invoke initial routine of sequential program.
```

The ENTRY routines of a process are the routines which provide operating system services to the sequential program running as a part of that process. In the virtual machine, the sequential program requests one of those services by executing the instruction

CALLSYS(index), where the operand is the index of the service in the sequential program prefix. CALLSYS is comparable to a conventional SVC instruction. Several events must have taken place at the time of sequential program invocation in support of CALLSYS, as described in section 3.4.1. Just before the program starts, code in the concurrent program puts on the stack a table of addresses of the process ENTRY routines. This jump table (figure 16) remains on the stack while the sequential program executes. CALLSYS uses this index operand to select an ENTRY routine address from the jump table, and then causes a jump to that address. The sequential program is, essentially, requesting system services by number. Notice that the operating system and sequential program must agree on the numbers assigned to the services. The assignments are based on the order in which the service identifiers appear in the interface definition (in the host process) and in the prefix (in the sequential program). Suppose, for example, that a process has eight ENTRY routines named OPEN, CLOSE, GET, PUT, ACCEPT. DISPLAY, MARK, and RELEASE, and that the sequential program SEQPROG will have access to the first six. That is, SEQPROG will only be able to use the first six; in fact, it will not even know of the existence of the other two. If the program and interface are defined as

```
SEQPROG (A, B: INTEGER; C: SEQL_CODE_TYPE);
ENTRY OPEN. CLOSE, GET. PUT, ACCEPT. DISPLAY;
```

then the sequential prefix must list those routines in the <u>same order</u> (although the identifiers can be different):

```
PROCEDURE OPENFILE (parameter list);
PROCEDURE CLOSEFILE (parameter list);
PROCEDURE GETPAGE (parameter list);
PROCEDURE PUTPAGE (parameter list);
PROCEDURE READCONSOLE (parameter list);
PROCEDURE WRITECONSOLE (parameter list);   .
```

If the two sequences fail to match (say the order of OPENFILE and CLOSEFILE is reversed), the results of a CALLSYS instruction will not be as expected (a call to OPENFILE will cause the process ENTRY routine CLOSE to be executed). As long as the interface and prefix definitions maintain the proper relationships, each program (concurrent and sequential) can be altered and recompiled independently of the other. This independence is possible because routines are known and invoked by their address, and CALLPROG indirectly takes the ENTRY routine address from the stack.

In the virtual machine the run-time jump table performs a mapping function from the prefix index number known to the sequential program to the corresponding ENTRY routine address-identifier. Since Microengine routines are known and invoked by their routine number, and the invocation instructions use only immediate operands, that mapping function must be performed by the interface segment described in sections 3.4.1 and 5.2.4.8. By the time the Microengine code equivalent to CALLSYS executes, the code segments (both shared and private to the host process) are configured as shown in figure 18. The interface segment consists of only one routine which operates as described in sections 3.4.1 and 5.2.4.8. On the Microengine, process ENTRY routines will be called by the interface routine, which is itself called by the sequential program. The interface routine requires a parameter which is the equivalent of the CALLSYS operand. The code to call the interface routine is:

```
push index     push parameter for interface routine
CXL (128, 1)   invoke interface routine in interface segment.
```

5.2.4.5 BEGIN Instructions

The virtual instructions BEGINCLASS, BEGINMON, and BEGINPROC are always the first instruction in the initial routine of classes, monitors, and processes, respectively. In execution they are always preceded by the corresponding INIT instruction. The BEGIN instructions are generated by pass six when ENTER1 is encountered, based on its mode operand. (ENTER1 also translates to the ENTER instructions described in the next section.) In general terms, BEGINCLASS checks for the possibility of a heap-stack collision during the routine, finishes construction of the markstack (begun by INIT), and sets the global base register so that it points to the "middle" of the class variable (see figure 25).

Before describing the equivalent Microengine code itself, some comments must be made concerning the actions of the compiler while generating that code. When MEPASS6 encounters ENTER1, the instruction operands are copied so that their values are preserved during the compilation of the entire routine (the variables ARGx take on new values almost every time a virtual instruction is read from the input file). The "save" variables have the following meanings:

BLOCK-- the numeric label by which the routine is known and which will be used by Microengine instructions (CPG and CPL, for example) to invoke the routine;

PARAMLENGTH-- the number of bytes of parameters for the routine;

VARLENGTH-- the number of bytes of local variables required for this routine;

STACKLENGTH-- the number of bytes of extra stack space to be reserved for the routine.

In the original compiler. PARAMLENGTH and VARLENGTH for initial routines of classes, monitors, and processes are always zero, reflecting the fact that the permanent variables are not located on the stack (rather, in the component variable record) and that the parameters (access rights) are popped off the stack into the component variable record by INIT. In MEPASCAL PARAMLENGTH and VARLENGTH always contain the corresponding length values (modification made to pass four-- see section 5.1). The length of the variables area is needed to determine the location of parameters, since variables intervene between the base (markstack) location and the parameters (see figure 19). After saving the instruction operands, AFTERBEGIN is set TRUE to indicate that the compiler is now in a routine bódy. In prior passes, NEWLINE instructions were generated for each source line, including lines in the declaration parts of routines. Since the intermediate code for declarations has been removed, the declaration parts consist of only NEWLINE operators. NEWLINE operators encountered <u>outside</u> of routine bodies (AFTERBEGIN = FALSE) are deleted from the code sent to pass seven. TEMP and MAXTEMP are used to calculate the run-time stack requirements of the routine (section 4.2). They are initialized at zero since the compiler has not yet encountered any code which will, at run time, push anything on the stack. The location counter (which is relative to the beginning of the <u>segment</u>) is incremented by two words to account for the space which the routine's EXIT-IC and DATASIZE fields will occupy in the final code (see figure 7).

After the compiler has completed the actions which are common to all routine modes, code is generated and other actions are taken, based on the value of the mode operand. For "class" mode the routine's DATASIZE value is inserted into its corresponding position in DATASIZETABLE. DATASIZE is the number of words this routine requires for local variables. This is not the same as VARLENGTH. In "class" mode, VARRLENGTH measures variable space in the class record, whereas DATASIZE is the number of words which must be pushed on the stack for variables. Since the parameters (access rights) and permanent variables are in the class record, only two words of local variable space are needed on the stack. These are the words for storing the source line number and old global base value. POPLENGTH is the number of bytes of stack space which must be popped at the end of the routine in order to return the stack to its configuration before the routine invocation. It will become the operand to the RPU instruction which terminates the routine, and in general, it is the number of bytes of stack space occupied by the routine's variables and parameters. For a class routine there are two words of variables and one word of parameters (the class record address left by code equivalent to INITCLASS). Code is generated to save the source line number and the old global base address, and store the new global base address into the BP register. The address of the class record cannot be used directly as the new global base since the Microengine hardware takes into account the size of the markstack when variable and parameter references are made (see section 3.4.2, figure 9, and figure 19).

Consequently, the Microengine code sequence which is similar to BEGINCLASS includes instructions to reduce the global base address by the size of a markstack, as shown here:

```
push line no.       save souce line number in local word 1
STL (1)
SLDC06              push global base register (BP) number
LPR                 push global base value
STL (2)             save global base in local word 2
SLDC06              prepare to put new global base in BP
LDL (3)             push class record address
SLDC08              push size (bytes) of a markstack
SBI                 BP must point to bottom of imaginary markstack
SPR                 put new global base in BP register.
```

This code does not include a check for heap-stack collision. Presumably, that code can be added when MEPASS6 is modified to pass STACKTABLE on to MEPASS7 as discussed in section 5.2.3. Also, there is no code here to build any part of a markstack since it is built completely by the Microengine code equivalent to INITCLASS.

The code for BEGINMON is the same as BEGINCLASS, except that after the new global base has been established, a call must be made to the kernel in order to initialize the monitor's gate.

Although the compiler actions are the same, the equivalent code for BEGINPROC is much simpler than that for the corresponding class instruction. The DATASIZE value (again, two words) is entered into the DATASIZETABLE and the POPLENGTH is calculated as two words (line number and old global base). POPLENGTH does not include the "parameter" location containing the process record address because that location is in the data space of the initializing process. The equivalent code for INITPROC includes a FJP(0) instruction to pop the component address (see section 5.2.4.4). The Microengine code for BEGINPROC is

```
push line no.       save source line number in local word 1
STL(1).
```

The kernel will handle establishment of the new global base address and construction of the markstack. Notice that there is no old global base since the process did not have any prior existence.

5.2.4.6 ENTER Instructions

The virtual instructions ENTER, ENTERCLASS, ENTERMON, ENTERPROC, and ENTERPROG are always the first instructions of non-initial routines. Non-ENTRY routines on the virtual machine always begin with an ENTER instruction, and in execution sequence it is always preceded by CALL. Generally, it checks for the possibility of a stack-heap collision, finishes construction of the markstack begun by CALL, and sets aside stack space for local variables. It does not affect the global base. When it encounters an ENTER1 instruction with a "procedure" mode operand MEPASS6 calculates the routine's DATASIZE and POPLENGTH values and generates a simple code sequence. Unlike initial routines, local variables actually reside on the stack, so DATASIZE is the number of words the routine needs for explicitly declared local variables (VARLENGTH) plus two words reserved for the source line number and old global base. Although the latter is not necessary, it is included for the sake of consistencey-- all routines have local words one and two reserved. The global base word could be removed as long as the routines and calculations affected by that removal are also modified. The calculated values for POPLENGTH and DATASIZE would have to be reduced by one word, and the procedure DISPL would need to use a separate calculation for determining variable displacements. The number of bytes to be popped off the stack at the end of the routine is determined by the length of the routine's parameters (PARAMLENGTH), the size of the declared variables (VARLENGTH), and the two local reserved words. The sequence of generated Microengine code is

```
push line no.     put source line number in local word 1
STL (1).
```

Just as for BEGINCLASS (section 5.2.4.5), there is no code here to check for a stack collision, or to explicitly build any part of the markstack.

ENTERCLASS is the first instruction of class ENTRY routines. Its function on the virtual machine is the same as BEGINCLASS (section 5.2.4.5). Consequently, the machine code which is generated in place of it is also the same. The compiler actions are different, however. The DATASIZE value for the routine is the size of the explicitly declared local variables (VARLENGTH) plus two words for the line number and old global base. Unlike the code for ENTER, the code for ENTERCLASS does use the local word reserved for the old global base. At the conclusion of the routine, its parameters, the address of the class record, its local declared, and its local reserved variables must be removed from the stack, so POPLENGTH is the sum of PARAMLENGTH, VARLENGTH, and three words.

ENTERMON is the first instruction of monitor ENTRY routines and performs the same actions as ENTERCLASS. It also makes a call on the kernel to request passage through the monitor's gate. MEPASS6 calculates DATASIZE and POPLENGTH the same way as for ENTERCLASS, then generates identical code. After the kernel call mechanism is known, code will be generated to request gate entry from the kernel.

The virtual instructions ENTERPROG and ENTERPROC were difficult to move to the Microengine because they are affected by the design and operation of the concurrent-sequential interface. ENTERPROG is the first instruction of the initial routine of a sequential program, so in execution sequence it is always preceded by CALLPROG. It completes construction of the markstack begun by CALLPROG, checks for a possible stack collision, and allocates stack space for the initial routine's local variables. The local variables of the initial routines are the global variables of the entire sequential program, so the instruction also resets the global base register so that it points to the same place as the local base register. In addition, the instruction sets to the value 1, a word (JOB) in the kernel associated with the invoking process. This is used to indicate that the process is in sequential (user) code, not concurent (operating system) code. The Microengine code generated in place of ENTERPROG is

```
push line no.
STL (1)        save source line number
SLDC06         push global base (BP) register number
LPR            push global base pointer
STL (2)        save old global base
SLDC06         BP register number for store
SLDC05         local base (MP) register number
LPR            push MP
SPR            store into BP.
```

The markstack was completely built by the Microengine instruction CXL which invoked the sequential program, so the code here does not do any markstack construction. The source line number (in the sequential program) and global base (pointer to the permanent variables record of the host process) are saved, as for most other routine entries. Finally, the sequential program's global base is established. No action comparable to setting the concurrent/sequential switch in the kernel is taken since the kernel has not yet been designed.

ENTERPROC is the first instruction of a process ENTRY routine, and it follows CALLSYS in execution. In the virtual machine, it checks for possible stack collision, finishes construction of the markstack, establishes the process global base, and zeroes the JOB word for the process in the kernel to indicate the execution of concurrent (supervisor) code, not sequential (user) code. Since the concurrent/sequential interface mechanism (section 3.4) is so different, the code which emulates ENTERPROC is not quite what one would expect. On the Microengine, process ENTRY routines are not called directly from sequential programs, as they are on the virtual machine. Instead, they are invoked from the interface routine (see figure 38).

The code which MEPASS6 generates is affected by the configuration of certain pointers immediately after a process ENTRY routine is invoked (see figure 39). Before the host process invokes the sequential program the global base register points to the record containing the permanent variables and access rights of the process. Microengine code equivalent to CALLPROG starts the program, and code equivalent to ENTERPROG stores the global base in the second local variable as described above. The global base register is then adjusted to point to the program's global storage area-- the same as the local area at the time the program is started. During execution the local base will change as routines are called and return, but the global base remains fixed. The companion version of Sequential Pascal apparently does not allow nested routine definitions. The global base register is not altered when the interface routine is invoked by code emulating CALLSYS, nor is it altered by the interface routine itself. Even by the time code equivalent to ENTERPROC is about to execute, the global base still points to the global variable area in the sequential program. That code saves the source line number and global base pointer in its own local space, and re-establishes the process's global base pointer by fetching the second word in the (sequential program) global space and storing it into the global base register.

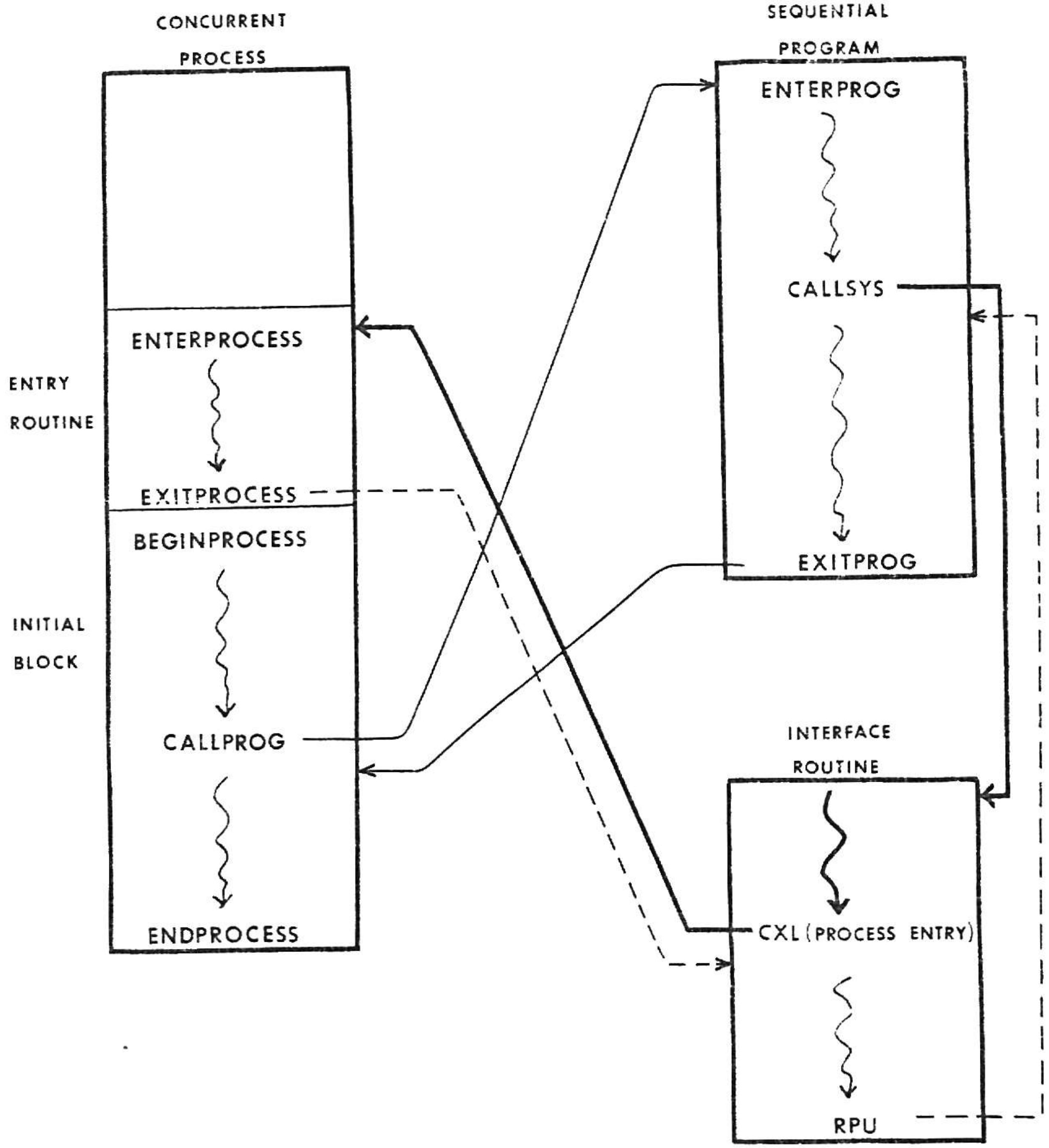


FIGURE 38. Flow control during the life of a process which uses an ENTRY routine, on the Microengine. Wavy lines represent sequential execution of instructions; narrow lines represent sequential program invocation and return; wide lines are the path taken to invoke the ENTRY routine; and dashed lines represent the return from the ENTRY routine. All of the code is Microengine machine code. Virtual instruction <u>names</u> have been used here only to indicate more clearly the function of the relevant code.

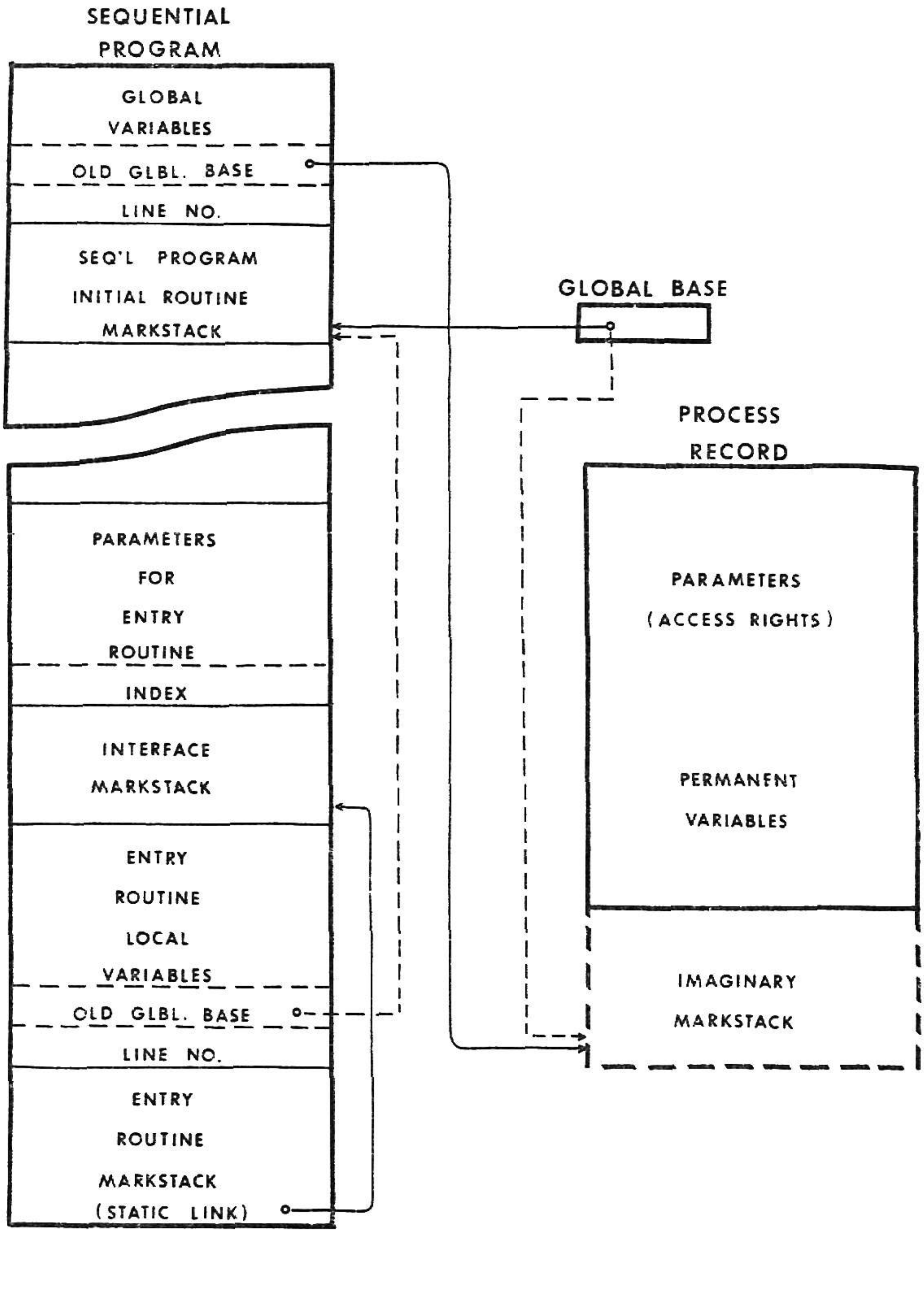


FIGURE 39. Configuration of global base pointers and process ENTRY routine's static link immediately after the ENTRY routine has been invoked. Solid pointers exist before the Microengine code comparable to ENTERPROC has executed. Dashed pointers exist after that code executes.

The Microengine code to do this is:

```
        push line no.
        STL (1)      save source line number
        SLDC06       BP register number
        LPR          push global base pointer
        STL (2)      save copy of global base
        SLDC06       BP register number
        SLDO02       push process global base saved by ENTERPROG code
        SPR          store into global base register.
```

The mechanism for indicating the return to supervisor code has not yet been determined. The compiler calculates the ENTRY routine's DATASIZE and POPLENGTH as the length of its declared local variables plus two words to store the global base (of the sequential program) and source line number. POPLENGTH does not include the size of the parameters since they are local to the interface markstack, not the ENTRY routine markstack (see figure 39). Also, POPLENGTH does not include an extra word to account for the component variable address "parameter" being on the stack, unlike ENTERCLASS and ENTERMON code. The component variable address never appears on the stack here since it is fetched out of the global variable space of the sequential program.

5.2.4.7 END and EXIT Instructions

Eight virtual instructions terminate routines by restoring the calling routine's environment. All of these use an RPU instruction on the Microengine. RPU (Return from Procedure--User) cleans up the stack in order to restore the stack to the condition it was in before the routine was called. It has one operand which is the number of words (not including the markstack) which must be popped in order to restore the calling routine's stacktop. The instruction automatically restores the caller's stack pointer (based on the operand) and local base pointer, and causes a jump to the return address in the caller's code. The instruction does not affect the global base pointer.

The virtual instruction EXIT terminates routines which are neither ENTRY nor initial routines. It performs the same function as RPU on the Microengine, except that it also restores the caller's global base pointer. Restoration of the global base is superfluous for non-ENTRY and non-initial routines since the global base is not changed by ENTER (section 5.2.4.6) and any system component routines called by the routine restore the caller's BP register when they return. The equivalent Microengine code is simply

RPU (POPLENGTH DIV WORDLENGTH),

where POPLENGTH is the value calculated when code for ENTER was generated at the start of the routine.

ENDCLASS and EXITCLASS terminate the initial and ENTRY routines of a class. Since the corresponding prologue instructions, BEGINCLASS and ENDCLASS, change the global base register, the caller's global base pointer must be restored before returning. The equivalent Microengine code is:

```
SLDC06       push global base register number
SLDL02       push caller's global base pointer
SPR          restore caller's global base
RPU (poplength div wordlength)   return to calling routine.
```

ENDMON and EXITMON terminate monitor routines and are analogous to ENDCLASS and ENTERCLASS. They perform the same function, except that the monitor instructions call on the kernel to perform a gate exit before restoring the caller's global base and returning. So, the equivalent Microengine code consists of a kernel call followed by the same code which was substituted for ENDCLASS and EXITCLASS.

EXITPROC terminates process ENTRY routines. It restores the caller's (sequential program's) global base and sets the JOB switch in the kernel to indicate the execution of sequential code. The Microengine code for this is the same as for ENDCLASS plus some, as yet undetermined, code to indicate the return to user code.

ENDPROC terminates the initial routine of processes, and is merely a return to the kernel to make the process "disappear". The Microengine code is just an RPU instruction.

EXITPROG is the last virtual instruction of a sequential program's initial statement, and in general terms, it restores the environment of the host process before the sequential program was invoked. Since it manipulates several locations in the kernel, the equivalent Microengine code has not been finalized. The code will, however, end with an RPU instruction.

Every time one of the instruction sequences discussed in this section is generated, the compiler enters the routine's STACKLENGTH into STACKTABLE. This provides a record of the routine's stack requirements for calling routines to use in calculating their own stack requirements (see section 4.2). AFTERBEGIN is set FALSE so that NEWLINE instructions for declaration source lines will not appear in the final code. If the location counter is not on a word boundary a NOP (No OPeration) is generated to force it to the required value. Alignment is required for the next routine's EXIT-IC field (see figure 7).

5.2.4.8 PUSHLABEL (Interface Generation)

The action of PUSHLABEL in the virtual machine is quite simple-- it merely pushes the address of a process ENTRY routine onto the stack. PUSHLABEL instuctions are used as part of the sequential program invocation sequence to build the jump table shown in figure 16. The jump table maps prefix routine indices to process ENTRY routine addresses during the execution of CALLSYS instructions (sections 3.4.1 and 5.2.4.4). The concurent/sequential interface mechanism for the Microengine is entirely different.

A skeletal concurrent process and interface definition are shown in figure 40. The process _can_ provide as many as six operating system services (P1 through P6) to sequential programs hosted by it. However, the code loaded into the variable C, and invoked as SEQLPGM, will only be offered three of these (P4, P6, and P3) for its use. The sequential program is invoked in line 16, and it is here that the run-time jump table is built to enforce access restrictions. When MEPASS6 encounters a PUSHLABEL instruction, it removes from the intermediate code the instructions which build the jump table, and generates a new code segment called the interface segment. The interface segment consists of only one routine (figure 17) and is placed in file number four (figure 26). The structure of the segment is known by MEPASS6, so the number of process ENTRY routines in the interface completely defines it. This permits the pass to generate a Microengine code segment which is extremely close to its final form. Pass seven will pack the code into machine words, pad it out to an integral multiple of the size of a disk block (512 bytes), and append it to the concurrent program's code segment.

The occurrence of a PUSHLABEL operator informs MEPASS6 that it has just found the start of an interface. The routine GEN INTERFACE reads through the incoming interface and generates the interface segment. The general machine-code form of the interface segment is shown in figure 41. Before emitting any code, the routine turns on the GENNINGINTFAC switch so that the routines which usually send generated code to the concurrent code file will start putting code into the interface segment file (see section 5.2.2). It also fetches

```
0001 TYPE PRC = PROCESS;
0002      TYPE CODE = ARRAY[1..1000] OF INTEGER;
0003      VAR PERM1, PERM2, PERM3: INTEGER;     CODEVAR: CODE;
0004
0005      PROCEDURE ENTRY P1;   BEGIN   END;
0006      PROCEDURE ENTRY P2;   BEGIN   END;
0007      PROCEDURE ENTRY P3;   BEGIN   END;
0008      PROCEDURE ENTRY P4;   BEGIN   END;
0009      PROCEDURE ENTRY P5;   BEGIN   END;
0010      PROCEDURE ENTRY P6;   BEGIN   END;
0011
0012      PROGRAM SEQLPGM(X, Y, Z: INTEGER; C: CODE);
0013      ENTRY P4, P6, P3;
0014
0015      BEGIN
0016      SEQLPGM(100, 200, 300, CODEVAR);
0017      END;
0018
0019 VAR    PRCV: PRC;
0020
0021 BEGIN
0022 INIT PRCV;
0023 END.
```

```
5555555555555555555555555555555555555555555555555555555555555555555555
5                     MEPASCAL INTERMEDIATE CODE PASS 5                 5
5555555555555555555555555555555555555555555555555555555555555555555555
   JUMP(1)   LNGCONST(16,HEX-VAL:0000,0000,0000,0000,0000,0000,
   0000,0000)

LINE    5   ENTER(3,3,0,0,0)   RETURN(3)
LINE    6   ENTER(3,4,0,0,0)   RETURN(3)
LINE    7   ENTER(3,5,0,0,0)   RETURN(3)
LINE    8   ENTER(3,6,0,0,0)   RETURN(3)
LINE    9   ENTER(3,7,0,0,0)   RETURN(3)
LINE   10   ENTER(3,8,0,0,0)   RETURN(3)

LINE   15   ENTER(6,2,0,2006,0)
LINE   16   PUSHLABL(5)   PUSHLABL(8)   PUSHLABL(6)
           PUSHCNST(100)   PUSHCNST(200)   PUSHCNST(300)
           PUSHADDR(6,-2006)   CALLPROG   POP(6)
LINE   17   RETURN(6)

LINE   21   ENTER(6,1,0,2,0)
LINE   22   PUSHADDR(6,-2)   DUP1TOS   INIT(6,2,0,2006)
LINE   23   RETURN(6)   EOM(2)
```

FIGURE 40. A concurrent process which invokes a sequential program, and the intermediate code produced by MEPASS5. The second operand of ENTER is the routine number.

```
  @ last word
min. case index
max. case index
      |
      |
      |
RTNS words of
CASE statement
   offsets
      |
      |
      |
   exit-ic
 datasize (0)
     SLDL01                push the prefix index parameter
     XJP (1)               CASEJUMP--case offsets start in word 1
     NOP
     UJPL (out)            jump if parameter out of range
     CXL (129, label 1)    call process entry routine
     UJPL (out)            jump out of CASE statement
     CXL (129, label 2)
     UJPL (out)
         .
         .
         .
         .

     CXL (129, label RTNS)
     UJPL (out)
out: NOP                   no operation, for word alignment
     RPU (1)               return to caller (sequential program)
  @ datasize
  1, segment id
```

FIGURE 41. General layout of an interface segment.

the size of (number of routine labels in) this interface from the array IFSEGSIZE which was loaded in procedure BEGINPASS from the PASSLINK record in the heap. The previous pass counts the number of interfaces in the concurrent program and the size of each one (see section 5.1). For the rest of this discussion the number of routines in the interface will be denoted "RTNS". The sizes (in bytes) of the various items in figure 41 are:

```
 1) @ last word-- 2
 2) min. case index-- 2
 3) max. case index-- 2
 4) offsets-- 2*RTNS
 5) EXIT-IC-- 2
 6) DATASIZE-- 2
 7) pre-case code:
        SLDL01, XJP, NOP, and UJPL-- 7
 8) code for each case:
        CXL and UJPL-- 6
 9) post-case code:
        NOP and RPU-- 3
10) @ DATASIZE-- 2
11) value 1-- 1
12) segment id.-- 1.
```

The first word of a Microengine code segment points to the last word in the segment (see section 3.2.2). For an interface segment, that pointer value can be calculated as the number of words in the segment less one word. Using the list above, that is

```
( (2 + 2 + 2 + 2*RTNS + 2 + 2 + 7 + 6*RTNS +
    3 + 2 + 1 + 1) DIV 2 "bytes per word") - 1,
```

or more concisely,

```
(8*RTNS + 22) DIV 2.
```

GEN INTERFACE puts that value in the interface file. It then starts building the constant pool which will only contain information required for the XJP instruction: the minimum and maximum case indices and the offset values for each case. The sequential program numbers its prefix routines consecutively from 1 to RTNS, so 1 and RTNS are emitted as the index limits. For each case, the distance from the instruction after the XJP (NOP) to its code is its offset. The case offsets are generated next. The offset for any case is four bytes (NOP and UJPL after XJP) plus the size of the cases between the UJPL and itself. Thus, the first four cases have offsets 4, 10, 16, 22. If the cases are numbered from 0 to RTNS-1, a case's number (let it be KASE) can be used to calculate its offset:

KASE * 6 "bytes per case" + 4.

The next field in the output file is the interface routine's EXIT-IC value, a segment-relative byte pointer to the epilogue code--the RPU instruction. By a calculation similar to the one above, the pointer value is

RTNS * 8 + 18.

The size of the stack space required by the routine for local variables, the routine's DATASIZE value, must be generated next. Interface routines have no local variables, so the value is always zero. The next section of the segment is the code itself. The first instruction (SLDL01--Short LoaD Local word 1) will, at run time, push onto the stack the interface (prefix) index number of the process ENTRY routine being accessed by the sequential program. The program pushed the index while executing Microengine code equivalent to CALLSYS (section 5.2.4.4). XJP (case jump) is the next instruction. It uses the top-of-stack index to select an offset from the case table which always starts in word location 1 of the segment (location of the minimum index word). That offset is added to the IPC to jump to the corresponding case code. A no-operation instruction comes next, although it might prove to be superfluous. It is generated only in imitation of the way the UCSD Pascal compiler handles case jumps. If the case index is not between the minimum and maximum values the next instruction (UJPL) jumps around the code for all cases. Since the cases all have six bytes of code, the jump distance is 6*RTNS. The code for all the cases follows, but first the compiler must change the order of the process ENTRY routine labels. In line 13 of the source code in figure 40 the accessible ENTRY routines are given in the order P4, P6, P3. Their corresponding routine labels are 6, 8, and 5, so the prefix routine index numbers (which the sequential program uses to call the routines) 1, 2, and 3 must map to concurrent routines 6, 8, and 5, respectively. By the time the labels reach pass six, however, their order has been reversed (figure 40, line 16 of the intermediate code). The reverse ordering works for building the jump table for the virtual machine, but it is the reverse of what is needed for the Microengine. GEN_INTERFACE removes the PUSHLABEL instructions fom the code stream and pushes the label numbers onto its own heap stack. The labels are popped off as code is generated, so that, using the example in figure 40, when the sequential program provides an index parameter of 1, concurrent routine 6 (P4) is invoked. The code for each case consists of two instructions-CXL (Call eXternal, Local routine) and UJPL. The CXL operands are the segment number of the concurrent segment (always 1), and the routine label of the process ENTRY routine to be invoked when the case is executed. The jump instruction skips around the other cases when the ENTRY routine returns. Its operand is the number of bytes of code for the cases which follow. At compile time, this is the size of the cases for which code has yet to be generated. For the interface in figure 40 the case code would be

```
CXL (129, 6)
UJPL (12)
CXL (129, 8)
UJPL (6)
CXL (129, 5)
UJPL (0).
```

A NOP is generated next for word alignment. Note that this is necessary only because of the NOP which follows the case jump operator. The procedure dictionary follows, and it cnsists of just a single entry since there is only one routine in the segment. The value to be entered is the word address, relative to the start of the segment, of the routine's EXIT-IC field. From figure 41 it can be seen that the value is RTNS+3. The last word to be generated contains two one-byte fields. The high-order byte is the number of routines in the segment, and the low-order byte is the segment number for this code segment. Once they have been loaded into main memory, all interface segments will have a segment number of 129 (see section 3.4.1), but the compiler numbers them consecutively after the concurrent segment number to help identify the segments within the code file. Finally, the switch GENNINGINTFAC is turned off so that output goes into the concurrent intermediate code file once again.

CONCLUSION

6.1 RESULTS AND OBSERVATIONS

Although a good deal of work remains to be done in order to produce a finished compiler, the work which has been completed so far is a major contribution toward that goal. There have been two major results of this project. First, modification of pass six is nearly complete. The structure and intended operation of the pass have been documented in this report, so completing the changes should not be difficult. Second, and more importantly, this report provides a substantial base of knowledge regarding the Concurrent Pascal virtual machine and Western Digital Pascal Microengine, and how the architecture of the former can be mapped to the latter. The software tools described in Appendix A are less significant results, but should be of great help to future workers, as they were to the author. The file transfer program will be useful-- necessary, in fact-- even after the development has been completed and the MEPASCAL compiler is used for production work.

The major observation during this project has been that small architectural differences in target stack machines can have profound consequences. The two target machines _seem_ to be similar. In some respects they are, but in others they are not; and those differences turned out to be quite significant. These ramifications impact almost every pass of a multipass compiler. Thus, porting a Concurrent Pascal compiler to a new stack machine involves a substantial amount of effort. The inability of the Microengine to call a routine whose number is on the stack forced the design of a completely new mechanism for the concurrent/sequential program interface, and a host of changes to the compiler. The position of the markstack relative to parameters and local variables is different in each machine and that forced a change in the way pass six calculates displacements. The differences in the location of tables for CASE statements was also a source of major changes to pass six. The ENTER and BEGIN virtual instructions have no analogs on the Microengine, and finding equivalent code sequences for them was one of the hardest parts of the project, second only to designing an interface mechanism. In short, the virtual machine instruction set has been specialized for Concurrent Pascal programs, but the Microengine has a more general-purpose instruction set, and it is not easy to simulate one with the other.

By month, the time spent on the project was:

October-December, 1979; 60 hours spent studying preliminary documents for the Microengine system;

February, 1980; 60 hours spent studying PDP-11 assembly code for interpreter, kernel, and low-level interaction with hardware;

March, 1980; 75 hours spent installing the Microengine, studying accompanying documents, gaining familiarity with Microengine system, and bringing up MCPASCAL on the 8/32;

April, 1980; 75 hours spent investigating Microengine architecture and working with the system;

May, 1980; 75 hours spent investigating virtual machine architecture and writing MNEM;

June, 1980; 75 hours spent studying the virtual machine architecture. integrating MNEM into MCPASCAL. Started making instruction modifications in the pass six code;

July, 1980; 75 hours spent studying the Microengine code file format, the architectures of the two target machines, and asynchronous I/O hardware on the Microengine;

August, 1980; 100 hours spent making more instruction modifications, bringing up MEPASCAL on the Interdata 8/32, writing the file transfer program (Appendix A), and designing the concurrent/sequential interface;

September, 1980; 75 hours spent making instruction modifications and writing the interface segment generator;

October, 1980; 75 hours writing and integrating MNEM into MEPASCAL, and testing the file transfer program;

November, 1980; 40 hours spent writing this report and reevaluating the code modifications;

December, 1980; 200 hours spent writing this report and reevaluating the code modifications;

January, 1981; 100 hours spent writing this report and reevaluating the code modifications.

The times shown for the months March through October are probably very conservative.

Of the 1085 hours of effort, 420 hours (38.7%) were spent in learning the stack machine semantics of both machines. If the future work (described below) to complete the porting takes two months of effort, then the learning effort is only 29.6% of the total. Discounting the learning effort, six months of effort would be required to port an operating system. We feel this is a minimal porting effort for an operating system and all of its associated Pascal utilities and applications programs.

6.2 FUTURE WORK

Although the major modifications to the compiler have already been made, work remains to produce a working compiler.

STACKTABLE should be left in the heap after pass six and used to re-implement the heap-stack collision detection mechanism built into the NEW, NEWINIT, BEGIN, and ENTER virtual instructions.

Integer and real formats will have to be changed. Since it runs on an Interdata 8/32, MEPASCAL generates integer and real constants in 8/32 format. Microengine integers have their low-order bits in the low address byte. The difference will probably only affect instructions with word operands-- LDCI (LoaD Constant Integer), UJPL (Unconditional JumP Long), and FJPL (False JumP Long)-- such that the operand bytes will need to be swapped. This could be done in pass seven. Real constants will have to be changed to the four-byte PDP-11 format. This would be best done at the point in the compiler (prior to pass six) where they are generated. Real constants are stored in the constant pool, and it would be impossible at any other point to distinguish a real constant from a string of the same length. Since Microengine reals are only half as long as those generated by the original compiler, the amount of space allocated for real variables must be adjusted in pass four.

Displacement calculations for variables will also have to be changed in pass four. The reason for the change involves the placement of variables relative to the markstack. Figure 42 shows a record variable on the virtual machine and the Microengine. Since pass six calculates variable displacements by negating the displacement it receives from pass five, the wrong "end" of multiword variables will be referenced.

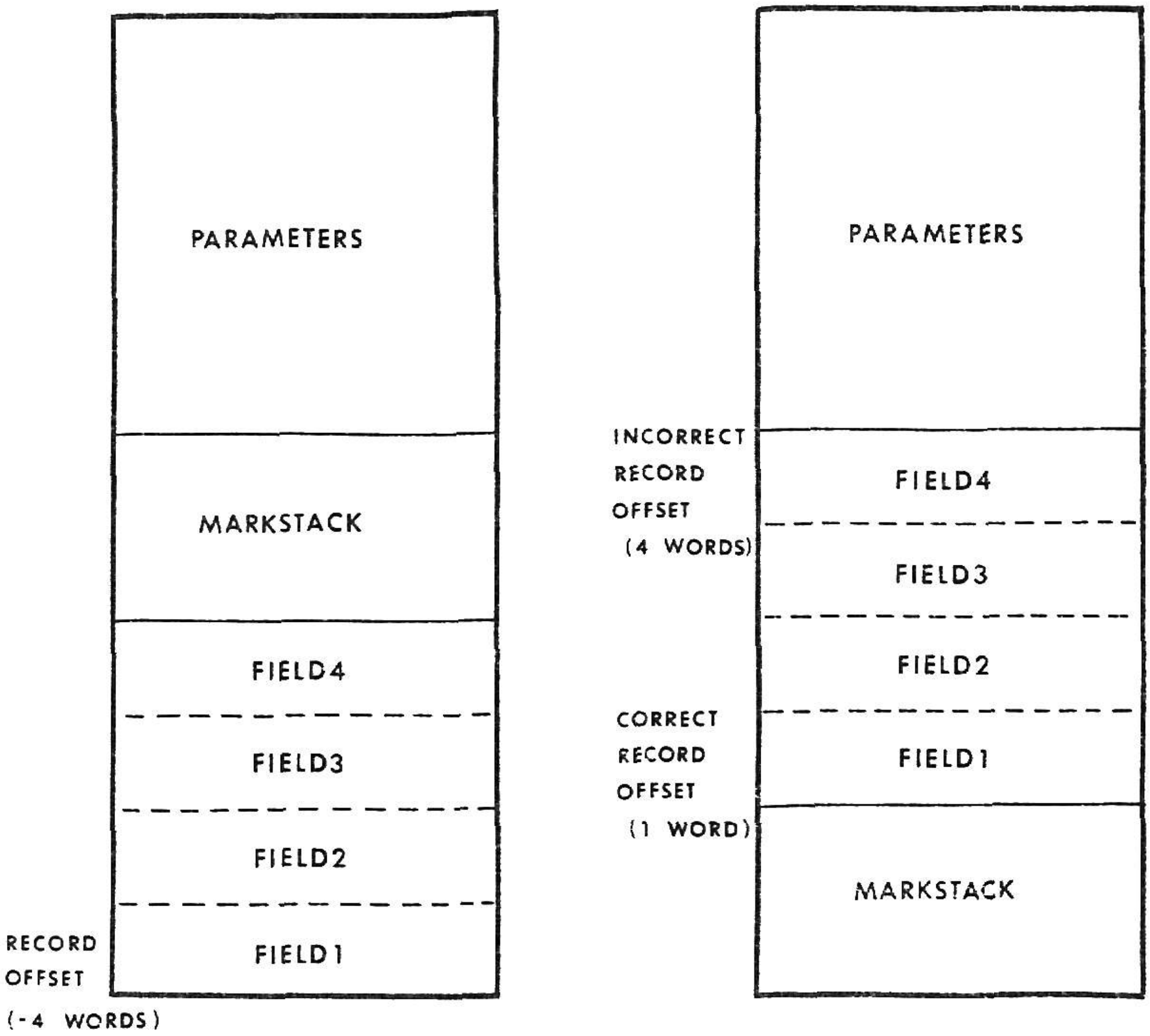


FIGURE 42. Record offsets for the Concurrent Pascal virtual machine (left) and the Microengine (right). Currently the MEPASCAL compiler calculates incorrect displacement values, as shown here.

Space must be reserved in the permanent variable space of monitors for the monitor gate (semaphore) address.

A change must be made to some pass prior to the code generator to convert variant record tags from bit numbers to bit values.

The kernel call mechanism must be designed. The mechanism could be calls to kernel routines which are well-known to the compiler.

The sequential compiler must be changed to pop the parameters from the stack after a call to the interface routine. The number of parameters pushed on the stack before calling the interface routine varies with the process ENTRY routine being invoked, so the RPU instruction which terminates the interface routine cannot pop the parameters.

Some optimizations of the final code are possible. First, the compiler always generates "long" jump instructions, regardless of the size of the distance operand. A smarter compiler would use long jumps only as required, but this would also complicate address calculations considerably. Second, only LDL (LoaD Local word) and LDO (LoaD glObal word) are generated to push onto the stack words from the local and global variable spaces. Shorter, faster instructions exist for pushing the first sixteen words in each space. Finally, it seems that the local word reserved for the old global base might be done away with. The local or global base register is saved as the static link, depending on the particular instruction used to call a routine. Perhaps the static link field of the markstack could be used to store the old global base by judicious use of the various routine-calling instructions. Before returning to the caller, the global base could be restored by using the LSL (Load Static Link) instruction.

Pass seven will have to be written. The pass must do the following:

-- Generate the header block for the code file;

-- Insert non-code fields (DATASIZE, for example) into the concurrent segment;

-- Build the constant pool for the concurrent segment;

-- Pack the code into words;

-- Calculate jump displacements;

-- Pad to the next block boundary;

-- For each interface, pack the code and pad to a block boundary.

Since pass seven is, at this time, only a program stub, a message indicating compilation errors is always generated, even if the source code is correct.

Finally, the compiler must be tested to ensure that it generates the intended Microengine code, and that the generated code actually behaves as expected when run on the hardware.

REFERENCES

BRIN75 Brinch Hansen, P., and Deverill, R. S. PDP-11 Assembler Source Code For Concurrent Pascal Kernel, California Institute of Technology, Pasadena, California, 1975.

BRIN76 Brinch Hansen, P. "Pascal Notebook," California Institute of Technology, Pasadena, California, 1976.

BRIN77 Brinch Hansen, P. The Architecture of Concurrent Programs, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1977.

DIGI76 Digital Equipment Corporation. PDP-11 04/34/45/55 Processor Handbook, 1976.

HART76 Hartmann, A.C. "A Concurrent Pascal Compiler For Minicomputers," Doctoral Thesis, California Institute of Technology, Pasadena, California, 1976.

MICR79 The Microengine Company. WD/90 Pascal Microengine Reference Manual, 1979.

REGE78 Regents of the University of California. "UCSD (Mini-Micro Computer) Pascal Revised Version I.4b," University of California at San Diego, La Jolla, California, 1978.

REGE79 Regents of the University of California. "Architecture Changes In (UCSD Pascal) Version III.0," University of California at San Diego, La Jolla, California, 1979.

SCHM77 Schmidt, D. Pascal32 Source Code For the MCPASCAL Compiler, Department of Computer Science, Kansas State University, Manhattan, Kansas, 1977.

YOUN79a Young, R., and Wallentine, V. "The NADEX core Operating System Services," Report TR-79-11, Departmant of Computer Science, Kansas State University, Manhattan, Kansas, 1979.

YOUN79b Young, R., and Wallentine, V. "The Structure of the NADEX Operating System," Report TR-79-12, Department of Computer Science, Kansas State University, Manhattan, Kansas, 1979.

ZEPK74 Zepko, T. PDP-11 Assembler Source Code For Concurrent Pascal Interpreter, California Institute of Technology, Pasadena, California, 1974.

APPENDIX A

SOFTWARE TOOLS

INTERMEDIATE CODE MNEMONICS PROGRAMS

MNEM (MNEMonics) is a sequential Pascal32 program, adapted from Robert Young's PASST, which prints intermediate and final code mnemonics for MCPASCAL. It was written to help better understand the architecture of the Concurrent Pascal virtual machine and the operation of the MCPASCAL compiler by generating reasonably intelligible operator mnemonics instead of the integer values produced by the mechanism built into MCPASCAL.

The program is invoked by specifying pass numbers as driver options. For example, entering the command

```
MCPASCAL SRCPGM,PR:,5,7
```

will cause mnemonics to be printed for the output of passes five and seven. After each compiler pass executes, the driver program runs MNEM if the pass terminated normally and the pass was specified by the user in the list of driver options. The pass number is sent to MNEM as a program parameter so that it knows which set of mnemonic literals should be generated. Currently, MNEM will only work on the output of passes five, six, and seven, since that of the other passes was not nearly as useful to the progress of the project. If the user requests mnemonics for one of the first four passes, the driver does not call MNEM, and prints a message to that effect. This allows the compilation to proceed to its conclusion, even though the request cannot be fulfilled.

After initializing I/O buffer variables and writing a pass header, MNEM calls the routine which will actually scan the appropriate intermediate code file. Each scanning routine (PASSx) contains a structured constant table which contains the mnemonic literals to be printed for each operator and the number of operands which the operator uses. An operator read from the input file is used as an index into the table to fetch its mnemonic character string and number of arguments, which are then passed to a generalized instruction output routine. That routine prints the character string, reads the appropriate number of operands from the input file and prints them. PASST has a mechanism for printing kind, type, mode, and context mnemonics instead of the numerical values, but it was not carried over into MNEM since it did not seem to be worth the extra effort. No great inconvenience was experienced as a result. The operator formatting routine implemented that mechanism by using four parameters (KIND_ARG, TYPE_ARG, MODE_ARG, and CONTEXT_ARG). If, for example, an operator has three operands, and the second one is the addressing mode, MODE_ARG would have the value 2, and the other parameters would have the value 0. Since the mechanism is unused in MNEM, the four parameters are always 0.

Three instructions are handled by special routines when their operators are encountered. They are NEWLINE, CASEJUMP, and LONGCONSTANT. When NEWLINE is found, the current line of output is ended and a new one started in order to format the intermediate code listing in parallel with the source listing. The other two instructions have their own routines since they have a variable number of operands. Each one prints the operator mnemonic and whatever operands the instruction always has, determines the number of variable operands, and prints them.

The output line length used in the program is 70 positions, which corresponds to no physical device length. If the output is directed to a disk file and the editor is used to peruse it, a line length of 70 ensures that all of the text for a logical line will fit on one physical line of the CRT. If a longer line length is used, a long line of text will spill onto a second CRT line-- something this writer finds extremely annoying.

MEMNEM (MicroEngine MNEMonics) prints intermediate code mnemonics for code generated by the MEPASCAL compiler. It is analogous to MNEM in the way it is invoked and in its general structure. Currently, it works only on the output of passes five and six. Pass five mnemonics are generated just as in MNEM, but the output of pass six is handled quite differently. In overall terms, the program scans the concurrent segment, scans the interface segment, prints some of the values in the PASSLINK record, and prints the contents of the heap tables.

The scan of the concurrent segment is straightforward, except that jumps and new lines are handled a little differently from MNEM. Microengine jump operators (UJPL and FJPL) have only one operand-- the displacement to the destination. However, when pass six generates these instructions the displacement is unknown. The jump operand emitted by it is the <u>label</u> of the destination. It then puts the current value of the location counter into the code stream for pass seven's use in calculating the displacement. MEMNEM prints jump instructions just as any other, but then prints the location value and flags it as a non-Microengine entity. Pass six also generates NEWLINE operators which are not part of the Microengine instruction set. It emits a NEWLINE virtual operator <u>and</u> the equivalent Microengine code when it finds a NEWLINE input operator. The equivalent code consists of two instructions-- one to push the line number onto the stack, and the other to store it into the first word of the local variable space. The store into local word 1 could be used to trigger output formatting actions as in MNEM, except that the line number itself would be on the wrong output line. Rather than use a lookahead mechanism to check every "push" for a following STL(1), the virtual operator mechanism is used. When NEWLINE is found by MEMNEM nothing is printed, but the output pointer is forced to the next line so that the "push" and STL(1) will be printed there. The non-Microengine operators MESSAGE and EOM are flagged with asterisks.

After scanning the file which contains the concurrent code, MEMNEM starts taking its input from the file of interface segments. The number of segments in the file is fetched from the PASSLINK record and then the content of each segment is printed. Printing the interface segments is straightforwad, although not smooth or elegant, since they consist of so many kinds of objects besides just straight code.

Printing some of the PASSLINK fields is another straightforward, field-by-field operation. Only non-pointer fields are printed.

The heap tables are printed last. The routine DUMP_TABLE simply reads all the entries in a table and prints them. The entries for all tables except CONSTTABLE are printed as 16-bit integer values. CONSTTABLE entries are printed as byte values.

The source code for MNEM and MEMNEM follows.

```
     1 "MCPASCAL MNEM - MNEMONIC TEST OUTPUT FORMATTING"
     2
     3 "% CONCURRENT := TRUE"
     4
     5 "############
C    6 #  PREFIX  #
C    7 ############"
     8
     9 CONST NL= '(:10:)';   FF= '(:12:)';   CH= '(:13:)';   EM= '(:25:)';          *     *     *     *
    10
    11 CONST PAGELENGTH = 512 DIV 2;                                                *
    12 TYPE PAGE = ARRAY (.1..PAGELENGTH.) OF SHORTINTEGER;                         *     11    *
    13 CONST BYTES_PER_INTEGER = 2;                                                 *
    14
    15 CONST LINELENGTH = 70;                                                       *
    16 TYPE LINE = ARRAY (.1..LINELENGTH.) OF CHAR;                                 *     15    S
    17
    18 CONST IDLENGTH = 12;                                                         *
    19 TYPE IDENTIFIER = ARRAY (.1..IDLENGTH.) OF CHAR;                             *     18    S
    20
    21 TYPE FILE = 1..5;                                                            *
    22
    23 TYPE FILEKIND = (EMPTY, SCRATCH, ASCII, SEQCODE, CONCODE);                   *     *     *     *     *     *
    24
    25 TYPE FILEATTR = RECORD                                                       *
    26                   KIND: FILEKIND;                                            *     23
    27                   ADDR: INTEGER;                                             *     S
    28                   PROTECTED: BOOLEAN;                                        *     S
    29                   NOTUSED: ARRAY (.1..5.) OF INTEGER                         *     S
    30                 END;
    31
    32 TYPE IODEVICE =                                                              *
    33   (TYPEDEVICE, DISKDEVICE, TAPEDEVICE, PRINTDEVICE, CARDDEVICE);             *     *     *     *     *
    34
    35 TYPE IOOPERATION = (INPUT, OUTPUT, MOVE, CONTROL);                           *     *     *     *     *
    36
    37 TYPE IOARG = (WRITEEOF, REWIND, UPSPACE, BACKSPACE);                         *     *     *     *     *
    38
    39 TYPE IORESULT =                                                              *
    40   (COMPLETE, INTERVENTION, TRANSMISSION, FAILURE,                            *     *     *     *
    41    ENDFILE, ENDMEDIUM, STARTMEDIUM);                                         *     *     *
    42
    43 TYPE IOPARAM = RECORD                                                        *
    44                  OPERATION: IOOPERATION;                                     *     35
    45                  STATUS: IORESULT;                                           *     39
    46                  ARG: IOARG                                                  *     37
    47                END;
    48
    49 TYPE TASKKIND = (INPUTTASK, JOBTASK, OUTPUTTASK);                            *     *     *     *
    50
    51 TYPE ARGTAG =                                                                *
    52   (NILTYPE, BOOLTYPE, INTTYPE, IDTYPE, PTRTYPE);                             *     *     *     *     *
    53
    54 TYPE POINTER = @BOOLEAN;                                                     *     S
    55
    56 TYPE ARGTYPE = RECORD                                                        *
    57                  CASE TAG: ARGTAG OF                                         *     51
    58                    NILTYPE, BOOLTYPE: (BOOL: BOOLEAN);                      52    52     *     S
    59                    INTTYPE: (INT: INTEGER);                                  *     *     S
```

```
 60                    IDTYPE: (ID: IDENTIFIER);                                *     *     19
 61                    PTRTYPE: (PTR: POINTER)                                  *     *     54
 62                END;
 63
 64 CONST MAXARG = 10;                                                          *
 65 TYPE ARGLIST = ARRAY (.1..MAXARG.) OF ARGTYPE;                              *     64    56
 66
 67 TYPE ARGSEQ = (INP, OUT);                                                   *     *     *
 68
 69 TYPE PROGRESULT =                                                           *
 70   (TERMINATED, OVERFLOW, POINTERERROR, RANGEERROR, VARIANTERROR,            *     *     *     *     *
 71    HEAPLIMIT, STACKLIMIT, CODELIMIT, TIMELIMIT, CALLERROR);                 *     *     *     *     *
 72
 73 PROCEDURE READ(VAR C: CHAR);                                                *     *     S
 74 PROCEDURE WRITE(C: CHAR);                                                   *     *     S
 75
 76 PROCEDURE OPEN(F: FILE; ID: IDENTIFIER; VAR FOUND: BOOLEAN);                *     *     21    *     19    *     S
 77 PROCEDURE CLOSE(F: FILE);                                                   *     *     21
 78 PROCEDURE GET(F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);                   *     *     21    *     S     *     12
 79 PROCEDURE PUT(F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);                   *     *     21    *     S     *     12
 80 FUNCTION LENGTH(F: FILE): INTEGER;                                          *     *     21    S
 81
 82 PROCEDURE MARK(VAR TOP: INTEGER);                                           *     *     S
 83 PROCEDURE RELEASE(TOP: INTEGER);                                            *     *     S
 84
 85 PROCEDURE IDENTIFY(HEADER: LINE);                                           *     *     16
 86 PROCEDURE ACCEPT(VAR C: CHAR);                                              *     *     S
 87 PROCEDURE DISPLAY(C: CHAR);                                                 *     *     S
 88
 89 PROCEDURE READPAGE(VAR BLOCK: UNIV PAGE; VAR EOF: BOOLEAN);                 *     *     12    *     S
 90 PROCEDURE WRITEPAGE(BLOCK: UNIV PAGE; EOF: BOOLEAN);                        *     *     12    *     S
 91 PROCEDURE READLINE(VAR TEXT: UNIV LINE);                                    *     *     16
 92 PROCEDURE WRITELINE(TEXT: UNIV LINE);                                       *     *     16
 93 PROCEDURE READARG(S: ARGSEQ; VAR ARG: ARGTYPE);                             *     *     67    *     56
 94 PROCEDURE WRITEARG(S: ARGSEQ; ARG: ARGTYPE);                                *     *     67    *     56
 95
 96 PROCEDURE LOOKUP(ID: IDENTIFIER; VAR ATTR: FILEATTR;                        *     *     19    *     25
 97                                  VAR FOUND: BOOLEAN);                       *     S
 98
 99 PROCEDURE IOTRANSFER                                                        *
100   (DEVICE: IODEVICE; VAR PARAM: IOPARAM; VAR BLOCK: UNIV PAGE);             *     32    *     43    *     12
101
102 PROCEDURE IOMOVE(DEVICE: IODEVICE; VAR PARAM: IOPARAM);                     *     *     32    *     43
103
104 FUNCTION TASK: TASKKIND;                                                    *     49
105
106 PROCEDURE RUN(ID: IDENTIFIER; VAR PARAM: ARGLIST;                           *     *     19    *     65
107               VAR LINE: INTEGER; VAR RESULT: PROGRESULT);                   *     S     *     69
108
109
110 PROGRAM MNEM(VAR PARAM: ARGLIST);                                           *     *     65
111
112 "SIMPLE TYPES AND CONSTANTS"
113
114 CONST LINE_LENGTH = 70;      "OUTPUT DEV LINE LENGTH"                       *
115
116 TYPE CHAR8 = ARRAY [1..8] OF CHAR;                                          *     S
117
118 CONST
119    MIN KIND = 0;   MAX_KIND = 0;                                            *     *
```

```
120    MIN_TYPE = 0;   MAX_TYPE = 0;                                          *     *
121    MIN_MODE = 0;   MAX_MODE = 0;                                          *     *
122    MIN_CONTEXT = 0;   MAX_CONTEXT = 0;                                    *     *
123
124
125 "VARIABLES"
126
127 VAR WORDS_IN: INTEGER;      "WORD IN IFL BUFFER"                          *     S
128     IN_FILE: INTEGER;        "INPUT FILE"                                 *     S
129     PAGES_IN: INTEGER;      "CURRENT PAGE NUMBER"                         *     S
130     PAGE_IN: PAGE;          "INPUT BUFFER"                                *    12
131     PASS_NO: INTEGER;       "COMPILER PASS NUMBER"                        *     S
132     OUT_COL_PTR: INTEGER;  "POSN ON OUTPUT LINE FOR BUF"                  *     S
133     OUT_BUF_PTR: INTEGER;  "POSN IN BUFFER"                               *     S
134     OUT_BUF: LINE;          "OPERATOR OUTPUT BUFFER"                      *   107
135     FIRST_COL: INTEGER;     "FIRST COL TO USE FOR NON-NEWLINE"            *     S
136     MIN_COL_SEP: INTEGER;  "MIN BLANKS BETWEEN OPS"                       *     S
137     COL_SEP: INTEGER;       "SEP BETWEEN COL BOUNDARIES"                  *     S
138     EOM_OP: SHORTINTEGER;        "EOM OPERATOR"                           *    12
139     LINE_OP: SHORTINTEGER;       "LINE NUMBER OPERATOR"                   *    12
140     LCONST_OP: SHORTINTEGER;     "LCONST OPERATOR"                        *    12
141     OP: SHORTINTEGER;            "CURRENT INPUT OPERATOR"                 *    12
142     CODELENGTH: SHORTINTEGER;                                             *    12
143     CODE_READ_SO_FAR: INTEGER;                                            *     S
144     I: INTEGER;                                                           *     S
145
146
147 "INPUT FILE SUPPORT ROUTINES"
148
149 PROCEDURE READ_IFL (VAR ARG: SHORTINTEGER);                               *     *    12
150 BEGIN
151   IF WORDS_IN = PAGELENGTH THEN BEGIN                                    127    11
152     GET(IN_FILE,PAGES_IN,PAGE_IN);                                        78   128   129   130
153     PAGES_IN:=PAGES_IN+1; WORDS_IN:=0;                                   129   129   127
154   END;
155   WORDS_IN:=WORDS IN+1;                                                  127   127
156   ARG:=PAGE_IN[WORDS IN];                                                149   130   127
157   CODE_READ_SO_FAR := CODE_READ_SO_FAR + BYTES_PER INTEGER;              143   143    13
158 END;
159
160 "BUFFER OUTPUT ROUTINES"
161
162 PROCEDURE NEXT_COL;                                                       *
163 VAR I, J: INTEGER;                                                        *     *     S
164 BEGIN  "PAD TO START OF NEXT OP COLUMN"
165   I:=FIRST_COL;                                                          163   135
166   I:=((OUT_COL_PTR+MIN_COL_SEP-FIRST_COL-1) DIV COL SEP + 1)*COL SEP;   163   132   136   135   137   137
167   IF I<0 THEN I:=0;                                                      163   163
168   I:=I+FIRST_COL;                                                        163   163   135
169   IF I+OUT_BUF_PTR-1 > LINE_LENGTH THEN BEGIN "WON'T FIT"                163   133   114
170     WRITE(NL); I:=FIRST_COL; OUT_COL PTR:=1;                              74     9   163   135   132
171   END;
172   FOR J:=OUT_COL_PTR TO I-1 DO WRITE(' ');                               163   132   163    74
173   OUT_COL_PTR:=I;                                                        132   163
174 END;
175
176 PROCEDURE WRITE_BUF (ALIGN: BOOLEAN);                                     *     *     S
177 VAR I: INTEGER;                                                           *     S
178 BEGIN
179   IF ALIGN THEN NEXT_COL;                                                176   162
```

```
  180   FOR I:=1 TO OUT_BUF_PTR-1 DO WRITE(OUT_BUF[I]);             177   133    74   134   177
  181   OUT_COL_PTR:=OUT_COL_PTR+OUT_BUF_PTR-1;                     132   132   133
  182   OUT_BUF_PTR:=1;                                             133
  183 END;
  184
  185 PROCEDURE WRITE_CHAR (C: CHAR);                                 *     *     S
  186 BEGIN
  187   IF OUT_BUF_PTR>LINE_LENGTH THEN                             133   114
  188     WRITE_BUF(TRUE);                                          176     *
  189   OUT_BUF[OUT_BUF_PTR]:=C;                                    134   133   185
  190   OUT_BUF_PTR:=OUT_BUF_PTR+1;                                 133   133
  191 END;
  192
  193 "BUFFER FORMATTING ROUTINES"
  194
  195 PROCEDURE WRITE_CHAR8 (TEXT: CHAR8);                            *     *   116
  196 VAR I: INTEGER;                                                 *     S
  197 BEGIN
  198      FOR I := 1 TO 8 DO                                       196
  199           IF (TEXT[I] <> ' ')                                 195   196
  200           THEN WRITE_CHAR (TEXT[I])                           185   195   196
  201 END;
  202
  203 PROCEDURE WRITE_HEX_CHAR (VAL: SHORTINTEGER);                   *     *    12
  204 BEGIN
  205   IF VAL > 9 THEN WRITE_CHAR(CHR(VAL-10+ORD('A')))            203   185     *   203     *
  206     ELSE WRITE_CHAR(CHR(VAL+ORD('0')));                       185   205   203   205
  207 END;
  208
  209 PROCEDURE WRITE_HEX (VAL: SHORTINTEGER; N: SHORTINTEGER);       *     *    12     *    12
  210 TYPE TAGS = (TAG1, TAG2);                                       *     *     *
  211 VAR I: SHORTINTEGER;                                            *    12
  212     X: RECORD CASE TAGS OF                                      *     *
  213       TAG1: (INT: SHORTINTEGER);                                *    59    12
  214       TAG2: (BYTES: ARRAY [1..2] OF BYTE);                      *     *     *
  215     END;
  216 BEGIN
  217   X.INT:=VAL;                                                 212    59   209
  218   FOR I:=3-N TO 2 DO BEGIN                                    211   209
  219     WRITE_HEX_CHAR(X.BYTES[I] DIV 16);                        203   212   214   211
  220     WRITE_HEX_CHAR(X.BYTES[I] MOD 16);                        203   212   214   211
  221   END;
  222 END;
  223
  224 PROCEDURE WRITE_INT (VAL: SHORTINTEGER);                        *     *    12
  225 VAR A: ARRAY [1..10] OF CHAR;                                   *     S
  226     I, J, REM: INTEGER;                                         *     *     *     S
  227 BEGIN
  228   REM:=VAL; I:=1;                                             226   224   226
  229   REPEAT
  230     A[I]:=CHR(ABS(REM MOD 10)+ORD('0'));                      225   226   205     *   226   205
  231     I:=I+1; REM:=REM DIV 10;                                  226   226   226   226
  232   UNTIL REM = 0;                                              226
  233   IF VAL<0 THEN WRITE_CHAR('-');                              224   185
  234   FOR J:=I-1 DOWNTO 1 DO WRITE_CHAR(A[J]);                    226   226   185   225   226
  235 END;
  236
  237 PROCEDURE WRITE_INTR (VAL: INTEGER; N: INTEGER);                *     *     S     *     S
  238 "PRINTS THE VALUE OF 'VAL' IN A FIELD OF 'N' POSITIONS.
C 239  VALUE WILL BE RIGHT JUSTIFIED IN THE FIELD."
```

```
  240 VAR A: ARRAY [1..10] OF CHAR;                                             *     S
  241     I, J, REM: INTEGER;                                                   *     *     *     S
  242 BEGIN
  243   REM:=VAL; I:=1;                                                         241   237   241
  244   REPEAT
  245     A[I]:=CHR(ABS(REM MOD 10)+ORD('0'));                                  240   241   205   230   241   205
  246     I:=I+1; REM:=REM DIV 10;                                              241   241   241   241
  247   UNTIL REM = 0;                                                          241
  248   IF VAL < 0 THEN BEGIN                                                   237
  249     A[I]:='-'; I:=I+1;                                                    240   241   241   241
  250   END;
  251   FOR J:=I TO N DO WRITE_CHAR(' ');                                       241   241   237   185
  252   FOR J:=I-1 DOWNTO 1 DO WRITE_CHAR(A[J]);                                241   241   185   240   241
  253 END;
  254
  255
  256 PROCEDURE INDEXERROR (TEXT:CHAR8; BADINDEX: SHORTINTEGER);                *     *     116   *     12
  257 "IF THIS PROCEDURE IS INVOKED IT PROBABLY MEANS SOME OPERATOR
C 258  WHICH WAS ENCOUNTERED EARLIER HAS THE WRONG 'NUM_ARGS' (NO. OF
C 259  OPERANDS) VALUE ASSIGNED TO IT IN THE OPERATOR TABLE."
  260 VAR I: INTEGER;                                                           *     S
  261 BEGIN
  262 WRITE_CHAR8 ('%%%%%%%%');                                                 195
  263 WRITE_CHAR8 ('INDEX   ');                                                 195
  264 WRITE_CHAR (' ');                                                         185
  265 WRITE_CHAR8 ('ERROR   ');                                                 195
  266 WRITE_CHAR8 ('%%%%%%%%');                                                 195
  267 WRITE_CHAR8 (TEXT);                                                       195   256
  268 WRITE_CHAR ('=');                                                         185
  269 WRITE_CHAR (' ');                                                         185
  270 WRITE_INT (BADINDEX);                                                     224   256
  271 WRITE_CHAR (' ');                                                         185
  272 WRITE_CHAR (' ');                                                         185
  273 END;
  274
  275
  276 "GENERAL OPERATOR FORMATTING ROUTINE"
  277
  278 PROCEDURE WRITE_OP (OP_NAME: CHAR8; NUM_ARGS, KIND_ARG, TYPE_ARG,         *     *     116   *     *     *
  279                     MODE_ARG, CONTEXT_ARG: SHORTINTEGER);                 *     *     12
  280 VAR ARG_NO, ARG_VAL, I: SHORTINTEGER;                                     *     *     *     12
  281 BEGIN
  282 WRITE_CHAR8 (OP_NAME);                                                    195   278
  283 FOR ARG_NO := 1 TO NUM_ARGS DO                                            280   278
  284     BEGIN
  285     IF ARG_NO = 1                                                         280
  286     THEN WRITE_CHAR (  '('  )                                             185
  287     ELSE WRITE_CHAR (  ','  );                                            185
  288     READ_IFL (ARG_VAL);                                                   149   280
  289     IF
  290          ARG_NO = KIND_ARG THEN                                           280   278
  291              BEGIN
  292              IF (ARG_VAL<MIN_KIND) OR (ARG_VAL>MAX_KIND)                  280   119   280   119
  293              THEN INDEXERROR ('BAD-KIND', ARG_VAL)                        256   280
  294              ELSE WRITE_CHAR8 ('NO_KIND_')                                195
  295              END   ELSE IF
  296          ARG_NO = TYPE_ARG THEN                                           280   278
  297              BEGIN
  298              IF (ARG_VAL<MIN TYPE) OR (ARG_VAL>MAX_TYPE)                  280   120   280   120
  299              THEN INDEXERROR ('BAD-TYPE', ARG_VAL)                        256   280
300              ELSE WRITE_CHAR8 ('NO_TYPE_')                                   195
301              END   ELSE IF
302          ARG_NO = MODE_ARG THEN                                              280   279
303              BEGIN
304              IF (ARG_VAL<MIN_MODE) OR (ARG_VAL>MAX_MODE)                     280   121   280   121
305              THEN INDEXERROR ('BAD-MODE', ARG_VAL)                           256   280
306              ELSE WRITE_CHAR8 ('NO_MODE_')                                   195
307              END   ELSE IF
308          ARG_NO = CONTEXT_ARG THEN                                           280   279
309              BEGIN
310              IF (ARG_VAL<MIN_CONTEXT) OR (ARG_VAL>MAX_CONTEXT)               280   122   280   122
311              THEN INDEXERROR ('BAD-CTXT', ARG_VAL)                           256   280
312              ELSE WRITE_CHAR8 ('NO_CNTXT')                                   195
313              END   ELSE IF
314          (ARG_VAL < -32768) OR (ARG_VAL > 32767)                             280   280
315              THEN WRITE_HEX (ARG_VAL, 2)                                     209   280
316          ELSE WRITE_INT (ARG_VAL)                                            224   280
317      END;  "OF BEGIN"
318   IF NUM_ARGS>0 THEN WRITE_CHAR(')');                                        278   185
319   WRITE_BUF(TRUE);                                                           176   188
320 END;
321
322 "SPECIAL CASE ROUTINE FOR NEW LINE OPERATOR"
323
324 PROCEDURE NEW_LINE;                                                            *
325 VAR ARG: SHORTINTEGER;                                                         *    12
326 BEGIN
327   READ_IFL(ARG);  "LINE NUMBER"                                              149   325
328   IF OUT_COL_PTR<>1 THEN WRITE(NL);                                          132    74     9
329   WRITE(NL);                                                                  74     9
330   OUT_COL_PTR:=1;                                                            132
331   WRITE_CHAR8('LINE    '); WRITE_INTR(ARG,5);                                195   237   325
332   WRITE_BUF(FALSE);  "WRITE WITH NO COL ALIGN"                               176     *
333 END;
334
335 "SPECIAL CASE ROUTINE FOR LONG CONSTANT"
336
337 PROCEDURE WRITE_LCONST;                                                        *
338 VAR LEN, I, N, ARG: SHORTINTEGER;                                              *     *     *     *    12
339 BEGIN
340   WRITE_CHAR8('LNGCONST'); WRITE_CHAR('(');                                  195   185
341   READ_IFL(LEN);                                                             149   338
342   WRITE_INT(LEN); WRITE_CHAR(',');                                           224   338   185
343   WRITE_CHAR8 ('HEX-VAL:');                                                  195
344   NEXT_COL; N:=(LEN-1) DIV 2 + 1;                                            162   338   338
345   FOR I:=1 TO N DO BEGIN                                                     338   338
346     READ IFL(ARG);                                                           149   338
347     IF OUT_COL_PTR+OUT_BUF_PTR-1+9 > LINE_LENGTH THEN BEGIN                  132   133   114
348       WRITE_BUF(FALSE); WRITE(NL); OUT_COL PTR:=1; NEXT_COL;                 176   332    74     9   132   162
349     END;
350     WRITE_HEX(ARG,2);                                                        209   338
351     IF I < N THEN WRITE_CHAR(',') ELSE WRITE_CHAR(')');                      338   338   185   185
352   END;
353   WRITE_BUF(FALSE);                                                          176   332
354 END;
355
356
357 "COMMON INITIALIZATION"
358
359 PROCEDURE INITIALIZE;                                                          *
```

```
360 BEGIN
361   WORDS_IN:=PAGELENGTH;                                 127    11
362   PAGES_IN:=1;                                          129
363   PASS_NO:=PARAM[10].INT;                               131   110    59
364   OUT_COL_PTR:=1;                                       132
365   OUT_BUF_PTR:=1;                                       133
366   FIRST_COL:=12;                                        135
367   MIN_COL_SEP:=3;                                       136
368   COL_SEP:=12;                                          137
369 END;
370
371 "WRITE PASS LISTING HEADER"
372
373 PROCEDURE WRITE_TEXT (TEXT: LINE);                        *     *   107
374 VAR I: INTEGER;                                           *     S
375 BEGIN
376   I:=1;                                                 374
377   WHILE TEXT[I] <> '$' DO BEGIN                         373   374
378     WRITE(TEXT[I]); I:=I+1;                              74   373   374   374   374
379   END;
380 END;
381
382 PROCEDURE WRITE_HEADER;                                   *
383 VAR I, J: INTEGER;                                        *     *     S
384      C: CHAR;                                             *     S
385 BEGIN
386   C:=CHR(PASS_NO+ORD('0'));                             384   205   131   205
387   WRITE (FF);                                            74     9
388   WRITE(NL);                                             74     9
389   FOR I:=1 TO LINE_LENGTH DO WRITE(C);                  383   114    74   384
390   WRITE(NL);                                             74     9
391   WRITE(C);                                              74   384
392   J:=LINE_LENGTH DIV 2 - 17;                            383   114
393   FOR I:=2 TO J-1 DO WRITE(' ');                        383   383    74
394   WRITE_TEXT('MCPASCAL INTERMEDIATE CODE PASS $');      373
395   WRITE(C);                                              74   384
396   FOR I:=J+33 TO LINE_LENGTH-1 DO WRITE(' ');           383   383   114    74
397   WRITE(C); WRITE(NL);                                   74   384    74     9
398   FOR I:=1 TO LINE_LENGTH DO WRITE(C);                  383   114    74   384
399   WRITE(NL);                                             74     9
400 END;
401
402 PROCEDURE WRITE 56 CASE (OP_NAME: CHAR8; PASS: INTEGER);    *     *   116     *     S
403 VAR
404      I: INTEGER;                                          *     S
405      MIN,                                                 *
406      MAX,                                                 *
407      MAX_MINUS MIN,                                       *
408      LOCATION,                                            *
409      STMT_LABEL:     SHORTINTEGER;                        *    12
410
411 BEGIN
412 WRITE_CHAR8(OP_NAME);                                   195   402
413 WRITE_CHAR( '(' );                                      185
414 READ_IFL(MIN);                                          149   405
415 WRITE_INT(MIN);                                         224   405
416 WRITE_CHAR( ',' );                                      185
417
418 IF PASS = 5                                             402
419 THEN BEGIN
```

```
420      READ_IFL (MAX);                                            149  406
421      WRITE_INT (MAX);                                           224  406
422      MAX_MINUS_MIN := MAX - MIN                                 407  406  405
423      END
424 ELSE BEGIN  "PASS 6"
425      READ_IFL(MAX_MINUS_MIN);                                   149  407
426      WRITE_INT(MAX_MINUS_MIN);                                  224  407
427      WRITE_CHAR ( ',' );                                        185
428      READ_IFL (LOCATION);                                       149  408
429      WRITE_INT (LOCATION)                                       224  408
430      END;
431 FOR I := 0 TO MAX_MINUS_MIN DO                                  404  407
432    BEGIN
433    WRITE_CHAR(',');                                             185
434    READ_IFL(STMT_LABEL);                                        149  409
435    WRITE_INT(STMT_LABEL);                                       224  409
436    END;
437 WRITE_CHAR( ')' );                                              185
438 END;
439
440
441 PROCEDURE WRITE_PASS7 CASE (OP_NAME: CHAR8);                     *    *   116
442 VAR
443    I: INTEGER;                                                   *    S
444    MIN,                                                          *
445    MAX_MINUS MIN,                                                *
446    NUM_OF_DISTANCES,                                             *
447    ARG:              SHORTINTEGER;                               *   12
448
449 BEGIN
450 WRITE_CHAR8(OP_NAME);                                           195  441
451 WRITE_CHAR( '(' );                                              185
452 READ_IFL(MIN);                                                  149  444
453 WRITE_INT(MIN);                                                 224  444
454 WRITE_CHAR( ',' );                                              185
455
456 READ_IFL(MAX_MINUS_MIN);                                        149  445
457 WRITE_INT(MAX_MINUS_MIN);                                       224  445
458 NUM_OF_DISTANCES := MAX_MINUS_MIN + 1;                          446  445
459 FOR I := 1 TO NUM_OF_DISTANCES DO                               443  446
460    BEGIN
461    WRITE_CHAR(',');                                             185
462    READ_IFL(ARG);                                               149  447
463    WRITE_INT(ARG);                                              224  447
464    END;
465 WRITE_CHAR( ')' );                                              185
466 END;
467
468
469 PROCEDURE PASS5;                                                 *
470 CONST MIN_OP5 = 0;      MAX_OP5 = 48;                            *    *
471 CONST
472    PASS5_TABLE = (                                               *
473          'PUSHCNST', 1,      'PUSHVAR ', 3,      " 1"
474          'PUSHIND ', 1,      'PUSHADDR', 2,
475          'FIELD   ', 1,      'INDEX   ', 3,
476          'POINTER ', 0,      'VARIANT ', 2,
477          'RANGE   ', 2,      'ASSIGN  ', 1,      " 9"
478          'ASSGNTAG', 0,      'COPY    ', 1,
479          'NEW     ', 0,      'NOT     ', 0,
480          'AND     ', 1,       'OR      ', 1,
481          'NEG     ', 1,       'ADD     ', 1,
482          'SUB     ', 1,       'MUL     ', 1,      "19"
483          'DIV     ', 1,       'MOD     ', 1,
484          'INVALID ', 0,       'INVALID ', 0,
485          'FUNCTION', 2,       'BUILDSET', 0,
486          'COMPARE ', 2,       'CMPSTRUC', 2,
487          'FUNCVALU', 2,       'DEFLABEL', 1,      "29"
488          'JUMP    ', 1,       'FALSJUMP', 1,
489          'CASEJUMP', 0,       'INITVAR ', 0,
490          'CALL    ', 3,       'ENTER   ', 5,
491          'RETURN  ', 1,       'POP     ', 1,
492          'NEWLINE ', 1,       'ERROR   ', 0,      "39"
493          'LNGCONST', 0,       'MESSAGE ', 2,
494          'INCRMENT', 0,       'DECRMENT', 0,
495          'PROCDURE', 1,       'INIT    ', 4,
496          'PUSHLABL', 1,       'CALLPROG', 0,      "47"
497          'EOM     ', 1):                          "48"
498
499          ARRAY[MIN_OP5 .. MAX_OP5] OF                                    470   470
500          RECORD
501             OP_NAME: CHAR8;                                                *   116
502             NUM_ARGS: BYTE                                                 *   214
503          END;
504
505 CONST                                                                      *
506    EOM5 = 48;                                                              *
507    NEWLINES5 = 38;                                                         *
508    CASEJUMP5 = 32;                                                         *
509    LONGCONSTANTS = 40;                                                     *
510
511 BEGIN
512    IN_FILE := 2;                                                     128
513    COL_SEP := 3;                                                     137
514    REPEAT
515       READ_IFL (OP);                                                 149   141
516       WITH PASS5_TABLE[OP] DO                                        472   141
517       IF (OP < MIN OP5) OR (OP > MAX_OP5)                            141   470   141   470
518       THEN INDEXERROR ('BAD_OP  ', OP)                               256   141
519       ELSE IF OP = NEWLINE5                                          141   507
520            THEN NEW LINE                                             324
521            ELSE IF OP = CASEJUMP5                                    141   508
522                 THEN WRITE_56_CASE (OP_NAME, PASS_NO)                402   501   131
523                 ELSE IF OP = LONGCONSTANT5                           141   509
524                      THEN WRITE LCONST                               337
525                      ELSE WRITE_OP (OP_NAME, NUM_ARGS, 0, 0, 0, 0)   278   501   502
526    UNTIL OP = EOM5;                                                  141   506
527    WRITE (M.)                                                         74     9
528 END;
529
530
531
532
533 PROCEDURE PASS6;                                                       *
534 CONST     MIN_OP6 = 0;    MAX_OP6 = 113;                               *     *
535 CONST   PASS6_TABLE = (                                                *
536          'CONSTADD', 1, 'LOCALADD', 1,    "  1"
537          'GLOBLADD', 1, 'PUSHCONS', 1,    "   "
538          'PUSHLOCL', 1, 'PUSHGLBL', 1,    "   "
539          'PUSHIND ', 0, 'PUSHBYTE', 0,    "   "
```

```
'PUSHREAL', 0, 'PUSHSET ', 0,   "  9"
'FIELD   ', 1, 'INDEX   ', 3,   "   "
'POINTER ', 0, 'VARIANT ', 2,   "   "
'RANGE   ', 2, 'COPYBYTE', 0,   "   "
'COPYWORD', 0, 'COPYREAL', 0,   "   "
'COPYSET ', 0, 'COPYTAG ', 1,   " 19"
'COPYSTRU', 1, 'NEW     ', 2,   "   "
'NEWINIT ', 2, 'NOT     ', 0,   "   "
'ANDWORD ', 0, 'ANDSET  ', 0,   "   "
'ORWORD  ', 0, 'ORSET   ', 0,   "   "
'NEGWORD ', 0, 'NEGREAL ', 0,   " 29"
'ADDWORD ', 0, 'ADDREAL ', 0,   "   "
'SUBWORD ', 0, 'SUBREAL ', 0,   "   "
'SUBTRSET', 0, 'MULWORD ', 0,   "   "
'MULREAL ', 0, 'DIVWORD ', 0,   "   "
'DIVREAL ', 0, 'MODWORD ', 0,   " 39"
'BUILDSET', 0, 'INSET   ', 0,   "   "
'LSWORD  ', 0, 'EQWORD  ', 0,   "   "
'GRWORD  ', 0, 'NLWORD  ', 0,   "   "
'NEWORD  ', 0, 'NGWORD  ', 0,   "   "
'LSREAL  ', 0, 'EQREAL  ', 0,   " 49"
'GRREAL  ', 0, 'NLREAL  ', 0,   "   "
'NEREAL  ', 0, 'NGREAL  ', 0,   "   "
'EQSET   ', 0, 'NLSET   ', 0,   "   "
'NESET   ', 0, 'NGSET   ', 0,   "   "
'LSSTRUCT', 1, 'EQSTRUCT', 1,   " 59"
'GRSTRUCT', 1, 'NLSTRUCT', 1,   "   "
'NESTRUCT', 1, 'NGSTRUCT', 1,   "   "
'FUNCVALU', 1, 'JUMP    ', 2,   "   "
'FALSEJMP', 2, 'CASEJUMP', 2,   "   "
'INITVAR ', 1, 'CALL    ', 2,   " 69"
'CALLSYS ', 1, 'ENTER   ', 4,   "   "
'EXIT    ', 0, 'ENTRPROG', 4,   "   "
'EXITPROG', 0, 'BEGNCLAS', 4,   "   "
'ENDCLASS', 0, 'ENTRCLAS', 4,   "   "
'EXITCLAS', 0, 'BEGINMON', 4,   " 79"
'ENDMON  ', 0, 'ENTERMON', 4,   "   "
'EXITMON ', 0, 'BEGNPRCS', 1,   "   "
'ENDPRCS ', 0, 'ENTRPRCS', 4,   "   "
'EXITPRCS', 0, 'POP     ', 1,   "   "
'NEWLINE ', 1, 'INCRWORD', 0,   " 89"
'DECRWORD', 0, 'INITCLAS', 3,   "   "
'INITMON ', 3, 'INITPROC', 5,   "   "
'PUSHLABL', 2, 'CALLPROG', 0,   "   "
'TRUNCREA', 0, 'ABSWORD ', 0,   "   "
'ABSREAL ', 0, 'SUCCWORD', 0,   " 99"
'PREDWORD', 0, 'CONVWORD', 0,   "   "
'EMPTY   ', 0, 'ATTRIBUT', 0,   "   "
'REALTIME', 0, 'DELAY   ', 0,   "   "
'CONTINUE', 0, 'I/O     ', 0,   "   "
'START   ', 0, 'STOP    ', 0,   "109"
'SETHEAP ', 0, 'WAIT    ', 0,   "   "
'MESSAGE ', 3, 'EOM     ', 0):  "113"

ARRAY [MIN_OP6..MAX_OP6] OF                                   534   534
     RECORD
     OP_NAME: CHAR8;                                            *   116
     NUM_ARGS: BYTE;                                            *   214
     END;

```

```
600
601 CONST EOM6 = 113;                                                           *
602       NEWLINE6 = 88;                                                        *
603       CASEJUMP6 = 67;                                                       *
604
605 BEGIN
606   IN_FILE:=1;                                                             128
607   COL_SEP:=3;                                                             137
608   REPEAT
609     READ_IFL(OP);                                                         149   141
610     WITH PASS6_TABLE[OP] DO                                               535   141
611         IF (OP < MIN_OP6) OR (OP > MAX_OP6)                               141   534   141   534
612         THEN INDEXERROR ('BAD-OP  ', OP)                                  256   141
613         ELSE IF OP = NEWLINE6                                             141   602
614              THEN NEW LINE                                                324    .
615              ELSE IF OP = CASEJUMP6                                       141   603
616                   THEN WRITE_56_CASE (OP_NAME, PASS_NO)                   402   596   131
617                   ELSE WRITE_OP (OP_NAME, NUM_ARGS, 0, 0, 0, 0);          278   596   597
618   UNTIL OP = EOM6;                                                        141   601
619   WRITE(NL);                                                               74     9
620 END;
621
622
623 PROCEDURE PASS7;                                                            *
624 CONST
625 PASS7_TABLE = (                                                             *
626          'CONSTADD', 1, 'LOCALADD', 1,   "  2"
627          'GLOBLADD', 1, 'PUSHCONS', 1,   "   "
628          'PUSHLOCL', 1, 'PUSHGLBL', 1,   "   "
629          'PUSHIND ', 0, 'PUSHBYTE', 0,   "   "
630          'PUSHREAL', 0, 'PUSHSET ', 0,   " 10"
631          'FIELD   ', 1, 'INDEX   ', 3,   "   "
632          'POINTER ', 0, 'VARIANT ', 2,   "   "
633          'RANGE   ', 2, 'COPYBYTE', 0,   "   "
634          'COPYWORD', 0, 'COPYREAL', 0,   "   "
635          'COPYSET ', 0, 'COPYTAG ', 1,   " 20"
636          'COPYSTRU', 1, 'NEW     ', 2,   "   "
637          'NEWINIT ', 2, 'NOT     ', 0,   "   "
638          'ANDWORD ', 0, 'ANDSET  ', 0,   "   "
639          'ORWORD  ', 0, 'ORSET   ', 0,   "   "
640          'NEGWORD ', 0, 'NEGREAL ', 0,   " 30"
641          'ADDWORD ', 0, 'ADDREAL ', 0,   "   "
642          'SUBWORD ', 0, 'SUBREAL ', 0,   "   "
643          'SUBTRSET', 0, 'MULWORD ', 0,   "   "
644          'MULREAL ', 0, 'DIVWORD ', 0,   "   "
645          'DIVREAL ', 0, 'MODWORD ', 0,   " 40"
646          'BUILDSET', 0, 'INSET   ', 0,   "   "
647          'LSWORD  ', 0, 'EQWORD  ', 0,   "   "
648          'GRWORD  ', 0, 'NLWORD  ', 0,   "   "
649          'NEWORD  ', 0, 'NGWORD  ', 0,   "   "
650          'LSREAL  ', 0, 'EQREAL  ', 0,   " 50"
651          'GRREAL  ', 0, 'NLREAL  ', 0,   "   "
652          'NEREAL  ', 0, 'NGREAL  ', 0,   "   "
653          'EQSET   ', 0, 'NLSET   ', 0,   "   "
654          'NESET   ', 0, 'NGSET   ', 0,   "   "
655          'LSSTRUCT', 1, 'EQSTRUCT', 1,   " 60"
656          'GRSTRUCT', 1, 'NLSTRUCT', 1,   "   "
657          'NESTRUCT', 1, 'NGSTRUCT', 1,   "   "
658          'FUNCVALU', 1, 'JUMP    ', 1,   "   "
659          'FALSEJMP', 1, 'CASEJUMP', 2,   "   "
660          'INITVAR ', 1, 'CALL     ', 1,    " 70"
661          'CALLSYS ', 1, 'ENTER    ', 4,    "   "
662          'EXIT    ', 0, 'ENTRPROG', 4,     "   "
663          'EXITPROG', 0, 'BEGNCLAS', 4,     "   "
664          'ENDCLASS', 0, 'ENTRCLAS', 4,     "   "
665          'EXITCLAS', 0, 'BEGINMON', 4,     " 80"
666          'ENDMON  ', 0, 'ENTERMON', 4,     "   "
667          'EXITMON ', 0, 'BEGNPRCS', 1,     "   "
668          'ENDPRCS ', 0, 'ENTRPRCS', 4,     "   "
669          'EXITPRCS', 0, 'POP     ', 1,     "   "
670          'NEWLINE ', 1, 'INCRWORD', 0,     " 90"
671          'DECRWORD', 0, 'INITCLAS', 2,     "   "
672          'INITMON ', 2, 'INITPROC', 4,     "   "
673          'PUSHLABL', 1, 'CALLPROG', 0,     "   "
674          'TRUNCREA', 0, 'ABSWORD ', 0,     "   "
675          'ABSREAL ', 0, 'SUCCWORD', 0,     "100"
676          'PREDWORD', 0, 'CONVWORD', 0,     "   "
677          'EMPTY   ', 0, 'ATTRIBUT', 0,     "   "
678          'REALTIME', 0, 'DELAY   ', 0,     "   "
679          'CONTINUE', 0, 'I/O     ', 0,     "   "
680          'START   ', 0, 'STOP    ', 0,     "110"
681          'SETHEAP ', 0, 'WAIT    ', 0):    "112"
682
683          ARRAY [1..112] OF
684               RECORD
685               OP_NAME: CHAR8;                                                  *   116
686               NUM_ARGS: SHORTINTEGER;                                          *    12
687               END;
688
689 NEWLINE7 = 178;                                                                *
690 CASEJUMP7 =136;                                                                *
691
692 VAR
693    HALF_OP: SHORTINTEGER;                                                      *    12
694    PROGLENGTH, STACKLENGTH, VARLENGTH, CONSTANTS: SHORTINTEGER;                *    *     *     *    12
695
696
697 BEGIN
698    IN_FILE := 3;                                                              128
699    COL_SEP := 1;                                                              137
700
701    READ_IFL(PROGLENGTH);                                                      149   694
702    READ_IFL(CODELENGTH);                                                      149   142
703    READ_IFL(STACKLENGTH);                                                     149   694
704    READ_IFL(VARLENGTH);                                                       149   694
705    READ_IFL(CONSTANTS);                                                       149   694
706    CODE_READ_SO_FAR := 0;                                                     143
707                                                                              *
708    WRITE_CHAR8 ('PROGLEN=');   WRITE_INT (PROGLENGTH);                        195   224   694
709    WRITE_CHAR8('        (:10:)'  );                                           195
710    WRITE_CHAR8 ('CODELEN=');   WRITE_INT (CODELENGTH);                        195   224   142
711    WRITE_CHAR8('        (:10:)'   );                                          195
712    WRITE_CHAR8 ('STCKLEN=');   WRITE_INT (STACKLENGTH);                       195   224   694
713    WRITE_CHAR8('        (:10:)'   );                                          195
714    WRITE_CHAR8 ('VARLEN =');   WRITE_INT (VARLENGTH);                         195   224   694
715    WRITE_CHAR8('        (:10:)'   );                                          195
716    WRITE_CHAR8 ('CONSTS =');  WRITE_INT (CONSTANTS);                          195   224   694
717    WRITE_CHAR8 ('        (:10:)'  );                                          195
718
719    REPEAT
```

```
720         READ_IFL(OP);                                                      149   141
721         HALF_OP := OP DIV 2;                                               693   141
722         IF (HALF_OP <= 0)   OR   (HALF_OP > 112)                           693   693
723         THEN BEGIN
724              WRITE_CHAR8 ('BAD OP: ');                                     195
725              WRITE_INT (OP);                                               224   141
726              WRITE_CHAR8('        (:10:)'  );                              195
727              FOR I := 1 TO PAGELENGTH + 10 DO WRITE_CHAR(' ');             443    11   185
728              END;
729         WITH PASS7_TABLE[HALF_OP] DO                                       625   693
730              IF OP = NEWLINE7                                              141   689
731              THEN NEW_LINE                                                 324
732              ELSE IF OP = CASEJUMP7                                        141   690
733                   THEN WRITE_PASS7_CASE (OP_NAME)                          441   685
734                   ELSE WRITE_OP (OP_NAME, NUM_ARGS, 0, 0, 0, 0)            278   685   686
735     UNTIL CODE_READ_SO_FAR >= CODELENGTH;                                  143   142
736     WRITE_CHAR8 ('        (:10:)'  );                                      195
737     WRITE_CHAR8 ('END-CODE');                                              195
738     WRITE_CHAR (NL);                                                       185     9
739     WRITE_CHAR8 ('CONSTANT');                                              195
740     WRITE_CHAR8 ('S-IN-HEX');                                              195
741     WRITE_CHAR8 ('        (:10:)'  );                                      195
742     REPEAT
743         READ_IFL (OP);                                                     149   141
744         IF OUT_COL_PTR + OUT_BUF_PTR - 1 + 9 > LINE_LENGTH                 132   133   114
745         THEN BEGIN
746              WRITE_BUF(FALSE);                                             176   332
747              WRITE (NL);                                                    74     9
748              OUT_COL_PTR := 1;                                             132
749              NEXT_COL;                                                     162
750              END;
751         WRITE_HEX (OP, 2);                                                 209   141
752         WRITE_CHAR (' ');                                                  185
753         WRITE_CHAR (' ');                                                  185
754     UNTIL CODE_READ_SO_FAR >= PROGLENGTH - 8;                              143   694
755     FOR I := 1 TO PAGELENGTH +10 DO WRITE_CHAR(' ');                       443    11   185
756 END;
757
758
759
760 BEGIN
761   INITIALIZE;                                                              359
762   WRITE_HEADER;                                                            382
763   CASE PASS_NO OF                                                          131
764     1:"PASS1";
765     2:"PASS2";
766     3:"PASS3";
767     4:"PASS4";
768     5: PASS5;                                                              469
769     6: PASS6;                                                              533
770     7: PASS7;                                                              623
771     8, 9: "NOT IMPLEMENTED";
772   END;
773 END.
```

```
CROSS REFERENCE   * IS DEF   = IS ASG

 -A-
  A              225*  230=  234   240*  245=  249=  252
  ABS            230   245
  ACCEPT          86*
  ADDR            27*
  ALIGN          176*  179
  ARG             46*   93*   94*  149*  156=  325*  327   331   338*  346   350   447*  462   463
  ARGLIST         65*  106   110
  ARGSEQ          67*   93    94
  ARGTAG          51*   57
  ARGTYPE         56*   65    93    94
  ARG_NO         280*  283=  285   290   296   302   308
  ARG_VAL        280*  288   292   292   293   298   298   299   304   304   305   310   310   311   314   314   315   316
  ASCII           23
  ATTR            96*

 -B-
  BACKSPACE       37
  BADINDEX       256*  270
  BLOCK           78*   79*   89*   90*  100*
  BOOL            58
  BOOLEAN         28    54    58    76    89    90    97   176
  BOOLTYPE        52    58
  BYTE           214   502   597
  BYTES          214   219   220
  BYTES_PER_IN    13*  157

 -C-
  C               73*   74*   86*   87*  185*  189   384*  386=  389   391   395   397   398
  CALLERROR       71
  CARDDEVICE      33
  CASEJUMP5      508*  521
  CASEJUMP6      603*  615
  CASEJUMP7      690*  732
  CHAR            16    19    73    74    86    87   116   185   225   240   384
  CHAR8          116*  195   256   278   402   441   501   596   685
  CHR            205   206   230   245   386
  CLOSE           77*
  CODELENGTH     142*  702   710   735
  CODELIMIT       71
  CODE_READ_SO   143*  157=  157   706=  735   754
  COL_SEP        137*  166   166   368=  513=  607=  699=
  COMPLETE        40
  CONCODE         23
  CONSTANTS      694*  705   716
  CONTEXT_ARG    279*  308
  CONTROL         35
  CR               9*

 -D-
  DEVICE         100*  102*
  DISKDEVICE      33
  DISPLAY         87*

 -E-
  EM               9*
  EMPTY           23
 ENDFILE        41
 ENDMEDIUM      41
 EOF            89*   90*
 EOM5          506*  526
 EOM6          601*  618
 EOM_OP        138*

-F-
 F              76*   77*   78*   79*   80*
 FAILURE        40
 FALSE         332   348   353   746
 FF              9*  387
 FILE           21*   76    77    78    79    80
 FILEATTR       25*   96
 FILEKIND       23*   26
 FIRST_COL     135*  165   166   168   170   366=
 FOUND          76*   97*

-G-
 GET            78*  152

-H-
 HALF_OP       693*  721=  722   722   729
 HEADER         85*
 HEAPLIMIT      71

-I-
 I             144*  163*  165=  166=  167   167=  168=  168   169   170=  172   173   177*  180=  180   196*  198=  199
               200   211*  218=  219   220   226*  228=  230   231=  231   234   241*  243=  245   246=  246   249   249=
               249   251   252   260*  280*  338*  345=  351   374*  376=  377   378   378=  378   383*  389=  393=  396=
               398=  404*  431=  443*  459=  727=  755=
 ID             60    76*   96*  106*
 IDENTIFIER     19*   60    76    96   106
 IDENTIFY       85*
 IDLENGTH       18*   19
 IDTYPE         52    60*
 INDEXERROR    256*  293   299   305   311   518   612
 INITIALIZE    359*  761
 INP            67
 INPUT          35
 INPUTTASK      49
 INT            59   213   217=  363
 INTEGER        27    29    59    78    79    80    82    83   107   127   128   129   131   132   133   135   136   137
               143   144   163   177   196   226   237   237   241   260   374   383   402   404   443
 INTERVENTION   40
 INTTYPE        52    59*
 IN_FILE       128*  152   512=  606=  698=
 IOARG          37*   46
 IODEVICE       32*  100   102
 IOMOVE        102*
 IOOPERATION    35*   44
 IOPARAM        43*  100   102
 IORESULT       39*   45
 IOTRANSFER     99*

-J-
 J             163*  172=  226*  234=  234   241*  251=  252=  252   383*  392=  393   396
 JOBTASK        49

-K-
```

```
KIND           26*
KIND_ARG      278*  290

-L-
LCONST_OP     140*
LEN           338*  341   342   344
LENGTH         80*
LINE           16*   85    91    92   107*  134   373
LINELENGTH     15*   16
LINE_LENGTH   114*  169   187   347   389   392   396   398   744
LINE_OP       139*
LOCATION      408*  428   429
LONGCONSTANT  509*  523
LOOKUP         96*

-M-
MARK           82*
MAX           406*  420   421   422
MAXARG         64*   65
MAX_CONTEXT   122*  310
MAX_KIND      119*  292
MAX_MINUS_MI  407*  422=  425   426   431   445*  456   457   458
MAX_MODE      121*  304
MAX_OP5       470*  499   517
MAX_OP6       534*  594   611
MAX_TYPE      120*  298
MIN           405*  414   415   422   444*  452   453
MIN_COL_SEP   136*  166   367=
MIN_CONTEXT   122*  310
MIN_KIND      119*  292
MIN_MODE      121*  304
MIN_OP5       470*  499   517
MIN_OP6       534*  594   611
MIN_TYPE      120*  298
MNEM          110*
MODE_ARG      279*  302
MOVE           35

-N-
N             209*  218   237*  251   338*  344=  345   351
NEWLINE5      507*  519
NEWLINE6      602*  613
NEWLINE7      689*  730
NEW_LINE      324*  520   614   731
NEXT_COL      162*  179   344   348   749
NILTYPE        52    58
NL              9*  170   328   329   348   388   390   397   399   527   619   738   747
NOTUSED        29*
NUM_ARGS      278*  283   318   502*  525   597*  617   686*  734
NUM_OF_DISTA  446*  458=  459

-O-
OP            141*  515   516   517   517   518   519   521   523   526   609   610   611   611   612   613   615   618
              720   721   725   730   732   743   751
OPEN           76*
OPERATION      44*
OP_NAME       278*  282   402*  412   441*  450   501*  522   525   596*  616   617   685*  733   734
ORD           205   206   230   245   386
OUT            67
OUTPUT         35
```

```
 OUTPUTTASK      49
 OUT_BUF        134*  180   189=
 OUT_BUF_PTR    133*  169   180   181   182=  187   189   190=  190   347   365=  744
 OUT_COL_PTR    132*  166   170=  172   173=  181=  181   328   330=  347   348=  364=  744   748=
 OVERFLOW        70

-P-
 P               78*   79*
 PAGE            12*   78    79    89    90   100   130
 PAGELENGTH      11*   12   151   361   727   755
 PAGES_IN       129*  152   153=  153   362=
 PAGE_IN        130*  152   156
 PARAM          100*  102*  106*  110*  363
 PASS           402*  418
 PASS5          469*  768
 PASS5_TABLE    472*  516
 PASS6          533*  769
 PASS36_TABLE   535*  610
 PASS7          623*  770
 PASS7_TABLE    625*  729
 PASS_NO        131*  363=  386   522   616   763
 POINTER         54*   61
 POINTERERROR    70
 PRINTDEVICE     33
 PROGLENGTH     694*  701   708   754
 PROGRESULT      69*  107
 PROTECTED       28*
 PTR             61
 PTRTYPE         52    61*
 PUT             79*

-R-
 RANGEERROR      70
 READ            73*
 READARG         93*
 READLINE        91*
 READPAGE        89*
 READ_IFL       149*  288   327   341   346   414   420   425   428   434   452   456   462   515   609   701   702   703
                704   705   720   743
 RELEASE         83*
 REM            226*  228=  230   231=  231   232   241*  243=  245   246=  246   247
 RESULT         107*
 REWIND          37
 RUN            106*

-S-
 S               93*   94*
 SCRATCH         23
 SROCODE         23
 SHORTINTEGER    12   138   139   140   141   142   149   203   209   209   211   213   224   256   279   280   325   338
                409   447   686   693   694
 STACKLENGTH    694*  703   712
 STACKLIMIT      71
 STARTMEDIUM     41
 STATUS          45*
 STMT_LABEL     409*  434   435

-T-
 TAG             57*
 TAG1           210   213*
 TAG2           210   214*
 TAGS           210*  212*
 TAPEDEVICE      33
 TASK           104*
 TASKKIND        49*  104
 TERMINATED      70
 TEXT            91*   92*  195*  199   200   256*  267   373*  377   378
 TIMELIMIT       71
 TOP             82*   83*
 TRANSMISSION    40
 TRUE           188   319
 TYPEDEVICE      33
 TYPE_ARG       278*  296

-U-
 UPSPACE         37

-V-
 VAL            203*  205   205   206   209*  217   224*  228   233   237*  243   248
 VARIANTERROR    70
 VARLENGTH      694*  704   714

-W-
 WORDS_IN       127*  151   153=  155=  155   156   361=
 WRITE           74*  170   172   180   328   329   348   378   387   388   389   390   391   393   395   396   397   397
                398   399   527   619   747
 WRITEARG        94*
 WRITEEOF        37
 WRITELINE       92*
 WRITEPAGE       90*
 WRITE_56_CAS   402*  522   616
 WRITE_BUF      176*  188   319   332   348   353   746
 WRITE_CHAR     185*  200   205   206   233   234   251   252   264   268   269   271   272   286   287   318   340   342
                351   351   413   416   427   433   437   451   454   461   465   727   738   752   753   755
 WRITE_CHAR8    195*  262   263   265   266   267   282   294   300   306   312   331   340   343   412   450   708   709
                710   711   712   713   714   715   716   717   724   726   736   737   739   740   741
 WRITE_HEADER   382*  762
 WRITE_HEX      209*  315   350   751
 WRITE_HEX_CH   203*  219   220
 WRITE_INT      224*  270   316   342   415   421   426   429   435   453   457   463   708   710   712   714   716   725
 WRITE_INTR     237*  331
 WRITE_LCONST   337*  524
 WRITE_OP       278*  525   617   734
 WRITE_PASS7_   441*  733
 WRITE_TEXT     373*  394

-X-
 X              212*  217   219   220

END XREF  230 IDENTIFIERS   948 TOTAL REFERENCES
 129 COLLISIONS.
```

```
       1 "MEPASCAL MEMNEM - MNEMONIC TEST OUTPUT FORMATTING"
       2 "FOR WESTERN DIGITAL PASCAL MICROENGINE"
       3
       4 "$ CONCURRENT := TRUE"
       5
       6 "############
C      7 #  PREFIX  #
C      8 ############"
       9
      10 TYPE FULLWORD = INTEGER;                                     *     S
      11      INTEGER  = SHORTINTEGFR;                                S     *
      12
      13 CONST NL= '(:10:)';   FF= '(:12:)';   CR= '(:13:)';   EM= '(:25:)';          *     *     *     *
      14
      15 CONST PAGELENGTH = 512 DIV 2;                                *
      16 TYPE PAGE = ARRAY (.1..PAGELENGTH.) OF INTEGER;              *     15    S
      17 CONST BYTES_PER_INTEGER = 2;                                 *
      18
      19 CONST LINELENGTH = 70;                                       *
      20 TYPE LINE = ARRAY (.1..LINELENGTH.) OF CHAR;                 *     19    S
      21
      22 CONST IDLENGTH = 12;                                         *
      23 TYPE IDENTIFIER = ARRAY (.1..IDLENGTH.) OF CHAR;             *     22    S
      24
      25 TYPE FILE = 1..5;                                            *
      26
      27 TYPE FILEKIND = (EMPTY, SCRATCH, ASCII, SEQCODE, CONCODE);   *     *     *     *     *     *
      28
      29 TYPE FILEATTR = RECORD                                       *
      30                   KIND: FILEKIND;                            *     27
      31                   ADDR: INTEGER;                             *     S
      32                   PROTECTED: BOOLEAN;                        *     S
      33                   NOTUSED: ARRAY (.1..5.) OF INTEGER         *     S
      34                 END;
      35
      36 TYPE IODEVICE =                                              *
      37   (TYPEDEVICE, DISKDEVICE, TAPEDEVICE, PRINTDEVICE, CARDDEVICE);                 *     *     *     *     *
      38
      39 TYPE IOOPERATION = (INPUT, OUTPUT, MOVE, CONTROL);           *     *     *     *     *
      40
      41 TYPE IOARG = (WRITEEOF, REWIND, UPSPACE, BACKSPACE);         *     *     *     *     *
      42
      43 TYPE IORESULT =                                              *
      44   (COMPLETE, INTERVENTION, TRANSMISSION, FAILURE,            *     *     *     *
      45    ENDFILE, ENDMEDIUM, STARTMEDIUM);                         *     *     *
      46
      47 TYPE IOPARAM = RECORD                                        *
      48                  OPERATION: IOOPERATION;                     *     39
      49                  STATUS: IORESULT;                           *     43
      50                  ARG: IOARG                                  *     41
      51                END;
      52
      53 TYPE TASKKIND = (INPUTTASK, JOBTASK, OUTPUTTASK);            *     *     *     *
      54
      55 CONST MAXWORD = 100;                                         *
      56
      57 TYPE
      58 POINTER = @INTEGER;                                          *     S
      59
```

```
 60 TABLEPTR = @TABLE;                                                  •     •
 61 TABLE = RECORD                                                      •
 62    NEXTPORTION: TABLEPTR;                                           •    60
 63    CONTENTS: ARRAY[1..MAXWORD] OF INTEGER                           •    55     S
 64    END;
 65
 66 TABLEPART = RECORD                                                  •
 67    SEGDISTANCE, STACKLENGTH: INTEGER;                               •     •     S
 68    JUMPTABLE, CONSTTABLE, XJPTABLE,                                 •     •     •
 69    EXITICTABLE, DATASIZETABLE: TABLEPTR                             •     •    60
 70    END;
 71
 72 TABLESPTR = @TABLEPART;                                             •    66
 73
 74 OPTION = 0..8;                                                      •
 75
 76 CONST MAXINTFAC = 14;                                               •
 77 TYPE
 78 IFPTR = @IFINFO;                                                    •     •
 79 IFINFO = RECORD                                                     •
 80    INTERFACES: INTEGER;                                             •     S
 81    INTERFACESIZES: ARRAY[1..MAXINTFAC] OF INTEGER                   •    76     S
 82    END;
 83
 84 PASSPTR = @PASSLINK;                                                •     •
 85 PASSLINK = RECORD                                                   •
 86    OPTIONS: SET OF OPTION;                                          •    74
 87    LABELS, BLOCKS, CONSTANTS, XJP_OFFSETS: INTEGER;                 •     •     •     •     S
 88    RESETPOINT: FULLWORD;                                            •    10
 89    TABLES: TABLESPTR;                                               •    72
 90    INTERFACE: IFPTR                                                 •    78
 91    END;
 92
 93 TYPE ARGTAG =                                                       •
 94   (NILTYPE, BOOLTYPE, INTTYPE, IDTYPE, PTRTYPE);                    •     •     •     •     •
 95
 96 TYPE ARGTYPE = RECORD                                               •
 97                   CASE TAG: ARGTAG OF                               •    93
 98                     NILTYPE, BOOLTYPE: (BOOL: BOOLEAN);            94    94     •     S
 99                     INTTYPE: (INT: INTEGER);                        •     •     S
100                     IDTYPE: (ID: IDENTIFIER);                       •     •    23
101                     PTRTYPE: (PTR: PASSPTR)                         •     •    84
102                 END;
103
104 CONST MAXARG = 10;                                                  •
105 TYPE ARGLIST = ARRAY (.1..MAXARG.) OF ARGTYPE;                      •   104    96
106
107 TYPE ARGSEQ = (INP, OUT);                                           •     •     •
108
109 TYPE PROGRESULT =                                                   •
110   (TERMINATED, OVERFLOW, POINTERERROR, RANGEERROR, VARIANTERROR,    •     •     •     •     •
111    HEAPLIMIT, STACKLIMIT, CODELIMIT, TIMELIMIT, CALLERROR);         •     •     •     •     •
112
113 PROCEDURE READ(VAR C: CHAR);                                        •     •     S
114 PROCEDURE WRITE(C: CHAR);                                           •     •     S
115
116 PROCEDURE OPEN(F: FILE; ID: IDENTIFIER; VAR FOUND: BOOLEAN);        •     •    25     •    23     •     S
117 PROCEDURE CLOSE(F: FILE);                                           •     •    25
118 PROCEDURE GET(F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);           •     •    25     •     S     •    16
119 PROCEDURE PUT(F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);           •     •    25     •     S     •    16
```

```
FUNCTION LENGTH(F: FILE): INTEGER;                                          *     *    25    S

PROCEDURE MARK(VAR TOP: INTEGER);                                           *     *    S
PROCEDURE RELEASE(TOP: INTEGER);                                            *     *    S

PROCEDURE IDENTIFY(HEADER: LINE);                                           *     *    20
PROCEDURE ACCEPT(VAR C: CHAR);                                              *     *    S
PROCEDURE DISPLAY(C: CHAR);                                                 *     *    S

PROCEDURE READPAGE(VAR BLOCK: UNIV PAGE; VAR EOF: BOOLEAN);                 *     *    16    *    S
PROCEDURE WRITEPAGE(BLOCK: UNIV PAGE; EOF: BOOLEAN);                        *     *    16    *    S
PROCEDURE READLINE(VAR TEXT: UNIV LINE);                                    *     *    20
PROCEDURE WRITELINE(TEXT: UNIV LINE);                                       *     *    20
PROCEDURE READARG(S: ARGSEQ; VAR ARG: ARGTYPE);                             *     *   107    *    96
PROCEDURE WRITEARG(S: ARGSEQ; ARG: ARGTYPE);                                *     *   107    *    96

PROCEDURE LOOKUP(ID: IDENTIFIER; VAR ATTR: FILEATTR; VAR FOUND: BOOLEAN);     *     *    23    *    29    *    S

PROCEDURE IOTRANSFER                                                        *
  (DEVICE: IODEVICE; VAR PARAM: IOPARAM; VAR BLOCK: UNIV PAGE);             *    36    *    47    *    16

PROCEDURE IOMOVE(DEVICE: IODEVICE; VAR PARAM: IOPARAM);                     *     *    36    *    47

FUNCTION TASK: TASKKIND;                                                    *    53

PROCEDURE RUN(ID: IDENTIFIER; VAR PARAM: ARGLIST;                           *     *    23    *   105
              VAR LINE: INTEGER; VAR RESULT: PROGRESULT);                   *     S    *   109


PROGRAM MEMNEM(VAR PARAM: ARGLIST);                                         *     *   105

"SIMPLE TYPES AND CONSTANTS"

CONST LINE_LENGTH = 70;      "OUTPUT DEV LINE LENGTH"                       *

TYPE CHAR8 = ARRAY [1..8] OF CHAR;                                          *     S

CONST
   MIN KIND = 0;   MAX_KIND = 0;                                            *     *
   MIN_TYPE = 0;   MAX_TYPE = 0;                                            *     *
   MIN MODE = 0;   MAX_MODE = 0;                                            *     *
   MIN CONTEXT = 0;   MAX_CONTEXT = 0;                                      *     *
SFIELD = 6; LFIELD = 11; "SHORT & LONG FIELD WIDTHS"                        *     *


"VARIABLES"

VAR WORDS_IN: INTEGER;     "WORD IN IFL BUFFER"                             *     S
    IN_FILE: INTEGER;       "INPUT FILE"                                    *     S
    PAGES_IN: INTEGER;     "CURRENT PAGE NUMBER"                            *     S
    PAGE_IN: PAGE;         "INPUT BUFFER"                                   *    16
    PASS_NO: INTEGER;      "COMPILER PASS NUMBER"                           *     S
    OUT_COL_PTR: INTEGER;  "POSN ON OUTPUT LINE FOR BUF"                    *     S
    OUT_BUF_PTR: INTEGER;  "POSN IN BUFFER"                                 *     S
    OUT_BUF: LINE;         "OPERATOR OUTPUT BUFFER"                         *   146
    FIRST_COL: INTEGER;    "FIRST COL TO USE FOR NON-NEWLINE"               *     S
    MIN_COL_SEP: INTEGER;  "MIN BLANKS BETWEEN OPS"                         *     S
    COL_SEP: INTEGER;      "SEP BETWEEN COL BOUNDARIES"                     *     S
    OP: INTEGER;           "CURRENT INPUT OPERATOR"                         *     S
    LINK: PASSPTR;                                                          *    84
```

```
180
181
182 "INPUT FILE SUPPORT ROUTINES"
183
184 PROCEDURE READ_IFL (VAR ARG: INTEGER);                        *     *     S
185 BEGIN
186   IF WORDS_IN = PAGELENGTH THEN BEGIN                         167   15
187     GET(IN_FILE,PAGES_IN,PAGE_IN);                            118   168   169   170
188     PAGES_IN:=PAGES_IN+1; WORDS_IN:=0;                        169   169   167
189   END;
190   WORDS_IN:=WORDS_IN+1;                                       167   167
191   ARG:=PAGE_IN[WORDS_IN];                                     184   170   167
192 END;
193
194 "BUFFER OUTPUT ROUTINES"
195
196 PROCEDURE NEXT_COL;                                           *
197 VAR I, J: INTEGER;                                            *     *     S
198 BEGIN  "PAD TO START OF NEXT OP COLUMN"
199   I := OUT_COL_PTR + MIN_COL_SEP;                             197   172   176
200   IF I+OUT_BUF_PTR >= LINE_LENGTH  "WON'T FIT"                197   173   153
201   THEN BEGIN
202       WRITE(NL);                                              114   13
203       I := FIRST_COL;                                         197   175
204       OUT_COL_PTR := 1;                                       172
205       END;
206   FOR J:=OUT_COL_PTR TO I-1 DO WRITE(' ');                    197   172   197   114
207   OUT_COL_PTR:=I;                                             172   197
208 END;
209
210 PROCEDURE WRITE_BUF (ALIGN: BOOLEAN);                         *     *     S
211 VAR I: INTEGER;                                               *     S
212 BEGIN
213   IF ALIGN THEN NEXT_COL;                                     210   196
214   FOR I:=1 TO OUT_BUF_PTR-1 DO WRITE(OUT_BUF[I]);             211   173   114   174   211
215   OUT_COL_PTR:=OUT_COL_PTR+OUT_BUF_PTR-1;                     172   172   173
216   OUT_BUF_PTR:=1;                                             173
217 END;
218
219 PROCEDURE WRITE_CHAR (C: CHAR);                               *     *     S
220 BEGIN
221   IF OUT_BUF_PTR >= LINE_LENGTH THEN                          173   153
222     WRITE_BUF(TRUE);                                          210   *
223   OUT_BUF[OUT_BUF_PTR]:=C;                                    174   173   219
224   OUT_BUF_PTR:=OUT_BUF_PTR+1;                                 173   173
225 END;
226
227 "BUFFER FORMATTING ROUTINES"
228
229 PROCEDURE WRITE_CHAR8 (TEXT: CHAR8);                          *     *     155
230 VAR I: INTEGER;                                               *     S
231 BEGIN
232       FOR I := 1 TO 8 DO                                      230
233           IF (TEXT[I] <> ' ')                                 229   230
234           THEN WRITE_CHAR (TEXT[I])                           219   229   230
235 END;
236
237 PROCEDURE WRITE_HEX_CHAR (VAL: INTEGER);                      *     *     S
238 BEGIN
239   IF VAL > 9 THEN WRITE_CHAR(CHR(VAL-10+ORD('A')))            237   219   *     237   *
```

```
240     ELSE WRITE_CHAR(CHR(VAL+ORD('0')));                          219  239  237  239
241 END;
242
243 PROCEDURE WRITE_HEX (VAL: INTEGER; N: INTEGER);                    *    *    S    *    S
244 TYPE TAGS = (TAG1, TAG2);                                          *    *    *
245 VAR I: INTEGER;                                                    *    S
246     X: RECORD CASE TAGS OF                                         *    *
247       TAG1: (INT: INTEGER);                                        *   99    S
248       TAG2: (BYTES: ARRAY [1..2] OF BYTE);                         *    *    *
249     END;
250 BEGIN
251   X.INT:=VAL;                                                    246   99  243
252   FOR I:=3-N TO 2 DO BEGIN                                       245  243
253     WRITE_HEX_CHAR(X.BYTES[I] DIV 16);                           237  246  248  245
254     WRITE_HEX_CHAR(X.BYTES[I] MOD 16);                           237  246  248  245
255   END;
256 END;
257
258 PROCEDURE WRITE_INT (VAL: INTEGER);                                *    *    S
259 VAR A: ARRAY [1..10] OF CHAR;                                      *    S
260     I, J, REM: INTEGER;                                            *    *    *    S
261 BEGIN
262   REM:=VAL; I:=1;                                                260  258  260
263   REPEAT
264     A[I]:=CHR(ABS(REM MOD 10)+ORD('0'));                         259  260  239    *  260  239
265     I:=I+1; REM:=REM DIV 10;                                     260  260  260  260
266   UNTIL REM = 0;                                                 260
267   IF VAL<0 THEN WRITE_CHAR('-');                                 258  219
268   FOR J:=I-1 DOWNTO 1 DO WRITE_CHAR(A[J]);                       260  260  219  259  260
269 END;
270
271 PROCEDURE WRITE_INTR (VAL: INTEGER; N: INTEGER);                   *    *    S    *    S
272 "WRITE INTEGER, RIGHT-JUST IN FIELD OF WIDTH N"
273 VAR A: ARRAY [1..10] OF CHAR;                                      *    S
274     I, J, REM: INTEGER;                                            *    *    *    S
275 BEGIN
276   REM:=VAL; I:=1;                                                274  271  274
277   REPEAT
278     A[I]:=CHR(ABS(REM MOD 10)+ORD('0'));                         273  274  239  264  274  239
279     I:=I+1; REM:=REM DIV 10;                                     274  274  274  274
280   UNTIL REM = 0;                                                 274
281   IF VAL < 0 THEN BEGIN                                          271
282     A[I]:='-'; I:=I+1;                                           273  274  274  274
283   END;
284   FOR J:=I TO N DO WRITE_CHAR(' ');                              274  274  271  219
285   FOR J:=I-1 DOWNTO 1 DO WRITE_CHAR(A[J]);                       274  274  219  273  274
286 END;
287
288
289 PROCEDURE INDEXERROR (TEXT:CHAR8; BADINDEX: INTEGER);              *    *  155    *    S
290 VAR I: INTEGER;                                                    *    S
291 BEGIN
292 WRITE_CHAR8 ('$$$$$$$$');                                        229
293 WRITE_CHAR8 ('INDEX   ');                                        229
294 WRITE_CHAR (' ');                                                219
295 WRITE_CHAR8 ('ERROR   ');                                        229
296 WRITE_CHAR8 ('$$$$$$$$');                                        229
297 WRITE_CHAR8 (TEXT);                                              229  289
298 WRITE_CHAR ('=');                                                219
299 WRITE_CHAR (' ');                                                219
```

```
300 WRITE_INT (BADINDEX);                                                   258   289
301 WRITE_CHAR (' ');                                                       219
302 WRITE_CHAR (' ');                                                       219
303 END;
304
305
306 "GENERAL OPERATOR FORMATTING ROUTINE"
307
308 PROCEDURE WRITE_OP (OP_NAME: CHAR8; NUM_ARGS, KIND_ARG, TYPE_ARG,          *     *    155    *     *     *
309                     MODE_ARG, CONTEXT_ARG: INTEGER);                        *     *     S
310 VAR ARG_NO, ARG_VAL, I: INTEGER;                                            *     *     *     S
311 BEGIN
312 WRITE_CHAR8 (OP_NAME);                                                  229   308
313 FOR ARG_NO := 1 TO NUM_ARGS DO                                          310   308
314     BEGIN
315     IF ARG_NO = 1                                                       310
316     THEN WRITE_CHAR (  '('  )                                           219
317     ELSE WRITE_CHAR (  ','  );                                          219
318     READ_IFL (ARG_VAL);                                                 184   310
319     IF
320          ARG_NO = KIND_ARG THEN                                         310   308
321              BEGIN
322              IF (ARG_VAL<MIN KIND) OR (ARG_VAL>MAX_KIND)                310   158   310   158
323              THEN INDEXERROR ('BAD-KIND', ARG_VAL)                      289   310
324              ELSE WRITE_CHAR8 ('NO_KIND_')                              229
325              END   ELSE IF
326          ARG_NO = TYPE_ARG THEN                                         310   308
327              BEGIN
328              IF (ARG_VAL<MIN TYPE) OR (ARG_VAL>MAX_TYPE)                310   159   310   159
329              THEN INDEXERROR ('BAD-TYPE', ARG_VAL)                      289   310
330              ELSE WRITE_CHAR8 ('NO_TYPE_')                              229
331              END   ELSE IF
332          ARG_NO = MODE_ARG THEN                                         310   309
333              BEGIN
334              IF (ARG_VAL<MIN MODE) OR (ARG_VAL>MAX_MODE)                310   160   310   160
335              THEN INDEXERROR ('BAD-MODE', ARG_VAL)                      289   310
336              ELSE WRITE_CHAR8 ('NO_MODE_')                              229
337              END   ELSE IF
338          ARG_NO = CONTEXT_ARG THEN                                      310   309
339              BEGIN
340              IF (ARG_VAL<MIN CONTEXT) OR (ARG_VAL>MAX_CONTEXT)          310   161   310   161
341              THEN INDEXERROR ('BAD-CTXT', ARG_VAL)                      289   310
342              ELSE WRITE_CHAR8 ('NO_CNTXT')                              229
343              END   ELSE IF
344          (ARG_VAL < -32768) OR (ARG_VAL > 32767)                        310   310
345              THEN WRITE_HEX (ARG_VAL, 2)                                243   310
346          ELSE WRITE_INT (ARG_VAL)                                       258   310
347     END;  "OF BEGIN"
348   IF NUM_ARGS>0 THEN WRITE_CHAR(')');                                   308   219
349   WRITE_BUF(TRUE);                                                      210   222
350 END;
351
352 "SPECIAL CASE ROUTINE FOR NEW LINE OPERATOR"
353
354 PROCEDURE NEW_LINE;                                                       *
355 VAR ARG: INTEGER;                                                         *     S
356 BEGIN
357   IF PASS_NO < 6                                                        171
358   THEN READ_IFL(ARG);  "LINE NUMBER"                                    184   355
359   IF OUT_COL_PTR<>1 THEN WRITE_CHAR(NL);                                172   219    13
```

```
360   WRITE_CHAR(NL);  "DOUBLE SPACE BEFORE NEW LINE"                     219    13
361   OUT_COL_PTR:=1;                                                     172
362   IF PASS_NO < 6                                                      171
363   THEN BEGIN
364        WRITE_CHAR8('LINE    ');                                       229
365        WRITE_INTR(ARG,5);                                             271   355
366        WRITE_BUF(FALSE)                                               210     *
367        END
368   ELSE WRITE_BUF(FALSE);                                              210   366
369 END;
370
371
372 PROCEDURE SKIP (LINES: INTEGER);                                        *     *     S
373 VAR I: INTEGER;                                                         *     S
374 BEGIN
375 IF OUT_COL_PTR <> 1                                                   172
376 THEN WRITE_CHAR (NL);                                                 219    13
377 FOR I := 1 TO LINES DO WRITE_CHAR(NL);                                373   372   219    13
378 OUT_COL_PTR := 1;                                                     172
379 WRITE_BUF(FALSE)                                                      210   366
380 END;
381
382
383
384 "SPECIAL CASE ROUTINE FOR LONG CONSTANT"
385
386 PROCEDURE WRITE_LCONST;                                                 *
387 VAR LEN, I, N, ARG: INTEGER;                                            *     *     *     *     S
388 BEGIN
389   WRITE_CHAR8('LNGCONST'); WRITE_CHAR('(');                           229   219
390   READ_IFL(LEN);                                                      184   387
391   WRITE_INT(LEN); WRITE_CHAR(',');                                    258   387   219
392   WRITE_CHAR8 ('HEX-VAL:');                                           229
393   NEXT_COL; N:=(LEN-1) DIV 2 + 1;                                     196   387   387
394   FOR I:=1 TO N DO BEGIN                                              387   387
395     READ_IFL(ARG);                                                    184   387
396     IF OUT_COL_PTR+OUT_BUF_PTR-1+9 > LINE_LENGTH THEN BEGIN           172   173   153
397       WRITE_BUF(FALSE); WRITE(NL); OUT_COL PTR:=1; NEXT_COL;          210   366   114    13   172   196
398     END;
399     WRITE_HEX(ARG,2);                                                 243   387
400     IF I < N THEN WRITE_CHAR(',') ELSE WRITE_CHAR(')');               387   387   219   219
401   END;
402   WRITE_BUF(FALSE);                                                   210   366
403 END;
404
405
406 "COMMON INITIALIZATION"
407
408 PROCEDURE INITIALIZE;                                                   *
409 BEGIN
410   WORDS_IN:=PAGELENGTH;                                               167    15
411   PAGES_IN:=1;                                                        169
412   PASS_NO:=PARAM[10].INT;                                             171   149    99
413   OUT_COL_PTR:=1;                                                     172
414   OUT_BUF_PTR:=1;                                                     173
415   FIRST_COL:=12;                                                      175
416   MIN_COL_SEP:=3;                                                     176
417   COL_SEP:=3;                                                         177
418   LINK := PARAM[2].PTR                                                179   149   101
419 END;
```

```
420
421 "WRITE PASS LISTING HEADER"
422
423 PROCEDURE WRITE_TEXT (TEXT: LINE);                                  *      *   146
424 VAR I: INTEGER;                                                     *      S
425 BEGIN
426   I:=1;                                                            424
427   WHILE TEXT[I] <> '$' DO BEGIN                                    423    424
428     WRITE(TEXT[I]); I:=I+1;                                        114    423    424    424    424
429   END;
430 END;
431
432 PROCEDURE WRITE_HEADER;                                             *
433 VAR I, J: INTEGER;                                                  *      *     S
434     C: CHAR;                                                        *      S
435 BEGIN
436   C:=CHR(PASS_NO+ORD('0'));                                        434    239    171    239
437   WRITE (FF);                                                      114     13
438   WRITE(NL);                                                       114     13
439   FOR I:=1 TO LINE_LENGTH DO WRITE(C);                             433    153    114    434
440   WRITE(NL);                                                       114     13
441   WRITE(C);                                                        114    434
442   J:=LINE_LENGTH DIV 2 - 17;                                       433    153
443   FOR I:=2 TO J-1 DO WRITE(' ');                                   433    433    114
444   WRITE_TEXT('MEPASCAL INTERMEDIATE CODE PASS $');                 423
445   WRITE(C);                                                        114    434
446   FOR I:=J+33 TO LINE_LENGTH-1 DO WRITE(' ');                      433    433    153    114
447   WRITE(C); WRITE(NL);                                             114    434    114     13
448   FOR I:=1 TO LINE_LENGTH DO WRITE(C);                             433    153    114    434
449   WRITE(NL);                                                       114     13
450 END;
451
452 PROCEDURE WRITE_5_CASE (OP_NAME: CHAR8);                            *      *   155
453 VAR
454    I: INTEGER;                                                      *      S
455    MIN,                                                             *
456    MAX,                                                             *
457    MAX_MINUS_MIN,                                                   *
458    LOCATION,                                                        *
459    STMT_LABEL:    INTEGER;                                          *      S
460
461 BEGIN
462 WRITE_CHAR8(OP_NAME);                                              229    452
463 WRITE_CHAR( '(' );                                                 219
464 READ_IFL(MIN);                                                     184    455
465 WRITE_INT(MIN);                                                    258    455
466 WRITE_CHAR( ',' );                                                 219
467 READ_IFL (MAX);                                                    184    456
468 WRITE_INT (MAX);                                                   258    456
469 MAX_MINUS_MIN := MAX - MIN;                                        457    456    455
470
471 FOR I := 0 TO MAX_MINUS_MIN DO                                     454    457
472    BEGIN
473    WRITE_CHAR(',');                                                219
474    READ_IFL(STMT_LABEL);                                           184    459
475    WRITE_INT(STMT_LABEL);                                          258    459
476    END;
477 WRITE_CHAR( ')' );  WRITE_BUF(TRUE)                                219    210    222
478 END;
479
```

```
480
481 PROCEDURE DUMP_TABLE (TABLE: TABLEPTR; ENTRIES:INTEGER);          *     *    60     *     S
482 CONST FLDSIZE = 6;                                                  *
483 VAR   I, J, K: INTEGER; PORTION: TABLEPTR;                          *     *     *     S     *    60
484       CONSTAB: BOOLEAN;                                             *     S
485 BEGIN
486 CONSTAB := TABLE = LINK@.TABLES@.CONSTTABLE;                       484   481   179    89    68
487 PORTION := TABLE;                                                  483   481
488 I := ENTRIES - MAXWORD;                                            483   481    55
489 WITH PORTION@ DO                                                   483
490    BEGIN
491    WHILE I>0 DO                                                    483
492       BEGIN
493       IF CONSTAB                                                   484
494       THEN FOR J := 1 TO MAXWORD DO                                483    55
495               BEGIN
496               K := CONTENTS[J];                                    483    63   483
497               WRITE_INTR(K DIV 256, FLDSIZE);                      271   483   482
498               WRITE_BUF(TRUE);                                     210   222
499               WRITE_INTR(K MOD 256, FLDSIZE);                      271   483   482
500               WRITE_BUF(TRUE);                                     210   222
501               END
502       ELSE FOR J := 1 TO MAXWORD DO                                483    55
503               BEGIN
504               WRITE_INTR(CONTENTS[J], FLDSIZE);                    271    63   483   482
505               WRITE_BUF(TRUE);                                     210   222
506               END;
507       I := I - MAXWORD;                                            483   483    55
508       PORTION := NEXTPORTION                                       483    62
509       END;
510    I := MAXWORD + I;  "I IS NEGATIVE AT THIS POINT"                483    55   483
511    IF CONSTAB                                                      484
512    THEN FOR J := 1 TO I DO                                         483   483
513            BEGIN
514            K := CONTENTS[J];                                       483    63   483
515            WRITE_INTR(K DIV 256, FLDSIZE);                         271   483   482
516            WRITE_BUF(TRUE);                                        210   222
517            WRITE_INTR(K MOD 256, FLDSIZE);                         271   483   482
518            WRITE_BUF(TRUE);                                        210   222
519            END
520    ELSE FOR J := 1 TO I DO                                         483   483
521            BEGIN
522            WRITE_INTR(CONTENTS[J], FLDSIZE);                       271    63   483   482
523            WRITE_BUF(TRUE);                                        210   222
524            END;
525    END;
526 END;
527
528
529 PROCEDURE PASS5;                                                    *
530 CONST MIN_OP5 = 0;     MAX_OP5 = 49;                                *     *
531 CONST
532    PASS5_TABLE = (                                                  *
533          'PUSHCNST', 1,      'PUSHVAR ', 3,      " 1"
534          'PUSHIND ', 1,      'PUSHADDR', 2,
535          'FIELD   ', 1,      'INDEX   ', 4,
536          'POINTER ', 0,      'VARIANT ', 2,
537          'RANGE   ', 2,      'ASSIGN  ', 1,      " 9"
538          'ASSGNTAG', 0,      'COPY    ', 1,
539          'NEW     ', 0,      'NOT     ', 0,
540          'AND     ', 1,     'OR      ', 1,
541          'NEG     ', 1,     'ADD     ', 1,
542          'SUB     ', 1,     'MUL     ', 1,      "19"
543          'DIV     ', 1,     'MOD     ', 1,
544          'INVALID ', 0,     'INVALID ', 0,
545          'FUNCTION', 2,     'BUILDSET', 0,
546          'COMPARE ', 2,     'CMPSTRUC', 2,
547          'FUNCVALU', 2,     'DEFLABEL', 1,      "29"
548          'JUMP    ', 1,     'FALSJUMP', 1,
549          'CASEJUMP', 0,     'INITVAR ', 0,
550          'CALL    ', 3,     'ENTER   ', 5,
551          'RETURN  ', 1,     'POP     ', 1,
552          'NEWLINE ', 1,     'ERROR   ', 0,      "39"
553          'LNGCONST', 0,     'MESSAGE ', 2,
554          'INCRMENT', 0,     'DECRMENT', 0,
555          'PROCDURE', 1,     'INIT    ', 4,
556          'PUSHLABL', 1,     'CALLPROG', 0,      "47"
557          'EOM     ', 1,     'DUP1TOS ', 0);     "49"
558
559          ARRAY[MIN_OP5 .. MAX_OP5] OF                           530   530
560          RECORD
561             OP_NAME: CHAR8;                                       *   155
562             NUM_ARGS: BYTE                                        *   248
563          END;
564
565 CONST
566    EOM5 = 48;                                                     *
567    NEWLINES = 38;                                                 *
568    CASEJUMP5 = 32;                                                *
569    LONGCONSTANT5 = 40;                                            *
570
571 BEGIN
572    IN_FILE := 2;                                                168
573    REPEAT
574       READ_IFL (OP);                                            184   178
575       WITH PASS5_TABLE[OP] DO                                   532   178
576       IF (OP < MIN_OP5) OR (OP > MAX_OP5)                       178   530   178   530
577       THEN INDEXERROR ('BAD_OP  ', OP)                          289   178
578       ELSE IF OP = NEWLINES                                     178   567
579            THEN NEW LINE                                        354
580            ELSE IF OP = CASEJUMP5                               178   568
581                 THEN WRITE_5_CASE (OP_NAME)                     452   561
582                 ELSE IF OP = LONGCONSTANT5                      178   569
583                      THEN WRITE_LCONST                          386
584                      ELSE WRITE_OP (OP_NAME, NUM_ARGS, 0, 0, 0, 0)    308   561   562
585    UNTIL OP = EOM5;                                             178   566
586    WRITE (NL)                                                   114    13
587 END;
588
589
590
591
592 PROCEDURE PASS6;                                                  *
593 CONST     MIN_OP6 = 0;    MAX_OP6 = 255;                          *     *
594 CONST   PASS6_TABLE = (                                           *
595
596 'SLDC00  ',  0,            'SLDC01  ',  0,          "001"
597 'SLDC02  ',  0,            'SLDC03  ',  0,
598 'SLDC04  ',  0,            'SLDC05  ',  0,          "005"
599 'SLDC06  ',  0,            'SLDC07  ',  0,
600 'SLDC08  ',  0,          'SLDC09  ',  0,
601 'SLDC10  ',  0,          'SLDC11  ',  0,
602 'SLDC12  ',  0,          'SLDC13  ',  0,
603 'SLDC14  ',  0,          'SLDC15  ',  0,        "015"
604 'SLDC16  ',  0,          'SLDC17  ',  0,
605 'SLDC18  ',  0,          'SLDC19  ',  0,
606 'SLDC20  ',  0,          'SLDC21  ',  0,
607 'SLDC22  ',  0,          'SLDC23  ',  0,
608 'SLDC24  ',  0,          'SLDC25  ',  0,        "025"
609 'SLDC26  ',  0,          'SLDC27  ',  0,
610 'SLDC28  ',  0,          'SLDC29  ',  0,
611 'SLDC30  ',  0,          'SLDC31  ',  0,
612 'SLDL01  ',  0,          'SLDL02  ',  0,
613 'SLDL03  ',  0,          'SLDL04  ',  0,        "035"
614 'SLDL05  ',  0,          'SLDL06  ',  0,
615 'SLDL07  ',  0,          'SLDL08  ',  0,
616 'SLDL09  ',  0,          'SLDL10  ',  0,
617 'SLDL11  ',  0,          'SLDL12  ',  0,
618 'SLDL13  ',  0,          'SLDL14  ',  0,        "045"
619 'SLDL15  ',  0,          'SLDL16  ',  0,
620 'SLD001  ',  0,          'SLD002  ',  0,
621 'SLD003  ',  0,          'SLD004  ',  0,
622 'SLD005  ',  0,          'SLD006  ',  0,
623 'SLD007  ',  0,          'SLD008  ',  0,        "055"
624 'SLD009  ',  0,          'SLD010  ',  0,
625 'SLD011  ',  0,          'SLD012  ',  0,
626 'SLD013  ',  0,          'SLD014  ',  0,
627 'SLD015  ',  0,          'SLD016  ',  0,
628 'INVAL064',  0,          'INVAL065',  0,        "065"
629 'INVAL066',  0,          'INVAL067',  0,
630 'INVAL068',  0,          'INVAL069',  0,
631 'INVAL070',  0,          'INVAL071',  0,
632 'INVAL072',  0,          'INVAL073',  0,
633 'INVAL074',  0,          'INVAL075',  0,        "075"
634 'INVAL076',  0,          'INVAL077',  0,
635 'INVAL078',  0,          'INVAL079',  0,
636 'INVAL080',  0,          'INVAL081',  0,
637 'INVAL082',  0,          'INVAL083',  0,
638 'INVAL084',  0,          'INVAL085',  0,        "085"
639 'INVAL086',  0,          'INVAL087',  0,
640 'INVAL088',  0,          'INVAL089',  0,
641 'INVAL090',  0,          'INVAL091',  0,
642 'INVAL092',  0,          'INVAL093',  0,
643 'INVAL094',  0,          'INVAL095',  0,        "095"
644 'INVAL096',  0,          'INVAL097',  0,
645 'INVAL098',  0,          'INVAL099',  0,
646 'INVAL100',  0,          'INVAL101',  0,
647 'INVAL102',  0,          'INVAL103',  0,
648 'INVAL104',  0,          'INVAL105',  0,        "105"
649 'INVAL106',  0,          'INVAL107',  0,
650 'INVAL108',  0,          'INVAL109',  0,
651 'INVAL110',  0,          'INVAL111',  0,
652 'INVAL112',  0,          'INVAL113',  0,
653 'INVAL114',  0,          'INVAL115',  0,        "115"
654 'INVAL116',  0,          'INVAL117',  0,
655 'INVAL118',  0,          'INVAL119',  0,
656 'SIND0   ',  0,          'SIND1   ',  0,
657 'SIND2   ',  0,          'SIND3   ',  0,
658 'SIND4   ',  0,          'SIND5   ',  0,        "125"
659 'SIND6   ',  0,          'SIND7   ',  0,
'LDCR    ',  1,          'LDCI    ',  1,
'LCA     ',  1,          'LDC     ',  2,
'LLA     ',  1,          'LDO     ',  1,
'LAO     ',  1,          'LDL     ',  1,          "135"
'LDA     ',  2,          'LOD     ',  2,
'UJP     ',  1,          'UJPL    ',  1,
'MPI     ',  0,          'DVI     ',  0,
'STM     ',  1,          'MODI    ',  0,
'CPL     ',  1,          'CPG     ',  1,          "145"
'CIP     ',  2,          'CXL     ',  2,
'CXG     ',  2,          'CXI     ',  3,
'RPU     ',  1,          'CPF     ',  0,
'LDCN    ',  0,          'LSL     ',  1,
'LDE     ',  2,          'LAE     ',  2,          "155"
'NOP     ',  0,          'LPR     ',  0,
'BPT     ',  0,          'RBP     ',  0,
'LOR     ',  0,          'LAND    ',  0,
'ADI     ',  0,          'SBI     ',  0,
'STL     ',  1,          'SRO     ',  1,          "165"
'STR     ',  2,          'LDB     ',  0,
'INVAL168',  0,          'INVAL169',  0,
'INVAL170',  0,          'INVAL171',  0,
'INVAL172',  0,          'INVAL173',  0,
'INVAL174',  0,          'INVAL175',  0,          "175"
'EQUI    ',  0,          'NEQI    ',  0,
'LEQI    ',  0,          'GEQI    ',  0,
'LEUSW   ',  0,          'GEUSW   ',  0,
'EQUPWR  ',  0,          'LEQPWR  ',  0,
'GEQPWR  ',  0,          'EQUBYT  ',  1,          "185"
'LEQBYT  ',  1,          'GEQBYT  ',  1,
'SRS     ',  0,          'SWAP    ',  0,
'TNC     ',  0,          'RND     ',  0,
'ADR     ',  0,          'SBR     ',  0,
'MPR     ',  0,          'DVR     ',  0,          "195"
'STO     ',  0,          'MOV     ',  1,
'DUP2    ',  0,          'ADJ     ',  1,
'STB     ',  0,          'LDP     ',  0,
'STP     ',  0,          'CHK     ',  0,
'FLT     ',  0,          'EQUREAL ',  0,          "205"
'LEQREAL ',  0,          'GEQREAL ',  0,
'LDM     ',  1,          'SPR     ',  0,
'EFJ     ',  1,          'NFJ     ',  1,
'FJP     ',  1,          'FJPL    ',  1,
'XJP     ',  1,          'IXA     ',  1,          "215"
'IXP     ',  2,          'STE     ',  2,
'INN     ',  0,          'UNI     ',  0,
'INT     ',  0,          'DIF     ',  0,
'SIGNAL  ',  0,          'WAIT    ',  0,
'ABI     ',  0,          'NGI     ',  0,          "225"
'DUP1    ',  0,          'ABR     ',  0,
'NGR     ',  0,          'LNOT    ',  0,
'IND     ',  1,          'INC     ',  1,
'*MESAGE*',  2,          '*EOM*   ',  0,
'*NEWLIN*',  0,          'INVAL235',  0,          "235"
'INVAL236',  0,          'INVAL237',  0,
'INVAL238',  0,          'INVAL239',  0,
'INVAL240',  0,          'INVAL241',  0,
'INVAL242',  0,          'INVAL243',  0,
'INVAL244',  0,          'INVAL245',  0,          "245"
'INVAL246',  0,          'INVAL247',  0.
```

```
720 'INVAL248',  0,              'INVAL249',  0,
721 'INVAL250',  0,              'INVAL251',  0,
722 'INVAL252',  0,              'INVAL253',  0,
723 'INVAL254',  0,              'INVAL255',  0):          "255"
724
725         ARRAY [MIN_OP6..MAX_OP6] OF                          593   593
726              RECORD
727              OP_NAME: CHAR8;                                   *   155
728              NUM_ARGS: BYTE;                                   *   248
729              END;
730
731
732 CONST EOM6 = 233;      UJPL = 139;      FJPL = 213;            *     *     *
733       NEWLINE6 = 234;                                          *
734       INTFLD = 6;                                              *
735
736 VAR  SEGS, SEGNO, SEGRTNS, ARG, I: INTEGER;                    *     *     *     *     *     S
737
738 BEGIN
739 "SCAN CONCURRENT SEGMENT"
740   IN_FILE:=1;                                                168
741   SKIP(1); WRITE_TEXT('** CONCURRENT SEGMENT **$'); SKIP(1); 372   423   372
742   REPEAT
743     READ_IFL(OP);                                            184   178
744     WITH PASS6_TABLE[OP] DO                                  594   178
745         IF (OP < MIN_OP6) OR (OP > MAX_OP6)                  178   593   178   593
746         THEN INDEXERROR ('BAD-OP  ', OP)                     289   178
747         ELSE IF OP = NEWLINE6                                178   733
748              THEN NEW_LINE                                   354
749              ELSE BEGIN
750                   WRITE_OP (OP_NAME, NUM_ARGS, 0, 0, 0, 0);  308   727   728
751                   IF (OP=UJPL) OR (OP=FJPL)                  178   732   178   732
752                   THEN BEGIN
753                        READ_IFL(ARG);                        184   736
754                        WRITE_CHAR8('*LOC=   ');              229
755                        WRITE_INT(ARG); WRITE_CHAR('*');      258   736   219
756                        WRITE_BUF(TRUE)                       210   222
757                        END
758                   END
759   UNTIL OP = EOM6;                                           178   732
760
761
762 "SCAN INTERFACE SEGMENT"
763 IN_FILE := 4;   WORDS_IN := PAGELENGTH;   PAGES IN := 1;     168   167    15   169
764 SKIP(2);                                                     372
765 WRITE_TEXT ('** INTERFACE SEGMENT(S) **$');                  423
766 SEGS := LINK@.INTERFACE@.INTERFACES;                         736   179    90    80
767 SKIP(0);                                                     372
768 IF SEGS = 0                                                  736
769 THEN WRITE_CHAR8('....NONE')                                 229
770 ELSE BEGIN WRITE_CHAR8('SEGMNTS='); WRITE_INT(SEGS) END;     229   258   736
771 SKIP(1);                                                     372
772 SEGNO := 1;                                                  736
773 WHILE SEGNO <= SEGS DO                                       736   736
774    BEGIN
775.   SEGRTNS := LINK@.INTERFACE@.INTERFACESIZES[SEGNO];        736   179    90    81   736
776    WRITE_TEXT('SEGMENT NO. $'); WRITE_INT(SEGNO);            423   258   736
777    WRITE_BUF(TRUE);  NEXT_COL;                               210   222   196
778    WRITE_TEXT('NO. OF ROUTINES: $'); WRITE_INT(SEGRTNS); SKIP(1);  423   258   736   372
779    READ_IFL(ARG); WRITE_CHAR8('SEGLEN= ');                   184   736   229
```

```
   WRITE_INT(ARG); WRITE_BUF(TRUE);                                          258   736   210   222
   READ_IFL(ARG); WRITE_CHAR8('MININDX=');                                   184   736   229
   WRITE_INT(ARG); WRITE_BUF(TRUE);                                          258   736   210   222
   READ_IFL(ARG); WRITE_CHAR8('MAXINDX=');                                   184   736   229
   WRITE_INT(ARG); WRITE_BUF(TRUE);                                          258   736   210   222
   SKIP(0);                                                                  372

   WRITE_TEXT('CASEJUMP OFFSETS:$'); WRITE_BUF(TRUE);                        423   210   222
   FOR J := 1 TO SEGRTNS DO                                                  736   736
      BEGIN
      READ_IFL(ARG);                                                         184   736
      WRITE_INT(ARG);  WRITE_BUF(TRUE)                                       258   736   210   222
      END;
   SKIP(0);                                                                  372

   READ_IFL(ARG); WRITE_CHAR8('EXIT-IC=');                                   184   736   229
   WRITE_INT(ARG); WRITE_BUF(TRUE); NEXT_COL;                                258   736   210   222   196
   READ_IFL(ARG); WRITE_CHAR8('DATASIZ=');                                   184   736   229
   WRITE_INT(ARG);                                                           258   736
   SKIP(0);                                                                  372

   FOR I := 1 TO (4 + 2*SEGRTNS + 2) DO                                      736   736
      BEGIN
      READ_IFL(OP);                                                          184   178
      WITH PASS6_TABLE[OP] DO                                                594   178
         IF (OP < MIN_OP6) OR (OP > MAX_OP6)                                 178   593   178   593
         THEN INDEXERROR ('BAD-OP  ', OP)                                    289   178
         ELSE WRITE_OP (OP_NAME, NUM_ARGS, 0, 0, 0, 0)                       308   727   728
      END;
   SKIP(0);                                                                  372

   READ_IFL(ARG); WRITE_CHAR8('RTN-PTR=');                                   184   736   229
   WRITE_INT(ARG); WRITE_BUF(TRUE);                                          258   736   210   222
   READ_IFL(ARG); WRITE_CHAR8('LO-ORDR=');                                   184   736   229
   WRITE_INT(ARG MOD 256); WRITE_BUF(TRUE); NEXT_COL;                        258   736   210   222   196
   WRITE_CHAR8('HI-ORDR='); WRITE_INT(ARG DIV 256);                          229   258   736

   SKIP(2);                                                                  372
   SEGNO := SUCC(SEGNO)                                                      736         736
   END;


WITH LINK@ DO                                                                179
   BEGIN

  "PRINT SOME OF THE PASSLINK FIELDS"
   SKIP(1); WRITE_TEXT('** PASSLINK FIELDS **$'); SKIP(0);                   372   423   372
   WRITE_CHAR8('LABELS  '); WRITE_INTR(LABELS, SFIELD);                      229   271    87   162
   SKIP(0);                                                                  372
   WRITE_CHAR8('BLOCKS  '); WRITE_INTR(BLOCKS, SFIELD);                      229   271    87   162
   SKIP(0);                                                                  372
   WRITE_TEXT('CONSTANTS$'); WRITE_INTR(CONSTANTS, SFIELD);                  423   271    87   162
   WRITE_CHAR(' '); WRITE_CHAR8('WORDS   '); SKIP(0);                        219   229   372
   WRITE_TEXT('XJP_OFFSETS$'); WRITE_INTR(XJP_OFFSETS, SFIELD);              423   271    87   162
   WRITE_CHAR(' '); WRITE_CHAR8('WORDS   '); SKIP(0);                        219   229   372
   WRITE_TEXT('SEGDISTANCE$');WRITE_INTR(TABLES@.SEGDISTANCE, SFIELD);       423   271    89    67   162
   WRITE_CHAR(' '); WRITE_CHAR8('BYTES   ');                                 219   229

   "PRINT CONTENTS OF THE TABLES PASSED THRU HEAP"
   SKIP(2); WRITE_TEXT('** TABLES **$');                                     372   423
```

```
840
841    SKIP(1); WRITE_CHAR8('CONSTNTS'); SKIP(0);                     372   229   372
842    IF CONSTANTS > 0                                                87
843    THEN DUMP_TABLE(TABLES@.CONSTTABLE, CONSTANTS)                 481    89    68    87
844    ELSE WRITE_CHAR8('....NONE');                                  229
845
846    SKIP(2); WRITE_CHAR8('XJPTABLE'); SKIP(0);                     372   229   372
847    IF XJP_OFFSETS > 0                                              87
848    THEN DUMP_TABLE (TABLES@.XJPTABLE, XJP_OFFSETS)                481    89    68    87
849    ELSE WRITE_CHAR8('....NONE');                                  229
850
851    SKIP(2);  WRITE_CHAR8('JUMPTABL'); SKIP(0);                    372   229   372
852    IF LABELS > 0                                                   87
853    THEN DUMP_TABLE (TABLES@.JUMPTABLE, LABELS)                    481    89    68    87
854    ELSE WRITE_CHAR8('....NONE');                                  229
855
856    SKIP(2); WRITE_CHAR8('EXIT-IC '); SKIP(0);                     372   229   372
857    IF BLOCKS > 0                                                   87
858    THEN DUMP_TABLE (TABLES@.EXITICTABLE, BLOCKS)                  481    89    69    87
859    ELSE WRITE_CHAR8('....NONE');                                  229
860
861    SKIP(2); WRITE_CHAR8('DATASIZE'); SKIP(0);                     372   229   372
862    IF BLOCKS > 0                                                   87
863    THEN DUMP_TABLE (TABLES@.DATASIZETABLE, BLOCKS)                481    89    69    87
864    ELSE WRITE_CHAR8('....NONE');                                  229
865    END;
866
867 FOR I := 1 TO PAGELENGTH + 10 DO WRITE_CHAR(' ')                  736    15   219
868 END;
869
870
871
872
873
874 PROCEDURE PASS7;                                                    *
875 CONST
876 MIN_OP7 = 1;      MAX_OP7 = 1;                                      *     *
877
878 PASS7_TABLE = ('NULLOP  ', 0):                                      *
879
880         ARRAY [MIN OP7..MAX_OP7] OF                               876   876
881              RECORD
882              OP_NAME: CHAR8;                                        *   155
883              NUM_ARGS: BYTE;                                        *   248
884              END;
885
886 VAR
887 I: INTEGER;                                                         *     S
888
889 BEGIN
890    IN_FILE := 3;                                                  168
891    COL_SEP := 1;                                                  177
892    FOR I := 1 TO PAGELENGTH +10 DO WRITE_CHAR(' ');               887    15   219
893 END;
894
895
896
897 BEGIN
898   INITIALIZE;                                                     408
899   WRITE_HEADER;                                                   432
```

```
900   CASE PASS_NO OF                                              171
901     1:"PASS1";
902     2:"PASS2";
903     3:"PASS3";
904     4:"PASS4";
905     5: PASS5;                                                  529
906     6: PASS6;                                                  592
907     7: PASS7;                                                  874
908     8, 9: "NOT IMPLEMENTED";
909   END;
910 END.
```

```
CROSS REFERENCE   * IS DEF   = IS ASG

 -A-
 A             259*  264=  268   273*  278=  282=  285
 ABS           264   278
 ACCEPT        126*
 ADDR           31*
 ALIGN         210*  213
 ARG            50*  133*  134*  184*  191=  355*  358   365   387*  395   399   736*  753   755   779   780   781   782
               783   784   790   791   795   796   797   798   811   812   813   814   815
 ARGLIST       105*  145   149
 ARGSEQ        107*  133   134
 ARGTAG         93*   97
 ARGTYPE        96*  105   133   134
 ARG_NO        310*  313=  315   320   326   332   338
 ARG_VAL       310*  318   322   322   323   328   328   329   334   334   335   340   340   341   344   344   345   346
 ASCII          27
 ATTR          136*

 -B-
 BACKSPACE      41
 BADINDEX      289*  300
 BLOCK         118*  119*  129*  130*  139*
 BLOCKS         87*  829   857   858   862   863
 BOOL           98
 BOOLEAN        32    98   116   129   130   136   210   484
 BOOLTYPE       94    98
 BYTE          248   562   728   883
 BYTES         248   253   254
 BYTES_PER_IN   17*

 -C-
 C             113*  114*  126*  127*  219*  223   434*  436=  439   441   445   447   448
 CALLERROR     111
 CARDDEVICE     37
 CASEJUMPS     568*  580
 CHAR           20    23   113   114   126   127   155   219   259   273   434
 CHAR8         155*  229   289   308   452   561   727   882
 CHR           239   240   264   278   436
 CLOSE         117*
 CODELIMIT     111
 COL_SEP       177*  417=  891=
 COMPLETE       44
 CONCODE        27
 CONSTAB       484*  486=  493   511
 CONSTANTS      87*  831   842   843
 CONSTTABLE     68*  486   843
 CONTENTS       63*  496   504   514   522
 CONTEXT_ARG   309*  338
 CONTROL        39
 CR             13*

 -D-
 DATASIZETABL   69*  863
 DEVICE        139*  141*
 DISKDEVICE     37
 DISPLAY       127*
 DUMP_TABLE    481*  843   848   853   858   863
```

```
-E-
 EM              13*
 EMPTY           27
 ENDFILE         45
 ENDMEDIUM       45
 ENTRIES        481*  488
 EOF            129*  130*
 EOM5           566*  585
 EOM6           732*  759
 EXITICTABLE     69*  858

-F-
 F              116*  117*  118*  119*  120*
 FAILURE         44
 FALSE          366   368   379   397   402
 FF              13*  437
 FILE            25*  116   117   118   119   120
 FILEATTR        29*  136
 FILEKIND        27*   30
 FIRST_COL      175*  203   415=
 FJPL           732*  751
 FLDSIZE        482*  497   499   504   515   517   522
 FOUND          116*  136*
 FULLWORD        10*   88

-G-
 GET            118*  187

-H-
 HEADER         125*
 HEAPLIMIT      111

-I-
 I              197*  199=  200   203=  206   207   211*  214=  214   230*  232=  233   234   245*  252=  253   254   260*
                262=  264   265=  265   268   274*  276=  278   279=  279   282   282=  282   284   285   290*  310*  373*
                377=  387*  394=  400   424*  426=  427   428   428=  428   433*  439=  443=  446=  448=  454*  471=  483*
                488=  491   507=  507   510=  510   512   520   736*  788=  801=  867=  887*  892=
 ID             100   116*  136*  145*
 IDENTIFIER      23*  100   116   136   145
 IDENTIFY       125*
 IDLENGTH        22*   23
 IDTYPE          94   100*
 IFINFO          78    79*
 IFPTR           78*   90
 INDEXERROR     289*  323   329   335   341   577   746   806
 INITIALIZE     408*  898
 INP            107
 INPUT           39
 INPUTTASK       53
 INT             99   247   251=  412
 INTEGER         10    11*   16    31    33    58    63    67    80    81    87    99   118   119   120   122   123   146
                167   168   169   171   172   173   175   176   177   178   184   197   211   230   237   243   243   245
                247   258   260   271   271   274   289   290   309   310   355   372   373   387   424   433   454   459
                481   483   736   887
 INTERFACE       90*  766   775
 INTERFACES      80*  766
 INTERFACESIZ    81*  775
 INTERVENTION    44
 INTFLD         734*
 INTTYPE         94    99*
 IN_FILE          168*  187   572=  740=  763=  890=
 IOARG             41*   50
 IODEVICE          36*  139   141
 IOMOVE           141*
 IOOPERATION       39*   48
 IOPARAM           47*  139   141
 IORESULT          43*   49
 IOTRANSFER       138*

-J-
 J                197*  206=  260*  268=  268   274*  284=  285=  285   433*  442=  443   446   483*  494=  496   502=  504
                  512=  514   520=  522
 JOBTASK           53
 JUMPTABLE         68*  853

-K-
 K                483*  496=  497   499   514=  515   517
 KIND              30*
 KIND_ARG         308*  320

-L-
 LABELS            87*  827   852   853
 LEN              387*  390   391   393
 LENGTH           120*
 LFIELD           162*
 LINE              20*  125   131   132   146*  174   423
 LINELENGTH        19*   20
 LINES            372*  377
 LINE_LENGTH      153*  200   221   396   439   442   446   448
 LINK             179*  418=  486   766   775   822
 LOCATION         458*
 LONGCONSTANT     569*  582
 LOOKUP           136*

-M-
 MARK             122*
 MAX              456*  467   468   469
 MAXARG           104*  105
 MAXINTFAC         76*   81
 MAXWORD           55*   63   488   494   502   507   510
 MAX_CONTEXT      161*  340
 MAX_KIND         158*  322
 MAX_MINUS_MI     457*  469=  471
 MAX_MODE         160*  334
 MAX_OP5          530*  559   576
 MAX_OP6          593*  725   745   805
 MAX_OP7          876*  880
 MAX_TYPE         159*  328
 MEMNEM           149*
 MIN              455*  464   465   469
 MIN_COL_SEP      176*  199   416=
 MIN_CONTEXT      161*  340
 MIN_KIND         158*  322
 MIN_MODE         160*  334
 MIN_OP5          530*  559   576
 MIN_OP6          593*  725   745   805
 MIN_OP7          876*  880
 MIN_TYPE         159*  328
 MODE_ARG         309*  332
 MOVE              39
```

```
-N-
 N              243*  252   271*  284   387*  393=  394   400
 NEWLINE5       567*  578
 NEWLINE6       733*  747
 NEW_LinE       354*  579   748
 NEXTPORTION     62*  508
 NEXT_COL       196*  213   393   397   777   796   814
 NILTYPE         94    98
 NL              13*  202   359   360   376   377   397   438   440   447   449   586
 NOTUSED         33*
 NUM_ARGS       308*  313   348   562*  584   728*  750   807   883*

-O-
 OP             178*  574   575   576   576   577   578   580   582   585   743   744   745   745   746   747   751   751
                759   803   804   805   805   806
 OPEN           116*
 OPERATION       48*
 OPTION          74*   86
 OPTIONS         86*
 OP_NAME        308*  312   452*  462   561*  581   584   727*  750   807   882*
 ORD            239   240   264   278   436
 OUT            107
 OUTPUT          39
 OUTPUTTASK      53
 OUT_BUF        174*  214   223=
 OUT_BUF_PTR    173*  200   214   215   216=  221   223   224=  224   396   414=
 OUT_COL_PTR    172*  199   204=  206   207=  215=  215   359   361=  375   378=  396   397=  413=
 OVERFLOW       110

-P-
 P              118*  119*
 PAGE            16*  118   119   129   130   139   170
 PAGELENGTH      15*   16   186   410   763   867   892
 PAGES_IN       169*  187   188=  188   411=  763=
 PAGE_IN        170*  187   191
 PARAM          139*  141*  145*  149*  412   418
 PASS5          529*  905
 PASS5_TABLE    532*  575
 PASS6          592*  906
 PASS6_TABLE    594*  744   804
 PASS7          874*  907
 PASS7_TABLE    878*
 PASSLINK        84    85*
 PASSPTR         84*  101   179
 PASS_NO        171*  357   362   412=  436   900
 POINTER         58*
 POINTERERROR   110
 PORTION        483*  487=  489   508=
 PRINTDEVICE     37
 PROGRESULT     109*  146
 PROTECTED       32*
 PTR            101   418
 PTRTYPE         94   101*
 PUT            119*

-R-
 RANGEERROR     110
 READ           113*
 READARG        133*
```

```
READLINE         131*
READPAGE         129*
READ_IFL         184*  318   358   390   395   464   467   474   574   743   753   779   781   783   790   795   797   803
                 811   813
RELEASE          123*
REM              260*  262=  264   265=  265   266   274*  276=  278   279=  279   280
RESETPOINT        88*
RESULT           146*
REWIND            41
RUN              145*

-S-
S                133*  134*
SCRATCH           27
SEGDISTANCE       67*  835
SEGNO            736*  772=  773   775   776   818=  818
SEGRTNS          736*  775=  778   788   801
SEGS             736*  766=  768   770   773
SEQCODE           27
SFIELD           162*  827   829   831   833   835
SHORTINTEGER      11
SKIP             372*  741   741   764   767   771   778   785   793   799   809   817   826   826   828   830   832   834
                 839   841   841   846   846   851   851   856   856   861   861
STACKLENGTH       67*
STACKLIMIT       111
STARTMEDIUM       45
STATUS            49*
STMT_LABEL       459*  474   475
SUCC             818

-T-
TABLE             60    61*  481*  486   487
TABLEPART         66*   72
TABLEPTR          60*   62    69   481   483
TABLES            89*  486   835   843   848   853   858   863
TABLESPTR         72*   89
TAG               97*
TAG1             244   247*
TAG2             244   248*
TAGS             244*  246*
TAPEDEVICE        37
TASK             143*
TASKKIND          53*  143
TERMINATED       110
TEXT             131*  132*  229*  233   234   289*  297   423*  427   428
TIMELIMIT        111
TOP              122*  123*
TRANSMISSION      44
TRUE             222   349   477   498   500   505   516   518   523   756   777   780   782   784   787   791   796   812
                 814
TYPEDEVICE        37
TYPE_ARG         308*  326

-U-
UJPL             732*  751
UPSPACE           41

-V-
VAL              237*  239   239   240   243*  251   258*  262   267   271*  276   281
VARIANTERROR     110
```

```
-W-
 WORDS_IN      167*  186   188=  190=  190   191   410=  763=
 WRITE         114*  202   206   214   397   428   437   438   439   440   441   443   445   446   447   447   448   449
               586
 WRITEARG      134*
 WRITEEOF       41
 WRITELINE     132*
 WRITEPAGE     130*
 WRITE_5_CASE  452*  581
 WRITE_BUF     210*  222   349   366   368   379   397   402   477   498   500   505   516   518   523   756   777   780
               782   784   787   791   796   812   814
 WRITE_CHAR    219*  234   239   240   267   268   284   285   294   298   299   301   302   316   317   348   359   360
               376   377   389   391   400   400   463   466   473   477   755   832   834   836   867   892
 WRITE_CHAR8   229*  292   293   295   296   297   312   324   330   336   342   364   389   392   462   754   769   770
               779   781   783   795   797   811   813   815   827   829   832   834   836   841   844   846   849   851
               854   856   859   861   864
 WRITE_HEADER  432*  899
 WRITE_HEX     243*  345   399
 WRITE_HEX_CH  237*  253   254
 WRITE_INT     258*  300   346   391   465   468   475   755   770   776   778   780   782   784   791   796   798   812
               814   815
 WRITE_INTR    271*  365   497   499   504   515   517   522   827   829   831   833   835
 WRITE_LCONST  386*  583
 WRITE_OP      308*  584   750   807
 WRITE_TEXT    423*  444   741   765   776   778   787   826   831   833   835   839

-X-
 X             246*  251   253   254
 XJPTABLE       68*  848
 XJP_OFFSETS    87*  833   847   848

END XREF  265 IDENTIFIERS  1174 TOTAL REFERENCES
 250 COLLISIONS.
```

DISKETTE BLOCK DUMP PROGRAM

Some time after the arrival of the Microengine hardware it became apparent that some aspects of the machine were not clearly (if at all) described in the accompanying documents, and could be determined only by direct inspection of disk blocks. The installed system contains a disk-dumping utility (PATCH), but it is inconvenient to use. PATCH displays the contents of disk blocks as hexadecimal numbers or in a MIXED format where non-printing characters are shown in hex and the others are shown in their character representation. This was unsatisfactory since what was needed was a format which shows both the numerical and character representation of each byte. BLOCKDUMP does just that.

The program is written in UCSD Pascal and allows inspection of any block on the disk. The user is prompted for five items of information:

1) Physical disk drive unit number-- 4 and 5 are currently valid;
2) Starting disk block number-- the number of the block where the dump is to begin. The program assumes that a single-sided, double-density, eight-inch disk is mounted, so numbers between 0 and 987 are valid.
3) Number of blocks to be displayed-- must be such that block numbers greater than 987 will not be accessed. A nonpositive entry will terminate the program after the fifth piece of information has been entered.
4) Number base-- number base of the values to be displayed. The characters H (Hex) and D (Decimal) are valid.
5) Output device-- the device where the dump should be written. Valid values are the characters S (CRT Screen), P (Printer),

and Q (Quit). Entering 'Q' terminates the program.

On the printer, one block is printed per page. On the CRT, roughly one half of a block is displayed at a time. When the screen has been filled with information, typing a carriage return moves to the next screenful of information. After all the user information has been entered, there is no way to terminate the program before all the blocks entered as item 3 above have been displayed. This program was especially useful in discovering the format of Microengine code files.

The source code for BLOCKDUMP follows.

```
    (* LPRINTER: *)
PROGRAM BLOCKDMP;
(***********************************************************************)
(*                                                                    *)
(* PROGRAM TO DISPLAY (ON SCREEN OR PRINTER) THE CON-                 *)
(* TENTS OF DISK BLOCKS. USER TYPES IN THE UNIT NUMBER                *)
(* OF THE DISK TO BE DUMPED, STARTING BLOCK NUMBER,                   *)
(* NUMBER OF BLOCKS TO BE DISPLAYED, AND WHERE OUTPUT                 *)
(* IS TO BE DIRECTED. CONTENTS OF EACH BLOCK ARE DIS-                 *)
(* PLAYED AS ASCII VALUES UNDER THE CHARACTER REP-                    *)
(* RESENTATIONS. BYTE ADDRESS (RELATIVE TO THE START OF               *)
(* THE BLOCK) OF THE FIRST BYTE ON THE LINE APPEARS TO                *)
(* THE LEFT OF THE ASCII VALUES.  .                                   *)
(*                                                                    *)
(***********************************************************************)
CONST
   CR = 13;
   LF = 10;
   BLLENGTH = 512;  (* BYTES IN DISK BLOCK *)
   LINELEN  =  80;  (* CHARS PER LINE *)
   MAXLINES =  22;  (* LINES PER SCREEN PAGE *)
   MAXBLOCK = 987;  (* MAX BLOCK # ON 8" DUAL DENS DISK *)

TYPE
   LINE = PACKED ARRAY[1..LINELEN] OF CHAR;

VAR
   QUIT, OK, PRTOPT, CONOPT: BOOLEAN;
   DISK, STARTBL, LENGTH, BLOCK,
      POSIT, CHORD, I, BYTE, LINES: INTEGER;
   OPT, CHARAC, CONTINUE, BASE: CHAR;
   OUTFLNAM: STRING[10];
   OUTFL: INTERACTIVE;
   BLOCKIN: PACKED ARRAY[1..BLLENGTH] OF CHAR;
   CHARLINE, NUMLINE: LINE;
   X: REAL;
```

```
PROCEDURE WRITENUM (VALUE: INTEGER; BASE: CHAR);
(* ROUTINE PUTS CHARACTER REPRESENTATION OF THE
   CONTENTS OF A BYTE INTO 'CHARLINE'. ALWAYS
   PUTS IN THREE DIGITS.                      *)
CONST
   ORD0 = 48;
   ORDA = 65;

VAR
   TEMP: INTEGER;

BEGIN
IF BASE = 'H'
THEN BEGIN
   TEMP := VALUE DIV 256;
   IF TEMP = 0
   THEN NUMLINE[POSIT] := ' '
   ELSE IF TEMP <= 9
        THEN NUMLINE[POSIT] := CHR(TEMP + ORD0)
        ELSE NUMLINE[POSIT] := CHR(TEMP-10 + ORDA);
   POSIT := SUCC(POSIT);
   TEMP := (VALUE MOD 256) DIV 16;
   IF TEMP <= 9
   THEN NUMLINE[POSIT] := CHR(TEMP + ORD0)
   ELSE NUMLINE[POSIT] := CHR(TEMP-10 + ORDA);
   POSIT := SUCC(POSIT);
   TEMP := VALUE MOD 16;
   IF TEMP <= 9
   THEN NUMLINE[POSIT] := CHR(TEMP + ORD0)
   ELSE NUMLINE[POSIT] := CHR(TEMP-10 + ORDA);
   POSIT := SUCC(POSIT)
   END
ELSE BEGIN
   NUMLINE[POSIT] := CHR((VALUE DIV 100) + ORD0); (* 100'S *)
   POSIT := SUCC(POSIT);
   NUMLINE[POSIT] := CHR(((VALUE MOD 100) DIV 10) + ORD0); (* 10'S *)
   POSIT := SUCC(POSIT);
   NUMLINE[POSIT] := CHR((VALUE MOD 10) + ORD0); (* UNITS *)
   POSIT := SUCC(POSIT)
   END
END;
```

```
 BEGIN
 QUIT := FALSE;    DISK := 0;
 PRTOPT := FALSE; CONOPT := FALSE;
 REPEAT
    WRITELN ('ENTER UNIT # (4 OR 5)');
    READLN(DISK)
 UNTIL (DISK=4) OR (DISK=5);
 REPEAT
    WRITELN ('STARTING BLOCK? (0 <= BLOCK <= ', MAXBLOCK, ')');
    READLN(STARTBL)
 UNTIL (STARTBL>=0) AND (STARTBL<=MAXBLOCK);
 REPEAT
    WRITELN ('# OF BLOCKS TO DISPLAY?');
    READLN (LENGTH);
    IF STARTBL+LENGTH-1 >MAXBLOCK
    THEN WRITELN ('FINAL BLOCK WILL BE OFF THE DISK')
 UNTIL STARTBL+LENGTH-1 <= MAXBLOCK;
 IF LENGTH<=0 THEN QUIT := TRUE;
 REPEAT
    WRITELN ('DECIMAL OR HEX? (D OR H)');
    READLN (BASE)
 UNTIL (BASE = 'D') OR (BASE = 'H');
 OK := FALSE;
 WRITELN ('PRINTER OR SCREEN OUTPUT, OR QUIT?');
 REPEAT
    WRITELN ('ENTER P, S, OR Q');    READLN(OPT).
    CASE OPT OF
    'P': BEGIN
         OK := TRUE;    PRTOPT := TRUE;    OUTFLNAM := 'PRINTER:'
         END;
    'S': BEGIN
         OK := TRUE;    CONOPT := TRUE;    OUTFLNAM := 'CONSOLE:';
         WRITELN; WRITELN;
         WRITELN ('CARRIAGE RETURN CONTINUES PROGRAM WHEN SCREEN FULL.');
         FOR I := 1 TO 5000 DO X := 1.1*2.2*3.3*4.4*5.5
         END;
    'Q': BEGIN
         OK := TRUE;    QUIT := TRUE
         END
   END
UNTIL OK;
```

```
IF NOT QUIT
THEN BEGIN
   RESET (OUTFL, OUTFLNAM);
   NUMLINE[4] := ' ';
   NUMLINE[5] := ' ';
   FOR POSIT := 1 TO LINELEN DO CHARLINE[POSIT] := ' ';
   FOR BLOCK := STARTBL TO (STARTBL+LENGTH-1) DO
      BEGIN
      UNITWAIT (DISK);
      UNITREAD (DISK, BLOCKIN, BLLENGTH, BLOCK);
      WRITELN (OUTFL);
      WRITELN (OUTFL, 'UNIT # ', DISK, '      BLOCK # ', BLOCK,
                      '      BASE ', BASE);
      WRITELN (OUTFL);
      POSIT := 1;   (* ADDRESS STARTS IN COL 1 *)
      WRITENUM (0, BASE); (* FILL IN BYTE ADDRESS *)
      POSIT := 6;   (* VALUES START IN COL 6 *)
      LINES := 3;   (* 3 LINES DISPLAYED SO FAR *)
      FOR BYTE := 1 TO BLLENGTH DO (* FOR EVERY BYTE IN BLOCK *)
         BEGIN
         CHARAC := BLOCKIN[BYTE]; (* UNPACK BYTE & GET VALUE *)
         CHORD := ORD(CHARAC);
         IF (CHORD<=31) OR (CHORD>=126) (* IF NOT PRINTABLE *)
         THEN CHARAC := '.';
         CHARLINE[POSIT+2] := CHARAC; (* R. JUST. CHAR OVER VALUE *)
         WRITENUM (CHORD, BASE);   (* PUT VALUE IN NUMLINE *)
         IF POSIT >= LINELEN (* IF LINE FULL THEN WRITE IT OUT *)
         THEN BEGIN          (* CENTRONICS DOESN'T DO AUTO LF-CR *)
            WRITE (OUTFL, CHARLINE);
            IF PRTOPT THEN WRITE (OUTFL, CHR(CR));
            WRITE (OUTFL, NUMLINE);
            IF PRTOPT THEN WRITE (OUTFL, CHR(CR), CHR(LF));
            POSIT := 1;
            LINES := LINES + 2; (* 2 MORE LINES PRINTED *)
            IF (CONOPT) AND (LINES >= MAXLINES)THEN
               BEGIN (* IF SCREEN FULL, WAIT FOR CAR. RET. *)
               LINES := 0;
               READLN (OUTFL, CONTINUE)
               END;
            WRITENUM (BYTE, BASE);(*FILL IN ADDR. OF NEXT BYTE*)
            POSIT := 6
                 END
        END;
      FOR I := POSIT TO LINELEN DO
         BEGIN     (* LAST LINE ISN'T FULL *)
         CHARLINE[I] := ' ';
         NUMLINE[I] := ' '
         END;
      WRITE (OUTFL, CHARLINE);
      IF PRTOPT THEN WRITE (OUTFL, CHR(CR));
      WRITE (OUTFL, NUMLINE);
      IF PRTOPT
      THEN BEGIN
         WRITE (OUTFL, CHR(CR));
         WRITE (OUTFL, CHR(CR));
         END;
      IF CONOPT THEN READLN (OUTFL, CONTINUE);
      END;
   CLOSE (OUTFL)
   END
END.
```

INTERDATA 8/32 TO MICROENGINE FILE TRANSFER PROGRAM

MEPASCAL runs on an Interdata 8/32 and never has direct contact with the Microengine. A one-way file transfer system was written to move files from the 8/32 to the Microengine. The sending program, FILEXFER, was written by Robert Young. The system uses a primitive protocol to transfer 512-byte data blocks over an asynchronous 4800 baud communication link. The UCSD Pascal program XFER is the implementation of the Microengine (receiving) side of that protocol.

The unit of transfer is a 516-byte packet in which the first 512 bytes contain the data. Byte 514 is always zero, except to indicate end-of-transmission, which is denoted by the value 7 (ASCII EOT). The packet which contains the end-of-transmission indicator is sent <u>after</u> the last packet of good <u>data</u>, so its data portion contains garbage and can be ignored. Byte 516 is a checksum on the <u>data bytes only</u>. The formula is the sum of the data bytes, modulus 256:

SUM := (SUM + BYTE) MOD 256.

Bytes 513 and 515 are unused.

The Microengine side of the protocol proceeds as follows. An ENQ character (ASCII 5) is sent to indicate that the Microengine is ready to receive. The input status bit for the port is polled until a character is received, or a timeout occurs. The Microengine is expecting to receive a packet at this point. Since there could be quite a long wait between packets if the 8/32 is busy, the timeout period is quite long (a minute or more). This long period also gives the operator time to start FILEXFER on the 8/32. After the first character of the packet has arrived, the timeout period is made much shorter. The balance of the packet should be received within a few seconds of the first character. Again, the arrival of characters is sensed by polling the status bits of the port. The number of characters received (less one) is displayed on the console, regardless of whether there is a timeout or not. So, if the entire packet is received, the number 515 is displayed. If the entie packet is received and there is no checksum error, an acknowledgement (ACK-- ASCII 6) is sent to the 8/32, otherwise a negative acknowledgement (NAK-- ASCII 21) is returned. The response consists of a single character, and is not sent until an ENQ is received, indicating that the 8/32 is ready to receive it. If the 8/32 receives a NAK, it will retransmit the packet, otherwise a new one will be sent. The Microengine sends an ENQ when it is ready for the next packet, as above.

Figure 43 shows the general format of the buffer into which the incoming data is placed. The buffer is large enough to hold 60 disk blocks of data (512 bytes each) plus four bytes. The extra four bytes are for the last four bytes of the 60th block. As characters come off the line they are placed in the input buffer. Once the checksum has been recalculated and it has been determined that the packet was received correctly, the last four bytes are unneeded, so they are overlaid by the data of the next packet. Disk accesses are quite slow on the Microengine and the large buffer size keeps the number of them to a minimum. An attempt was made to reserve a larger buffer by using negative and positive array bounds centered on zero, but the UCSD compiler crashed during the compilation. The present size has worked well, and as it is, it consumes nearly half of main storage.

Since the hardware does not yet support interrupts and does not have a timer, timeouts are calculated by decrementing a counter while polling the port's status bit.

At the present time, there is no port on the 8/32 dedicated to

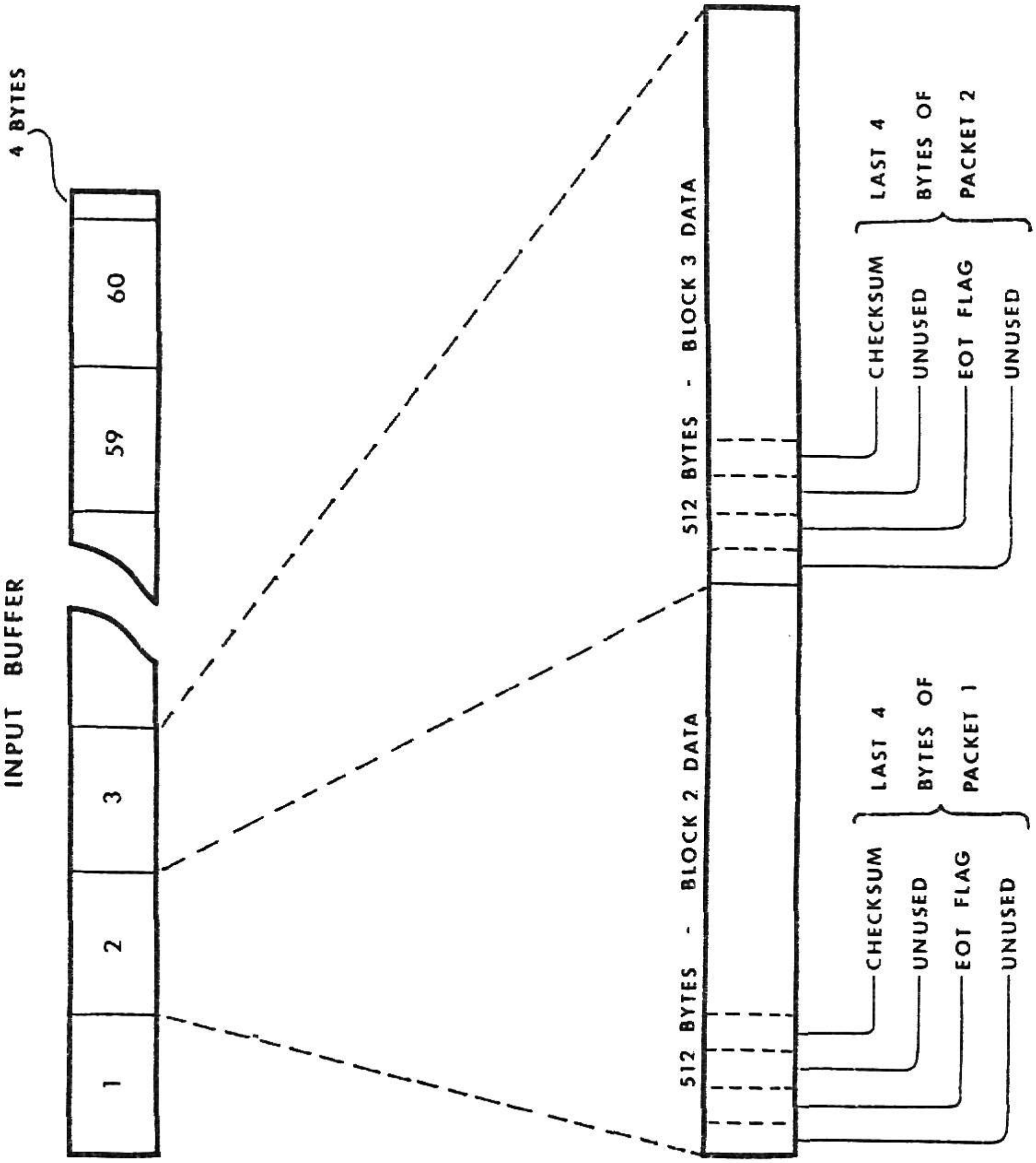


FIGURE 43. Layout of the input buffer in XFER program. The last four bytes of a packet extend into the space for the data of the next block, and a data block overlays the last four bytes of the previous packet.

communication with the Microengine. This means that two plugs must be switched, and a new baud rate set on one of the ports. Unfortunately, this makes two terminals on the 8/32 unusable. Hopefully, the situation can be changed in the near future.

The file transfer system has exhibited a behavior which so far has been unexplainable. On a regular basis a small, variable number of characters never reach the Microengine. For example, in every fifth packet, from six to thirty characters are lost. This is merely an inconvenience, since the faulty packets are retransmitted and the file eventually arrives intact. Although this does slow things up, a 50K byte file can still be transferred in less than fifteen minutes.

The source code for both the sending and receiving programs follows.

```
 1 "FILEXFER - ASC LINE FILE TRANSFER PROGRAM"
 2 "WRITTEN BY ROBERT YOUNG"
 3
 4 "$INCLUDE SVC_TYPES"
 5 "$INCLUDE FULL PREFIX"
 6
 7 PROGRAM FILEXFER;                                      *
 8 CONST EOT=7; ACK=6; NAK=21;                            *    *    *
 9 TYPE FILE_MODES = (BYTE_MODE, PAGE_MODE, ASC_MODE);    *    *    *    *
10 TYPE PAGE = ARRAY [1..512] OF BYTE;                    *    *
11
12 VAR BUF: RECORD                                        *
13       DATA: PAGE;                                      *    10
14       EOF: SHORTINTEGER;                               *    *
15       CKSUM: SHORTINTEGER;                             *    14
16     END;
17 IN_MODE, OUT_MODE: FILE_MODES;                         *    *    9
18 EM FOUND: BOOLEAN;                                     *    S
19 BLOCKS: INTEGER;                                       *    S
20
21 PROCEDURE SVC1 (VAR PARM: SVC1 BLOCK); EXTERN;         *    *    *
22 PROCEDURE SVC7 (VAR PARM: SVC7 BLOCK); EXTERN;         *    *    *
23 PROCEDURE SVC2PAUS; EXTERN;                            *
24
25 "DISPLAY ROUTINES"
26
27 PROCEDURE DISPLAY_TEXT (TEXT: LINE);                   *    *    *
28 VAR I: INTEGER;                                        *    S
29 BEGIN
30   I:=1;                                                28
31   WHILE TEXT[I]<>'$' DO BEGIN                          27   28
32     DISPLAY(TEXT[I]); I:=I+1;                          *    27   28   28   28
33   END;
34 END;
35
36 PROCEDURE DISPLAY_HEX_CHAR (VAL: INTEGER);             *    *    S
37 BEGIN
38   IF VAL>9 THEN DISPLAY(CHR(VAL-10+ORD('A')))          36   32   *    36   *
39     ELSE DISPLAY(CHR(VAL+ORD('0')));                   32   38   36   38
40 END;
41
42 PROCEDURE DISPLAY_HEX_CHARS (VAL: INTEGER);            *    *    S
43 BEGIN
44   DISPLAY_HEX_CHAR(VAL DIV 16);                        42   42
45   DISPLAY_HEX_CHAR(VAL MOD 16);                        42   42
46 END;
47
48 PROCEDURE DISPLAY_HEX (VAL: INTEGER; N: INTEGER);      *    *    S    *    S
49 TYPE TAGS = (TAG1, TAG2);                              *    *    *
50 VAR I,J: INTEGER;                                      *    *    S
51     X: RECORD CASE TAGS OF                             *    *
52       TAG1: (INT: INTEGER);                            *    *    S
53       TAG2: (BYTES: ARRAY [1..4] OF BYTE);             *    *    10
54     END;
55 BEGIN
56   X.INT:=VAL;                                          51   52   48
57   FOR I:=5-N TO 4 DO                                   50   48
58     DISPLAY_HEX_CHARS(X.BYTES[I]);                     42   51   53   50
59 END;
```

```
60
61 "INITIALIZATION"
62
63 PROCEDURE INIT_FILE (LU: INTEGER; VAR MODE: FILE_MODES);                    *    *    S    *    9
64 VAR SVC7_PARM: SVC7_BLOCK;                                                 *   22
65 BEGIN
66   WITH SVC7_PARM DO BEGIN                                                 64
67     SVC7_LU:=LU; SVC7_MOD:=SVC7_FETCH_ATTR;                               *   63    *    *
68     SVC7(SVC7_PARM);                                                     22   64
69     IF SVC7_STAT<>0 THEN BEGIN                                            *
70       DISPLAY_TEXT('SVC7 ERROR $');                                      27
71       DISPLAY_HEX(SVC7_STAT,1);                                          48   69
72       DISPLAY_TEXT(' ON LU $');                                          27
73       DISPLAY_HEX(SVC7_LU,1);                                            48   67
74       DISPLAY(NL);                                                       32    *
75       SVC7_MOD:=0; SVC7 RECLEN:=0;                                       67    *
76     END;
77     IF SVC7 MOD = 240 THEN MODE:=ASC_MODE                                67   63    9
78     ELSE IF (SVC7_RECLEN=0) OR (SVC7 RECLEN>=512) THEN MODE:=PAGE_MODE   75   75   63    9
79     ELSE MODE:=BYTE_MODE;                                                63    9
80   END;
81 END;
82
83 "ERROR LOGGING"
84
85 PROCEDURE SVC1_ERROR_LOG (SVC1_PARM: SVC1 BLOCK);                         *    *   21
86 BEGIN
87   WITH SVC1_PARM DO BEGIN                                                 85
88     DISPLAY_TEXT('SVC1 ERROR $');                                        27
89     DISPLAY_HEX(SVC1_STAT,1);                                            48    *
90     DISPLAY_HEX(SVC1_DEV_STAT,1);                                        48    *
91     DISPLAY_TEXT(' ON LU $');                                            27
92     DISPLAY_HEX(SVC1_LU,1);                                              48    *
93     DISPLAY_TEXT(' FUNC $');                                             27
94     DISPLAY_HEX(SVC1_FUNC,1);                                            48    *
95     DISPLAY_TEXT(' XLEN $');                                             27
96     DISPLAY_HEX(SVC1_XFER_LEN,2);                                        48    *
97     DISPLAY_TEXT(' BLOCK $');                                            27
98     DISPLAY_HEX(BLOCKS,2);                                               48   19
99     DISPLAY(NL);                                                         32   74
100  END;
101 END;
102
103 "ASC INPUT PROCESSING"
104
105 PROCEDURE READ_ASC (VAR EOF: BOOLEAN);                                    *    *    S
106 VAR SVC1_PARM: SVC1 BLOCK;                                                *   21
107     I, J: SHORTINTEGER;                                                   *    *   14
108     REPLY: BYTE;                                                          *   10
109     ERRORS: INTEGER;                                                      *    S
110 BEGIN
111   ERRORS:=0;                                                            109
112   WITH SVC1_PARM DO REPEAT                                              106
113     SVC1_LU:=1; SVC1_FUNC:=SVC1_READ+SVC1 IMAGE+SVC1 WAIT;               92   94    *    *    *
114     SVC1_BUFSTART:=ADDRESS(BUF);                                          *    *   12
115     SVC1_BUFEND:=SVC1_BUFSTART+515;                                       *  114
116     SVC1(SVC1_PARM);                                                     21  106
117     IF SVC1_STAT<>0 THEN BEGIN                                           89
118       REPLY:=NAK;                                                       108    8
119       SVC1_ERROR_LOG(SVC1_PARM);                                         85  106
```

```
120       END
121       ELSE REPLY:=ACK;                                         108    8
122       J:=0;                                                    107
123       FOR I:=1 TO 512 DO                                       107
124         J:=J+(BUF.DATA[I]);                                    107  107   12   13  107
125       J:=J MOD 256;                                            107  107
126       IF ((J MOD 256)<>BUF.CKSUM) AND (REPLY=ACK) THEN BEGIN   107   12   15  108    8
127         DISPLAY_TEXT('CHECKSUM ERROR$');                        27
128         DISPLAY_TEXT(' XPTD=$'); DISPLAY_HEX(J,2);              27   48  107
129         DISPLAY_TEXT(' RCVD=$'); DISPLAY_HEX(BUF.CKSUM,2);      27   48   12   15
130         DISPLAY(NL);                                            32   74
131         REPLY:=NAK;                                            108    8
132       END;
133       SVC1_FUNC:=SVC1_WRITE+SVC1_IMAGE+SVC1_WAIT;               94    *  113  113
134       SVC1_BUFSTART:=ADDRESS(REPLY);                           114  114  108
135       SVC1_BUFEND:=SVC1_BUFSTART;                              115  114
136       SVC1(SVC1_PARM);                                          21  106
137       IF SVC1_STAT<>0 THEN SVC1_ERROR_LOG(SVC1_PARM);           89   85  106
138       IF REPLY=NAK THEN ERRORS:=ERRORS+1;                      108    8  109  109
139     UNTIL (ERRORS>16) OR (REPLY=ACK);                          109  108    8
140     IF BUF.EOF = EOT THEN EOF:=TRUE ELSE EOF:=FALSE;            12  105    8  105    *  105    *
141     IF REPLY<>ACK THEN EOF:=TRUE;  "FORCE ABORT"               108    8  105  140
142 END;
143
144 "ASC OUTPUT"
145
146 PROCEDURE WRITE_ASC (EOF: BOOLEAN);                              *    *    S
147 VAR REPLY: BYTE;                                                 *   10
148       SVC1_PARM: SVC1_BLOCK;                                     *   21
149       I: INTEGER;                                                *    S
150       ERRORS: INTEGER;                                           *    S
151       OK: BOOLEAN;                                               *    S
152 BEGIN
153     ERRORS:=0;                                                 150
154     BUF.CKSUM:=0;                                               12   15
155     FOR I:=1 TO 512 DO                                         149
156       BUF.CKSUM:=BUF.CKSUM+(BUF.DATA[I]);                       12   15   12   15   12   13  149
157     BUF.CKSUM:=BUF.CKSUM MOD 256;                               12   15   12   15
158     IF EOF THEN BUF.EOF:=EOT ELSE BUF.EOF:=0;                  146   12  146    8   12  146
159     ERRORS:=0;                                                 150
160     WITH SVC1_PARM DO REPEAT                                   148
161       SVC1_LU:=2; SVC1_FUNC:=SVC1_WRITE+SVC1_IMAGE+SVC1_WAIT;   92   94  133  113  113
162       SVC1_BUFSTART:=ADDRESS(BUF);                             114  114   12
163       SVC1_BUFEND:=SVC1_BUFSTART+515;                          115  114
164       SVC1(SVC1_PARM);                                          21  148
165       IF SVC1_STAT<>0 THEN BEGIN                                89
166         SVC1_ERROR_LOG(SVC1_PARM);                              85  148
167         OK:=FALSE;                                             151  140
168       END
169       ELSE OK:=TRUE;                                           151  140
170       SVC1_FUNC:=SVC1_READ+SVC1_IMAGE+SVC1_WAIT;                94  113  113  113
171       SVC1_BUFSTART:=ADDRESS(REPLY);                           114  114  147
172       SVC1_BUFEND:=SVC1_BUFSTART;                              115  114
173       SVC1(SVC1_PARM);                                          21  148
174       IF SVC1_STAT<>0 THEN BEGIN                                89
175         SVC1_ERROR_LOG(SVC1_PARM); OK:=FALSE;                   85  148  151  140
176       END;
177       IF OK AND (REPLY <> ACK) THEN BEGIN                      151  147    8
178         DISPLAY_TEXT('NEGATIVE ACKNOWLEDGEMENT $');             27
179         DISPLAY_HEX(REPLY,1); DISPLAY(NL);                      48  147   32   74
```

```
180       OK:=FALSE;                                          151   140
181     END;
182     IF NOT OK THEN ERRORS:=ERRORS+1;                      151   150   150
183   UNTIL (ERRORS>16) OR OK;                                150   151
184 END;
185
186 PROCEDURE READ_FILE (VAR EOF: BOOLEAN);                    *     *     S
187 VAR I: INTEGER; C: CHAR;                                   *     S     *     S
188 BEGIN
189   IF IN_MODE = ASC_MODE THEN READ_ASC(EOF)                17     9   105   186
190   ELSE IF IN_MODE = PAGE_MODE THEN READPAGE(BUF.DATA,EOF) 17     9     *    12    13   186
191   ELSE BEGIN
192     I:=1; EOF:=FALSE;                                    187   186   140
193     IF EM_FOUND THEN EOF:=TRUE                            18   186   140
194       ELSE WHILE I<=512 DO BEGIN                         187
195         READ(C);  BUF.DATA[I]:=ORD(C);                     *   187    12    13   187    38   187
196         IF C = EM THEN BEGIN                             187     *
197           I:=512; EM_FOUND:=TRUE;                        187    18   140
198         END;
199         I:=I+1;                                          187   187
200       END;
201   END;
202 END;
203
204 PROCEDURE WRITE_FILE (EOF: BOOLEAN);                       *     *     S
205 VAR I: INTEGER; C: CHAR;                                   *     S     *     S
206 BEGIN
207   IF OUT_MODE = ASC_MODE THEN WRITE_ASC(EOF)              17     9   146   204
208   ELSE IF OUT_MODE = PAGE_MODE THEN WRITEPAGE(BUF.DATA,EOF) 17   9     *    12    13   204
209   ELSE IF NOT EOF THEN BEGIN                             204
210     I:=1;                                                205
211     WHILE I<=512 DO BEGIN                                205
212       C:=CHR(BUF.DATA[I]);                               205    38    12    13   205
213       WRITE(C);                                            *   205
214       IF C=EM THEN I:=512;                               205   196   205
215       I:=I+1;                                            205   205
216     END;
217   END;
218 END;
219
220 PROCEDURE RUN;                                             *
221 VAR EOF: BOOLEAN;                                          *     S
222 BEGIN
223   EM_FOUND:=FALSE;                                        18   140
224   BLOCKS:=1;                                              19
225   INIT_FILE(1,IN_MODE);                                   63    17
226   INIT_FILE(2,OUT_MODE);                                  63    17
227   REPEAT
228     READ_FILE(EOF);                                      186   221
229     WRITE_FILE(EOF);                                     204   221
230     BLOCKS:=BLOCKS+1;                                     19    19
231   UNTIL EOF;                                             221
232 END;
233
234 BEGIN
235   RUN                                                    220
236 END.
```

```
CROSS REFERENCE   * IS DEF   = IS ASG

-A-
 ACK                 8*  121   126   139   141   177
 ADDRESS           114   134   162   171
 ASC_MODE            9    77   189   207

-B-
 BLOCKS             19*   98   224=  230=  230
 BOOLEAN            18   105   146   151   186   204   221
 BUF                12*  114   124   126   129   140   154   156   156   156   157   157   158   158   162   190   195   208
                   212
 BYTE               10    53   108   147
 BYTES              53    58
 BYTE_MODE           9    79

-C-
 C                 187*  195   195   196   205*  212=  213   214
 CHAR              187   205
 CHR                38    39   212
 CKSUM              15*  126   129   154=  156=  156   157=  157

-D-
 DATA               13*  124   156   190   195=  208   212
 DISPLAY            32    38    39    74    99   130   179
 DISPLAY_HEX        48*   71    73    89    90    92    94    96    98   128   129   179
 DISPLAY_HEX_       36*   42*   44    45    58
 DISPLAY_TEXT       27*   70    72    88    91    93    95    97   127   128   129   178

-E-
 EM                196   214
 EM FOUND           18*  193   197=  223=
 EOF                14*  105*  140   140=  140=  141=  146*  158   158=  158=  186*  189   190   192=  193=  204*  207   208
                   209   221*  228   229   231
 EOT                 8*  140   158
 ERRORS            109*  111=  138=  138   139   150*  153=  159=  182=  182   183

-F-
 FALSE             140   167   175   180   192   223
 FILEXFER            7*
 FILE_MODES          9*   17    63

-I-
 I                  28*   30=   31    32    32=   32    50*   57=   58   107*  123=  124   149*  155=  156   187*  192=  194
                   195   197=  199=  199   205*  210=  211   212   214=  215=  219
 INIT_FILE          63*  225   226
 INT                52    56=
 INTEGER            19    28    36    42    48    48    50    52    63   109   149   150   187   205
 IN_MODE            17*  189   190   225

-J-
 J                  50*  107*  122=  124=  124   125=  125   126   128

-L-
 LINE               27
 LU                 63*   67

-M-
 MODE               63*   77=   78=   79=
-N-
 N              48*   57
 NAK             8*  118   131   138
 NL             74    99   130   179

-O-
 OK            151*  167=  169=  175=  177   180=  182   183
 ORD            38    39   195
 OUT_MODE       17*  207   208   226

-P-
 PAGE           10*   13
 PAGE_MODE       9    78   190   208
 PARM           21*   22*

-R-
 READ          195
 READPAGE      190
 READ_ASC      105*  189
 READ_FILE     186*  228
 REPLY         108*  118=  121=  126   131=  134   138   139   141   147*  171   177   179
 RUN           220*  235

-S-
 SHORTINTEGER   14    15   107
 SVC1           21*  116   136   164   173
 SVC1_BLOCK     21    85   106   148
 SVC1_BUFEND   115=  135=  163=  172=
 SVC1_BUFSTAR  114=  115   134=  135   162=  163   171=  172
 SVC1_DEV_STA   90
 SVC1_ERROR_L   85*  119   137   166   175
 SVC1_FUNC      94   113=  133=  161=  170=
 SVC1_IMAGE    113   133   161   170
 SVC1_LU        92   113=  161=
 SVC1_PARM      85*   87   106*  112   116   119   136   137   148*  160   164   166   173   175
 SVC1_READ     113   170
 SVC1_STAT      89   117   137   165   174
 SVC1_WAIT     113   133   161   170
 SVC1_WRITE    133   161
 SVC1_XFER_LE   96
 SVC2PAUS       23*
 SVC7           22*   68
 SVC7_BLOCK     22    64
 SVC7_FETCH_A   67
 SVC7_LU        67=   73
 SVC7_MOD       67=   75=   77
 SVC7_PARM      64*   66    68
 SVC7_RECLEN    75=   78    78
 SVC7_STAT      69    71

-T-
 TAG1           49    52*
 TAG2           49    53*
 TAGS           49*   51*
 TEXT           27*   31    32
 TRUE          140   141   169   193   197

-V-
 VAL            36*   38    38    39    42*   44    45    48*   56
 -W-
  WRITE           213
  WRITEPAGE       208
  WRITE_ASC       146*  207
  WRITE_FILE      204*  229

 -X-
  X                51*   56    58

END XREF   86 IDENTIFIERS   413 TOTAL REFERENCES
  33 COLLISIONS.
```

```
(* Q+ *)
(* LV1:ERRORS.TEXT *)
(* LPRINTER: *)

PROGRAM X832_WD;
(***********************************************************************)
(*                                                                     *)
(* PROGRAM TO RECEIVE DATA FROM THE INTERDATA 8/32. SERIAL,            *)
(* ASYNCHRONOUS LINE TRANSMISSION.                                     *)
(*                                                                     *)
(***********************************************************************)
CONST
   BUFFSIZE  =  30724;   (* 60 DISK BLOCKS + 4 BYTES *)
   BLLEN     =    512;   (* BYTES IN DISK BLOCK *)
   BLINBUFF  =     60;   (* DISK BLOCKS IN BUFFER *)
   FLAGDIST  =    513;   (* DISTANCE FROM BEGINNING OF BLOCK TO
                            "LASTBLOCK" FLAG *)
   CRCDIST   =    515;   (* DISTANCE TO CRC BYTE *)
                         (* 1 TIME UNIT = ABOUT .22 SECOND *)
   LONGTIME  =  (* TEST 1000 *) 200;    (* LONG TIMEOUT, ABOUT 5.5 MIN *)
   MEDTIME   =    136;   (* MEDIUM TIMEOUT, ABOUT 30 SEC *)
   SHRTTIME  =     68;   (* SHORT TIMEOUT, ABOUT 15 SEC *)
   PORT_B    =   -992;   (* PORT B (REMOTE UNIT) HAS
                            ADDRESS -992 (FC20 HEX) *)
   PKTLEN    =    516;   (* DATA PACKET IS 516 BYTES:
                            512 OF DATA,
                              1 UNUSED,
                              1 "LASTBLOCK" FLAG,
                              1 UNUSED,
                              1 CRC *)
```

```
      (* SEE MICROENGINE USER'S MANUAL (PP 35-41) FOR DETAILS ON SERIAL
         PORT REGISTERS *)
      (* BITS IN SERIAL PORT STATUS REGISTER *)
      SNDEMPTY = 0;
      RECFULL  = 1;
      OVERRUN  = 2;
      PARITY   = 3;
      FRAMING  = 4;
      CARRIER  = 5;
      READY    = 6;
      CHANGE   = 7;

      (* BITS IN SERIAL PORT CONTROL REGICTER 1 *)
      RNGREADY = 0;
      REQTOSND = 1;
      RECENABL = 2;
      PARENABL = 3;
      ECHO     = 4;
      STOPBITS = 5:
      BREAK    = 6;
      LOOPNORM = 7;

      (* BITS IN SERIAL PORT CONTROL REGISTER 2 *)
      CLOCK0   = 0;
      CLOCK1   = 1;
      CLOCK2   = 2;
      RCVRRATE = 3;
      PARITSET = 4;
      CHARMODE = 5;
      CHARLEN6 = 6;
      CHARLEN7 = 7;

  TYPE
     STATBITS = SNDEMPTY. . CHANGE;
     CR1BITS  = RNGREADY. . LOOPNORM;
     CR2BITS  = CLOCK0   . . CHARLEN7;
     TWOCASES = 1. . 2;
     BYTETYPE = PACKED ARRAY [1. . 1] OF CHAR;

     SERPORT = RECORD    (* SERIAL PORT REGISTERS *)
               DATA:     CHAR;
               STATUS:   PACKED ARRAY [STATBITS] OF BOOLEAN;
               CR2:      PACKED ARRAY [CR2BITS ] OF BOOLEAN;
               CR1:      PACKED ARRAY [CR1BITS ] OF BOOLEAN
               END;

     SERDEV = RECORD CASE TWOCASES OF
              1: (ADDRESS: INTEGER);
              2: (PORTPTR: ^SERPORT)
              END;
```

```
VAR
   BUFF: PACKED ARRAY [1..BUFFSIZE] OF CHAR;  (* INPUT BUFFER *)
   LINE: SERDEV;
   EOT, ACK, NAK, ENQ: CHAR;    (* ASCII CONTROL CHARS *)
   OK,      (* RESPONSE MEANING PACKET WAS REC'D OK *)
   NOTOK,   (* RESPONSE MEANING PACKET NOT REC'D OK *)
   FLAGBYTE: CHAR;   (* CONTAINS EOT IF THIS IS LAST PACKET
                        IN THE TRANSMISSION. ANY OTHER CODE
                        MEANS NOT LAST PACKET *)
   BLOCKS,     (* # OF BLOCKS TO BE WRITTEN TO DISK FROM INPUT BUFFER *)
   BLOKSIN,    (* # OF BLOCKS REC'D OK OFF THE LINE *)
   BLOKSOUT,   (* # OF BLOCKS SUCCESSFULLY WRITTEN TO DISK *)
   STARTBYT: INTEGER; (* BYTE WITHIN BUFFER WHERE THE CURRENT
                         PACKET (AND BLOCK) STARTS *)
   DISKFILE: FILE;    (* NAME BY WHICH THIS PGM. KNOWS THE OUTPUT FILE *)
   OUTFLNAM: STRING[20];   (* NAME OF OUTPUT FILE WHICH WILL APPEAR IN
                              .THE DISK'S VOLUME DIRECTORY *)
   ERROR,    (* TRUE= SOME SORT OF ERROR WAS DETECTED *)
   FATALERR, (* TRUE= AN ERROR WAS DETECTED & IT IS FATAL *)
   LASTPKT: BOOLEAN; (* TRUE= JUST RECEIVED THE LAST PACKET IN THE
                        TRANSMISSION *)
     I: INTEGER;   (* TEST *)
```

```
PROCEDURE RESPOND (CCHAR: CHAR; VAR ABORT: BOOLEAN);
(* WHEN 8/32 IS READY TO RECEIVE, IT SENDS AN ENQ CHAR.
   PROCEDURE WAITS FOR ENQ THEN SENDS RESPONSE MESSAGE OF A
   SINGLE CHAR *)
VAR
   TIME: INTEGER;
   EXPCTENQ: CHAR;
   ABYTE: BYTETYPE;
   ERROR: BOOLEAN;
   TICK: INTEGER,
   HAVECHAR: BOOLEAN;
BEGIN
ABORT := FALSE;
TIME := MEDTIME;
ERROR := FALSE;

REPEAT
   TICK := 1000;
   REPEAT   (* POLL LINE UNTIL CHAR ARRIVES OR TIME TO DECR. TIME PARAM *)
      HAVECHAR := LINE.PORTPTR^.STATUS [RECFULL];
      (* TEST  IF TICK <= 1000 THEN HAVECHAR := TRUE; *)
      TICK := TICK - 1
   UNTIL HAVECHAR OR (TICK <= 0);
   IF NOT HAVECHAR
   THEN TIME := TIME -1
UNTIL HAVECHAR OR (TIME <= 0);

(* GET WHATEVER IS IN RECEIVER HOLDING REGISTER *)
ABYTE[1] := LINE.PORTPTR^.DATA;
(* TEST  ABYTE[1] := ENQ; *)
WITH LINE.PORTPTR^ DO
   BEGIN
   ERROR := STATUS [OVERRUN] OR (*CHECK FOR ERRORS*)
            STATUS [PARITY ] OR
            STATUS [FRAMING];
   IF STATUS[OVERRUN] THEN WRITELN ('OVERRUN');
   IF STATUS[PARITY]  THEN WRITELN('PARITY');
   IF STATUS[FRAMING] THEN WRITELN('FRAMING')
   END;
```

```
PROCEDURE READPKT (STARTBYT: INTEGER; VAR PKTERROR, ABORT: BOOLEAN);
VAR
    BYT,
    LASTBYT,
    TIME: INTEGER;
    ERROR: BOOLEAN;
    ABYTE: BYTETYPE;
    TICK: INTEGER;
    HAVECHAR: BOOLEAN;
    I: INTEGER;     (* TEST *)
    R: INTEGER;     (* TEST *)
BEGIN
PKTERROR := FALSE;
ABORT := FALSE;
BYT := STARTBYT;
LASTBYT := STARTBYT + PKTLEN;
LINE.PORTPTR^.DATA := ENQ;
TIME := LONGTIME;
ERROR := FALSE;

REPEAT
   TICK := 1000;
   REPEAT   (* POLL LINE UNTIL CHAR ARRIVES OR TIME TO DECR. TIME PARAM *)
      HAVECHAR := LINE.PORTPTR^.STATUS [RECFULL];
      (* TEST  IF TICK <= 1000 THEN HAVECHAR := TRUE;  *)
      TICK := TICK - 1
   UNTIL HAVECHAR OR (TICK <= 0);
   IF NOT HAVECHAR
   THEN TIME := TIME -1
UNTIL HAVECHAR OF (TIME <= 0);

(* GET WHATEVER IS IN RECEIVER HOLDING REGISTER *)
ABYTE[1] := LINE.PORTPTR^.DATA;
(* TEST  ABYTE[1] := 'A';   *)
WITH LINE.PORTPTR^ DO
   BEGIN
   ERROR := STATUS [OVERRUN] OR (*CHECK FOR ERRORS*)
            STATUS [PARITY  ] OR
            STATUS [FRAMING];
   IF STATUS[OVERRUN] THEN WRITELN ('OVERRUN');
   IF STATUS[PARITY]  THEN WRITELN('PARITY');
   IF STATUS[FRAMING] THEN WRITELN('FRAMING')
   ,  IF ERROR THEN WRITELN ('ERROR 1')
   END;
BUFF[BYT] := ABYTE[1];
```

```
EXPCTENQ := ABYTE[1];
IF TIME <= 0
THEN BEGIN    (* TIMED OUT *)
   WRITELN;
   WRITELN ('PROCEDURE RESPOND:');
   WRITELN ('TIMED OUT WAITING FOR ENQ');
   ABORT := TRUE
   END;
IF EXPCTENQ <> ENQ
THEN BEGIN (* EXPECTING AN ENQ BUT ANY CHAR WILL DO FOR SYNC PURPOSES *)
   WRITELN;
   WRITELN ('PROCEDURE RESPOND:');
   WRITELN ('RECEIVED ', EXPCTENQ);
   WRITELN ('EXPECTED ENQ.  CONTINUING...')
   END;
LINE.PORTPTR^.DATA := CCHAR (* PUT THE RESPONSE CHAR ON LINE *)
;  IF CCHAR=ACK THEN WRITELN ('SENDING ACK')
ELSE IF CCHAR=NAK THEN WRITELN ('SENDING NAK')
ELSE WRITELN ('SENDING NEITHER ACK NOR NAK');
END;


FUNCTION CRC_OK (STARTBYT: INTEGER): BOOLEAN;
(* RECALCULATE THE CHECKSUM & SEE IF IT MATCHES CHECKSUM IN PACKET *)
(* ONLY BYTES IN THE DATA PORTION OF THE PACKET ARE USED TO        *)
(* CALCULATE THE CHECKSUM.  FORMULA IS SUM OF BYTES MOD 256.       *)
   VAR
   BYTE,
   CALC_CRC: INTEGER;   (* THE CALCULATED CHECKSUM *)
   CHARA: CHAR;
BEGIN
CALC_CRC := 0;
FOR BYTE := STARTBYT TO (STARTBYT+BLLEN-1) DO (*RECALCULATION*)
   BEGIN
   CHARA := BUFF [BYTE];
   CALC_CRC := (CALC_CRC + ORD(CHARA)) MOD 256
   END;
CHARA := BUFF [STARTBYT + CRCDIST];   (* GET CRC FROM PACKET *)
CRC_OK := CALC_CRC = ORD(CHARA)       (* COMPARE & RETURN RESULT *)
(* TEST *)  ; CRC_OK := TRUE
END;
```

```
 PROCEDURE READPKT (STARTBYT: INTEGER; VAR PKTERROR, ABORT: BOOLEAN);
 VAR
     BYT,
     LASTBYT,
     TIME: INTEGER;
     ERROR: BOOLEAN;
     ABYTE: BYTETYPE;
     TICK: INTEGER;
     HAVECHAR: BOOLEAN;
     I: INTEGER;    (* TEST *)
     R: INTEGER;    (* TEST *)
 BEGIN
 PKTERROR := FALSE;
 ABORT := FALSE;
 BYT := STARTBYT;
 LASTBYT := STARTBYT + PKTLEN;
 LINE.PORTPTR^.DATA := ENQ;
 TIME := LONGTIME;
 ERROR := FALSE;

 REPEAT
    TICK := 1000;
    REPEAT   (* POLL LINE UNTIL CHAR ARRIVES OR TIME TO DECR. TIME PARAM *)
       HAVECHAR := LINE.PORTPTR^.STATUS [RECFULL];
       (* TEST  IF TICK <= 1000 THEN HAVECHAR := TRUE; *)
       TICK := TICK - 1
    UNTIL HAVECHAR OR (TICK <= 0);
    IF NOT HAVECHAR
    THEN TIME := TIME -1
UNTIL HAVECHAR OR (TIME <= 0);

(* GET WHATEVER IS IN RECEIVER HOLDING REGISTER *)
ABYTE[1] := LINE.PORTPTR^.DATA;
(* TEST  ABYTE[1] := 'A';   *)
WITH LINE.PORTPTR^ DO
   BEGIN
   ERROR := STATUS [OVERRUN] OR (*CHECK FOR ERRORS*)
            STATUS [PARITY ] OR
            STATUS [FRAMING];
   IF STATUS[OVERRUN] THEN WRITELN ('OVERRUN');
   IF STATUS[PARITY]  THEN WRITELN('PARITY');
   IF STATUS[FRAMING] THEN WRITELN('FRAMING')
   ;  IF ERROR THEN WRITELN ('ERROR 1')
   END;
BUFF[BYT] := ABYTE[1];
```

```
IF ERROR
THEN PKTERROR := TRUE;
IF TIME <= 0
THEN BEGIN
   WRITELN;
   WRITELN ('PROC READPKT:');
   WRITELN ('TIMEOUT BETWEEN PACKETS');
   ABORT := TRUE
   END;
BYT := SUCC(BYT);
R := 0;
REPEAT
   TIME := SHRTTIME;
   ERROR := FALSE;

   REPEAT
      TICK := 1000;
      REPEAT   (* POLL LINE UNTIL CHAR ARRIVES OR TIME TO DECR.  TIME PARAM *)
         HAVECHAR := LINE.PORTPTR^.STATUS [RECFULL];
         (* TEST  IF TICK <= 1000 THEN HAVECHAR := TRUE;  *)
         TICK := TICK - 1
      UNTIL HAVECHAR OR (TICK <= 0);
      IF NOT HAVECHAR
      THEN TIME := TIME -1
   UNTIL HAVECHAR OR (TIME <= 0);

   (* GET WHATEVER IS IN RECEIVER HOLDING REGISTER *)
   ABYTE[1] := LINE.PORTPTR^.DATA;
   (* TEST  ABYTE[1] := 'B';  *)
   WITH LINE.PORTPTR^ DO
      BEGIN
      ERROR := STATUS [OVERRUN] OR (*CHECK FOR ERRORS*)
               STATUS [PARITY ] OR
               STATUS [FRAMING];
      IF STATUS[OVERRUN] THEN WRITELN ('OVERRUN');
      IF STATUS[PARITY]  THEN WRITELN('PARITY');
      IF STATUS[FRAMING] THEN WRITELN('FRAMING')
      ;  IF ERROR THEN WRITELN ('ERROR 2')
      END;

   BUFF[BYT] := ABYTE[1];
   IF ERROR
   THEN PKTERROR := TRUE;
   BYT := SUCC(BYT)
   ;  R := SUCC(R)
UNTIL (BYT >= LASTBYT) OR (TIME <= 0);

IF TIME <= 0
THEN PKTERROR := TRUE  (* TIMED OUT *)
; (* TEST  FOR I := STARTBYT TO LASTBYT DO WRITE(BUFF[I]);   *)
WRITELN(R)
END;
```

```
 PROCEDURE INITLINE;
 (* INITIALIZE SERIAL PORT REGISTERS FOR PORT ASSOCIATED WITH LINE *)
 BEGIN
 LINE.ADDRESS := PORT_B;
 WITH LINE.PORTPTR^ DO
    BEGIN
    CR1 [RNGREADY] := TRUE;     (* ENABLE CARRIER & DATA SET READY BITS *)
    CR1 [REQTOSND] := TRUE;     (* ENABLE TRANSMISSION *)
    CR1 [RECENABL] := TRUE;     (* ENABLE RECEPTION *)
    CR1 [PARENABL] := FALSE;    (* NO HDWRE PARITY GENERATION/CHECKING *)
    CR1 [ECHO    ] := FALSE;    (* DON'T ECHO RECEIVED DATA *)
    CR1 [STOPBITS] := TRUE;     (* 1 STOP BIT *)
    CR1 [BREAK   ] := FALSE;
    CR1 [LOOPNORM] := TRUE;     (* NORMAL (NOT TEST) OPERATION *)

    CR2 [CLOCK0  ] := TRUE;     (* CLOCK RATE 1 (32X) *)
    CR2 [CLOCK1  ] := FALSE;
    CR2 [CLOCK2  ] := FALSE;
    CR2 [RCVRRATE] := FALSE;    (* RECEIVER CLOCK RATE = RATE 1 *)
    CR2 [PARITSET] := FALSE;    (* EVEN PARITY, IF APPLICABLE *)
    CR2 [CHARMODE] := FALSE;    (* ASYNCHRONOUS LINE *)
    CR2 [CHARLEN6] := FALSE;    (* DATA CHARACTERS ARE 8 BITS *)
    CR2 [CHARLEN7] := FALSE;
    END
END;

PROCEDURE INIT;
BEGIN
STARTBYT := 1;
BLOCKS := 0;
BLOKSIN := 0;
BLOKSOUT :=0;

ACK := CHR(6);    OK := ACK;
NAK := CHR(21);   NOTOK := NAK;
ENQ := CHR(5);
EOT := CHR(7);

WRITELN;
WRITELN ('NAME OF OUTPUT FILE?');
READLN (OUTFLNAM);
REWRITE (DISKFILE, OUTFLNAM);
IF IORESULT <> 0
THEN BEGIN
   WRITELN;
   WRITELN ('PROCEDURE INIT:');
   WRITELN ('I/O ERROR # ', IORESULT);
   HALT
   END;
INITLINE
END;
```

```
PROCEDURE FINISH (ABORTING: BOOLEAN);
BEGIN
CLOSE (DISKFILE, LOCK);
WRITELN;
WRITELN ('NUMBER OF BLOCKS WRITTEN = ', BLOKSOUT);
IF ABORTING
THEN WRITELN ('DATA TRANSFER ABORTED')
ELSE WRITELN ('DATA TRANSFER FINISHED')
END;


BEGIN
FOR I := 1 TO BUFFSIZE DO BUFF[I] := '/';
INIT;
REPEAT
   READPKT (STARTBYT, ERROR, FATALERR);
   IF CRC_OK (STARTBYT) AND NOT (ERROR OR FATALERR)
   THEN BEGIN
      IF NOT FATALERR
      THEN BEGIN
         BLOCKS := SUCC(BLOCKS);
         FLAGBYTE := BUFF [STARTBYT+FLAGDIST];
         LASTPKT := FLAGBYTE = EOT;
         (* TEST  IF BLOCKS >= 5 THEN LASTPKT := TRUE;  *)
         IF (BLOCKS >= BLINBUFF) OR LASTPKT
         THEN BEGIN  (* BUFFER FULL OR RECEIVED LAST PACKET *)
            BLOKSIN := BLOKSIN + BLOCKS;
            BLOKSOUT := BLOKSOUT + BLOCKWRITE
                        (DISKFILE, BUFF, BLOCKS, BLOKSOUT);
            (* IF IORESULT <> 0
            THEN WRITELN ('DISK WRITE ERROR. # ', IORESULT);  *)
            BLOCKS := 0;
            STARTBYT := 1
            END
         ELSE STARTBYT := STARTBYT + BLLEN
         END;
      RESPOND (OK, FATALERR)
      END
   ELSE IF NOT FATALERR
        THEN RESPOND (NOTOK, FATALERR)
UNTIL LASTPKT OR FATALERR;
FINISH (FATALERR)
END
```

APPENDIX B

SOURCE CODE FOR MEPASCAL COMPILER PASS SIX: MEPASS6

```
     1 "################################################"
     2 "# MODIFIED TO PRODUCE CODE FOR MICROENGINE. #"
     3 "################################################"
     4
     5 "CODE TO BE FINALIZED LATER IS MARKED $$$"
     6
     7 "%SRC_32 := FALSE"
     8 "%SRC_16 := NOT SRC_32"
     9
    10 "PER BRINCH HANSEN
C   11  INFORMATION SCIENCE
C   12  CALIFORNIA INSTITUTE OF TECHNOLOGY
C   13  PASADENA, CALIFORNIA 91125
C   14  PDP 11/45 CONCURRENT/SEQUENTIAL PASCAL
C   15  COMPILER PASS 6: CODE SELECTION
C   16  FOR PASCAL MICROENGINE."
    17
    18 TYPE FULLWORD = INTEGER;                                                     *     S
    19 TYPE INTEGER = SHORTINTEGER;                                                 S     *
    20
    21 "###########
C   22 #  PREFIX  #
C   23 ###########"
    24
    25 CONST          EOL = '(:10:)';     FF = '(:12:)';      EOM = '(:25:)';       *     *     *
    26 MAXDIGIT = 6;                                                                *
    27 PRINTLIMIT = 18;                                                             *
    28 WORDLENGTH = 2 "BYTES";                                                      *
    29 REALLENGTH = 4 "BYTES";                                                      *
    30 SETLENGTH = 16 "BYTES";                                                      *
    31 "ON STACK, WORD LENGTH (8) OF SET IS ALWAYS PUSHED
C   32 AFTER THE SET ITSELF, TO CONFORM TO MICROENGINE CONVENTION.
C   33 MICROE SUPPORTS VARIABLE-LENGTH SETS, BUT THEY ARE
C   34 NOT USED BY THIS VERSION OF CONCURRENT PASCAL."
    35
    36 SOURCE_WORD_LENGTH = "%IF SRC_16" 2; "%END"                                  *
    37                      "%IF SRC_32" 4; "%END"
    38 LISTOPTION = 0;  SUMMARYOPTION = 1; TESTOPTION = 2;  CHECKOPTION = 3;        *     *     *     *
    39 CODEOPTION = 4;  NUMBEROPTION = 5;  VARNTCHECKOPTION = 8;  "VARNT*"          *     *     *
    40 MAXWORD = 100;                                                               *
    41
    42 TYPE FILE = 1..4;                                                            *
    43
    44 CONST IDLENGTH = 12;                                                         *
    45 TYPE IDENTIFIER = ARRAY (.1..IDLENGTH.) OF CHAR;                             *     44    S
    46
    47 TYPE   POINTER = @ INTEGER;                                                  *     S
    48 TABLEPTR = @TABLE;                                                           *     *
    49 TABLE = RECORD                                                               *
    50           NEXTPORTION: TABLEPTR;                                             *     48
    51           CONTENTS: ARRAY (.1..MAXWORD.) OF INTEGER                          *     40    S
    52         END;
    53 TABLEPART = RECORD                                                           *
    54               SEGDISTANCE, STACKLENGTH: INTEGER;                             *     *     S
    55               JUMPTABLE, CONSTTABLE, XJPTABLE,                               *     *     *
    56               EXITICTABLE, DATASIZETABLE:   TABLEPTR                         *     *     48
    57             END;
    58 TABLESPTR = @TABLEPART;                                                      *     53
    59 OPTION = LISTOPTION..VARNTCHECKOPTION;            "VARNT*"                   *     38    39
```

```
   60
   61 CONST MAXINTFAC = 14;                                           *
   62 TYPE
   63 IFPTR = @IFINFO;                                                *     *
   64 IFINFO = RECORD                                                 *
   65    INTERFACES: INTEGER;                                         *     S
   66    INTERFACESIZES: ARRAY[1..MAXINTFAC] OF INTEGER               *     61    S
   67    END;
   68
   69 PASSPTR = @PASSLINK;                                            *     *
   70 PASSLINK = RECORD                                               *
   71     OPTIONS: SET OF OPTION;                                     *     59
   72     LABELS, BLOCKS, CONSTANTS, XJP_OFFSETS: INTEGER;            *     *     *     *     S
   73     RESETPOINT: FULLWORD;                                       *     18
   74     TABLES: TABLESPTR;                                          *     58
   75     INTERFACE: IFPTR                                            *     63
   76     END;
   77
   78 TYPE ARGTAG =                                                   *
   79   (NILTYPE, BOOLTYPE, INTTYPE, IDTYPE, PTRTYPE);                *     *     *     *     *
   80
   81 TYPE ARGTYPE = RECORD                                           *
   82                  CASE TAG: ARGTAG OF                            *     78
   83                    NILTYPE, BOOLTYPE: (BOOL: BOOLEAN);         79    79    *     S
   84                    INTTYPE: (INT: INTEGER);                     *     *     S
   85                    IDTYPE: (ID: IDENTIFIER);                    *     *     45
   86                    PTRTYPE: (PTR: PASSPTR)                      *     *     69
   87                END;                                            *
   88
   89 CONST MAXARG = 10;                                              *
   90 TEXT_LENGTH = 18;                                               *
   91 TYPE ARGLIST = ARRAY (.1..MAXARG.) OF ARGTYPE;                  *     89    81
   92 TEXT_TYPE = ARRAY (.1..TEXT_LENGTH.) OF CHAR;                   *     90    S
   93
   94 CONST PAGELENGTH = "$IF SRC_16" 256; "$END"                     *
   95                    "$IF SRC_32" 128; "$END"
   96 TYPE PAGE = ARRAY (.1..PAGELENGTH.) OF INTEGER;                 *     94    S
   97
   98 PROCEDURE READ(VAR C: CHAR);                                    *     *     S
   99 PROCEDURE WRITE(C: CHAR);                                       *     *     S
  100 PROCEDURE NOTUSED1;                                             *
  101 PROCEDURE NOTUSED2;                                             *
  102 PROCEDURE GET(F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);       *     *     42    *     S     *     96
  103 PROCEDURE PUT(F: FILE; P: INTEGER; BLOCK: UNIV PAGE);           *     *     42    *     S     *     96
  104 FUNCTION  FILE_LENGTH(F:FILE): INTEGER;                         *     *     42    S
  105 PROCEDURE MARK(VAR TOP: FULLWORD);                              *     *     18
  106 PROCEDURE RELEASE(TOP: FULLWORD);                               *     *     18
  107
  108 PROGRAM MAIN(VAR PARAM: ARGLIST);                               *     *     91
  109
  110 "#############################################
C 111 #  PASS(VAR OK: BOOLEAN; VAR LINK: POINTER)  #
C 112 #############################################"
  113
  114 CONST
  115
  116 "INPUT OPERATORS"
  117 "(ASSIGNTAG1 IS NEVER PRODUCED BY PASS 5.)"
  118 "(INITVAR1   IS NEVER PRODUCED BY PASS 5.)"
  119
```

```
PUSHCONST1 = 0;  PUSHVAR1 = 1;     PUSHIND1 = 2;      PUSHADDR1 = 3;        *    *    *    *
FIELD1 = 4;      INDEX1 = 5;       POINTER1 = 6;      VARIANT1 = 7;         *    *    *    *
RANGE1 = 8;      ASSIGN1 = 9;      ASSIGNTAG1 = 10;   COPY1 = 11;           *    *    *    *
NEW1 = 12;       NOT1 = 13;        AND1 = 14;         OR1 = 15;             *    *    *    *
NEG1 = 16;       ADD1 = 17;        SUB1 = 18;         MUL1 = 19;            *    *    *    *
DIV1 = 20;       MOD1 = 21;        "NOT USED"         "NOT USED"            *    *
FUNCTION1 = 24;  BUILDSET1 = 25;   COMPARE1 = 26;     COMPSTRUC1 = 27;      *    *    *    *
FUNCVALUE1 = 28; DEFLABEL1 = 29;   JUMP1 = 30;        FALSEJUMP1 = 31;      *    *    *    *
CASEJUMP1 = 32;  INITVAR1 = 33;    CALL1 = 34;        ENTER1 = 35;          *    *    *    *
RETURN1 = 36;    POP1 = 37;        NEWLINE1 = 38;     ERROR1 = 39;          *    *    *    *
CONSTANT1 = 40;  MESSAGE1 = 41;    INCREMENT1 = 42;   DECREMENT1 = 43;      *    *    *    *
PROCEDURE1 = 44; INIT1 = 45;       PUSHLABEL1 = 46;   CALLPROG1 = 47;       *    *    *    *
EOM1=48;         DUPTOS1 = 49;                                              *    *

"VIRTUAL DATA TYPES"
BYTETYPE = 0;     WORDTYPE = 1;      REALTYPE = 2;      SETTYPE = 3;        *    *    *    *

"VIRTUAL ADDRESSING MODES"

MODE0 = 0 "CONSTANT";                                                       *
MODE1 = 1 "PROCEDURE";                                                      *
MODE2 = 2 "PROGRAM";                                                        *
MODE3 = 3 "PROCESS ENTRY";                                                  *
MODE4 = 4 "CLASS ENTRY";                                                    *
MODE5 = 5 "MONITOR ENTRY";                                                  *
MODE6 = 6 "PROCESS";                                                        *
MODE7 = 7 "CLASS";                                                          *
MODE8 = 8 "MONITOR";                                                        *
MODE9 = 9 "STANDARD";                                                       *
MODE10=10 "UNDEFINED";                                                      *
MODE11=12 "MANAGER";                                      "MGR*"            *
MODE12=13 "MGR ENTRY";                                     "MGR*"           *

"COMPARISON OPERATORS"

LESS = 0;        EQUAL = 1;        GREATER = 2;      NOTLESS = 3;           *    *    *    *
NOTEQUAL = 4;    NOTGREATER = 5;   INSET = 6;                               *    *    *

"STANDARD FUNCTIONS"

TRUNC1 = 0;      ABS1 = 1;         SUCC1 = 2;         PRED1 = 3;            *    *    *    *
CONV1 = 4;       EMPTY1 = 5;       ATTRIBUTE1 = 6;    REALTIME1 = 7;        *    *    *    *
MIN_FUNC = 0;    MAX_FUNC = 7;                                              *    *

"STANDARD PROCEDURES"

DELAY1 = 0;      CONTINUE1 = 1;    IO1 = 2;           START1 = 3;           *    *    *    *
STOP1 = 4;       SETHEAP1 = 5;     WAIT1 = 6;                               *    *    *
MIN_PROC = 0;    MAX_PROC = 6;                                              *    *

"OUTPUT OPERATORS (MICROENGINE OPERATION CODES)"

SLDC00_2 = 000;  SLDC01_2 = 001;  SLDC02_2 = 002;  SLDC03_2 = 003;          *    *    *    *
SLDC04_2 = 004;  SLDC05_2 = 005;  SLDC06_2 = 006;  SLDC07_2 = 007;          *    *    *    *
SLDC08_2 = 008;  SLDC09_2 = 009;  SLDC10_2 = 010;  SLDC11_2 = 011;          *    *    *    *
SLDC12_2 = 012;  SLDC13_2 = 013;  SLDC14_2 = 014;  SLDC15_2 = 015;          *    *    *    *
SLDC16_2 = 016;  SLDC17_2 = 017;  SLDC18_2 = 018;  SLDC19_2 = 019;          *    *    *    *
SLDC20_2 = 020;  SLDC21_2 = 021;  SLDC22_2 = 022;  SLDC23_2 = 023;          *    *    *    *
SLDC24_2 = 024;  SLDC25_2 = 025;  SLDC26_2 = 026;  SLDC27_2 = 027;          *    *    *    *
SLDC28_2 = 028;  SLDC29_2 = 029;  SLDC30_2 = 030;  SLDC31_2 = 031;          *    *    *    *
```

```
SLDL01_2 = 032;  SLDL02_2 = 033;  SLDL03_2 = 034;  SLDL04_2 = 035;
SLDL05_2 = 036;  SLDL06_2 = 037;  SLDL07_2 = 038;  SLDL08_2 = 039;
SLDL09_2 = 040;  SLDL10_2 = 041;  SLDL11_2 = 042;  SLDL12_2 = 043;
SLDL13_2 = 044;  SLDL14_2 = 045;  SLDL15_2 = 046;  SLDL16_2 = 047;
SLDO01_2 = 048;  SLDO02_2 = 049;  SLDO03_2 = 050;  SLDO04_2 = 051;
SLDO05_2 = 052;  SLDO06_2 = 053;  SLDO07_2 = 054;  SLDO08_2 = 055;
SLDO09_2 = 056;  SLDO10_2 = 057;  SLDO11_2 = 058;  SLDO12_2 = 059;
SLDO13_2 = 060;  SLDO14_2 = 061;  SLDO15_2 = 062;  SLDO16_2 = 063;
          "NO MICROENGINE OPERATION CODES FROM 64 TO 119."
SINDO_2  = 120;  SIND1_2  = 121;  SIND2_2  = 122;  SIND3_2  = 123;
SIND4_2  = 124;  SIND5_2  = 125;  SIND6_2  = 126;  SIND7_2  = 127;
LDCB_2   = 128;  LDCI_2   = 129;  LCA_2    = 130;  LDC_2    = 131;
LLA_2    = 132;  LDO_2    = 133;  LAO_2    = 134;  LDL_2    = 135;
LDA_2    = 136;  LOD_2    = 137;  UJP_2    = 138;  UJPL_2   = 139;
MPI_2    = 140;  DVI_2    = 141;  STM_2    = 142;  MODI_2   = 143;
CPL_2    = 144;  CPG_2    = 145;  CPI_2    = 146;  CXL_2    = 147;
CXG_2    = 148;  CXI_2    = 149;  RPU_2    = 150;  CPF_2    = 151;
LDCN_2   = 152;  LSL_2    = 153;  LDE_2    = 154;  LAE_2    = 155;
NOP_2    = 156;  LPR_2    = 157;  BPT_2    = 158;  BBP_2    = 159;
LOR_2    = 160;  LAND_2   = 161;  ADI_2    = 162;  SBI_2    = 163;
STL_2    = 164;  SRO_2    = 165;  STR_2    = 166;  LDB_2    = 167;
          "NO MICROENGINE OPERATION CODES FROM 168 TO 175."
EQUI_2   = 176;  NEQI_2   = 177;  LEQI_2   = 178;  GEQI_2   = 179;
LEUSW_2  = 180;  GEUSW_2  = 181;  EQUPWR_2 = 182;  LEQPWR_2 = 183;
GEQPWR_2 = 184;  EQUBYT_2 = 185;  LEQBYT_2 = 186;  GEQBYT_2 = 187;
SRS_2    = 188;  SWAP_2   = 189;  TNC_2    = 190;  RND_2    = 191;
ADR_2    = 192;  SBR_2    = 193;  MPR_2    = 194;  DVR_2    = 195;
STO_2    = 196;  MOV_2    = 197;  DUP2_2   - 198;  ADJ_2    = 199;
STB_2    = 200;  LDP_2    = 201;  STP_2    = 202;  CHK_2    = 203;
FLT_2    = 204;  EQUREAL_2= 205;  LEQREAL_2= 206;  GEQREAL_2= 207;
LDM_2    = 208;  SPR_2    = 209;  EFJ_2    = 210;  NFJ_2    = 211;
FJP_2    = 212;  FJPL_2   = 213;  XJP_2    = 214;  IXA_2    = 215;
IXP_2    = 216;  STE_2    = 217;  INN_2    = 218;  UNI_2    = 219;
INT_2    = 220;  DIF_2    = 221;  SIGNAL_2 = 222;  WAIT_2   = 223;
ABI_2    = 224;  NGI_2    = 225;  DUP1_2   = 226;  ABR_2    = 227;
NGR_2    = 228;  LNOT_2   = 229;  IND_2    = 230;  INC_2    = 231;
          "NO MICROENGINE OPERATION CODES > 231."
MESSAGE_2= 232;  EOM_2    = 233;  NEWLIN_2 = 234;
"OPERATORS 232, 233, 234 ARE VIRTUAL OPS TO BE REMOVED IN PASS7"

"OTHER CONSTANTS"

"MACHINE REGISTERS (NOTE: POSITIVE REGS ARE FIELDS IN ACTIVE TIB)"
READYQ_REG =     -3;  "POINTS TO READY QUEUE"
SEGDICT_REG =    -2;  "POINTS TO SEGMENT DICTIONARY"
TASK_REG =       -1;  "POINTS TO RUNNING TASK'S TIB"
QLINK_REG =       0;  "LINK FIELD FOR SEMAPHORES"
"NOTE: REG 1 HAS 2 FIELDS"
PRIORITY_REG =    1;  "TASK'S CPU PRIORITY"
FLAG_REG =        1;  "STATE FLAGS"
LSPLIM =          2;  "LOWER STACK LIMIT (HEAPTOP)"
USPLIM =          3;  "UPPER STACK LIMIT"
STACKPTR_REG =    4;  "STACK POINTER"
LBASE_REG =       5;  "LOCAL BASE"
GBASE_REG =       6;  "GLOBAL BASE"
PC_REG =          7;  "PROGRAM COUNTER"
SEG_REG =         8;  "CURRENT SEGMENT ADDRESS"

FIXLCL_LINE = 1;  "LOCAL WORD 1 IS ALWAYS LINE # "
FIXLCL_OLDG = 2;  "LOCAL WORD 2 IS ALWAYS OLD GLOBAL BASE"
```

```
   240 FIXGBL_OLDG = 2;  "GLOBAL WORD 2 IS OLD GLOBAL BASE"                     *
   241 MSCWSIZE = 8;      "MARK STACK TAKES 8 BYTES"                            *
   242 PDP11 = TRUE;                                                            *     *
   243 CONCURRENT = TRUE;                                                       *     242
   244 SPLITLENGTH = 4 "WORDS PER REAL";                                        *
   245 HALFWORD = 1;                                                            *
   246 ONEWORD = 2;                                                             *
   247 TWOWORDS = 4;    THREEWORDS = 6;   FOURWORDS = 8;      FIVEWORDS = 10;   *     *     *     *
   248 STACK_LIMIT = 32667 "GREATEST INTEGER - 100";     CODE_LIMIT = 32667;    *     *
   249 THIS_PASS = 6;                                                           *
   250 INFILE = 2;         OUTFILE = 1;        INTERFACEFILE = 4;               *     *     *
   251 "(FILE NUMBER 3 IS USED BY PASS7 FOR FINAL CODE OUTPUT)"
   252 SIBS_OFFSET = 11;  "SIBS FIELD IS 11TH IN TIB"                           *
   253 SEQL_SEG = 129;    "AT RUN TIME, SEGMENT NO. OF SEQ'L PGM IS 129"        *
   254 INTFSEG = 128;     "AT RUN TIME, SEGMENT NO. OF INTERFACE SEGMENT"       *
   255 STARTPROC = 1;     "START EXECUTION OF SEGMENT WITH PROC NO. 1"          *
   256 STACK_ERROR = 1;   CODE_ERROR = 2;                                       *     *
   257 CALLERRORDISPL = 0;   "$$$"   "BYTES OF OBJECT CODE REQUIRED TO          *
C  258                                  CALL A RUN-TIME ERROR ROUTINE."
   259
   260 "$IF SRC_32"
   261 TYPE SPLIT_INTEGER = ARRAY [1..2] OF SHORTINTEGER;                       *     19
   262 "$END SRC_32"
   263
   264 "TYPES OF OUTPUT CODE PRODUCED BY THIS PASS:
C  265 OPERATORS, UNSIGNED BYTE OPERANDS, SIGNED BYTE OPERANDS,
C  266 'DON'T CARE' BYTE OPERANDS, 'BIG' OPERANDS, WORD OPERANDS,
C  267 OR SOME VIRTUAL CODE (NOT MICROENGINE)."
   268
   269 TYPE TYPEOFCODE = (OPTR, UB, SB, DB, B, W, NOTME);                       *     *     *     *     *     *     *     *
   270
   271 VAR
   272
   273 LINK: PASSPTR;                                                           *     69
   274 TABLERESET: FULLWORD;                                                    *     18
   275
   276 SUMMARY,       TEST,         CHECK,                                      *     *     *
   277 GENERATE,      NUMBER,       AFTERBEGIN,                                 *     *     *
   278 AFTERERROR,    DONE,         VARNTCHECK:      BOOLEAN; "VARNT*"          *     *     *     S
   279
   280 GENNINGINTFAC: BOOLEAN;  "TRUE IF GENERATING AN INTERFACE SEG"           *     S
   281
   282 JUMPTABLE, STACKTABLE, CONSTTABLE, XJPTABLE,                             *     *     *     *
   283     EXITICTABLE, DATASIZETABLE: TABLEPTR;                                *     *     48
   284
   285 "NUMBER OF ROUTINES IN EACH INTERFACE"
   286 IFSEGSIZE: ARRAY[1..MAXINTFAC] OF INTEGER;                               *     61    S
   287
   288 CONSTANTS,      "COUNTS NO. OF WORDS OF CONSTANTS IN CONST TABLE"        *
   289 XJPOFFSETPTR,   "COUNTS NO. OF WORDS OF XJP OFFSETS IN CONST TABLE"      *
   290 POOLSIZE,       "TOTAL BYTES IN CONST POOL, INCLUDING XJP OFFSETS"       *
   291 STACKLENGTH,    "AS IT COMES FROM PASS5, STACKLENGTH IS THE AMOUNT       *
C  292                  OF EXTRA STACK SPACE REQUIRED BY A ROUTINE.
C  293                  IT IS INCREASED BY THE AMOUNT OF STACK SPACE
C  294                  THE ROUTINE NEEDS FOR VARIABLES & CALCULATIONS (IN-
C  295                  CLUDING STACKSPACE NEEDED BY ROUTINES CALLED BY
C  296                  THE CURRENT ROUTINE)."
   297 VARLENGTH,      "BYTES OF LOCAL VARS NEEDED FOR CURRENT ROUTINE"         *
   298 PARAMLENGTH,    "BYTES OF PARAMS COMING INTO CURRENT ROUTINE"            *
   299 POPLENGTH,      "BYTES TO POP AFTER CURRENT ROUTINE"                     *
```

```
  300 TEMP,                                                                             *
  301 MAXTEMP,         "MAX BYTES OF CALCULATION STACK FOR CURRNT RTN"                  *
  302 BLOCK,           "ROUTINE (PROCEDURE) LABEL FOR CURRENT ROUTINE"                  *
  303 LOCATION,        "BYTE LOCATION COUNTER, RELATIVE TO START OF SEGMENT,            *
C 304                   & POINTS TO NEXT BYTE TO BE GENERATED."
  305 IFSEGNUM,        "SEQUENTIAL NUMBER OF THE LAST INTERFACE SEG GEN'D"              *
  306 LINE,            "SOURCE LINE NUMBER"                                             *
  307 OP,              "VIRTUAL MACHINE INSTRUCTION OPERATOR, FROM PASS5"               *
  308 ARG1, ARG2, ARG3, ARG4, ARG5   "VIRTUAL MACHINE INSTRUC. OPERANDS"                *     *     *     *     *
  309    :INTEGER;                                                                      S
  310
  311
  312 "############################"
  313 "COMMON TEST OUTPUT MECHANISM"
  314 "############################"
  315
  316 PRINTED: INTEGER;                                                                 *     S
  317
  318 OK: BOOLEAN;                                                                      *     S
  319   "PASS1 TO 6:  OK = NOT DISK OVERFLOW
C 320    PASS7:       OK = NOT DISK OVERFLOW & PROGRAM CORRECT"
  321
  322 PAGE_IN: PAGE;  PAGES_IN, WORDS_IN: INTEGER;                                      *    96     *     *     S
  323 PAGE_OUT: PAGE;  PAGES_OUT, WORDS_OUT: INTEGER;                                   *    96     *     *     S
  324 IFPAGE_OUT: PAGE;  IFPAGES_OUT, IFWORDS_OUT: INTEGER;                             *    96     *     *     S
  325
  326 PROCEDURE PRINT_TEXT (TEXT: TEXT_TYPE);                                           *     *    92
  327 VAR I: INTEGER;                                                                   *     S
  328 BEGIN
  329   WRITE(EOL);                                                                    99    25
  330   FOR I:= 1 TO TEXT_LENGTH DO WRITE(TEXT(.I.));                                 327    90    99   326   327
  331   WRITE(EOL)                                                                     99    25
  332 END;
  333
  334 PROCEDURE FILE_LIMIT;                                                             *
  335 BEGIN
  336   PRINT_TEXT('PASS 6: FILE_LIMIT');                                             326
  337   OK:= FALSE                                                                    318     *
  338 END;
  339
  340 PROCEDURE INIT_PASS (VAR LINK: PASSPTR);                                          *     *    69
  341 BEGIN
  342   LINK:= PARAM(.2.).PTR;                                                        340   108    86
  343   OK:= TRUE;                                                                    318   242
  344   PAGES_IN:= 1; WORDS_IN:= PAGELENGTH;                                          322   322    94
  345   PAGES_OUT:= 1; WORDS_OUT:= 0;                                                 323   323
  346   IFPAGES_OUT := 1; IFWORDS_OUT := 0;                                           324   324
  347 END;
  348
  349 PROCEDURE NEXT_PASS (LINK: PASSPTR);                                              *     *    69
  350 BEGIN
  351   IF WORDS_OUT > 0                                                              323
  352   THEN IF PAGES_OUT > FILE_LENGTH(OUTFILE)                                      323   104   250
  353         THEN FILE_LIMIT                                                         334
  354         ELSE PUT(OUTFILE, PAGES_OUT, PAGE_OUT);                                 103   250   323   323
  355   IF IFWORDS_OUT > 0                                                            324
  356   THEN IF IFPAGES_OUT > FILE_LENGTH(INTERFACEFILE)                              324   104   250
  357         THEN FILE_LIMIT                                                         334
  358         ELSE PUT(INTERFACEFILE, IFPAGES_OUT, IFPAGE_OUT);                       103   250   324   324
  359   WITH PARAM(.1.) DO BEGIN                                                      108
```

```
360     TAG:= BOOLTYPE; BOOL:=OK END;                                         82    79    83   318
361   WITH PARAM(.2.) DO BEGIN                                               108
362     TAG:= PTRTYPE; PTR:= LINK END;                                        82    86    86   349
363   WITH PARAM(.4.) DO BEGIN                                    .          108
364     TAG:= INTTYPE;  INT:= PAGES_OUT  END;                                 82    84    84   323
365 END;
366
367 PROCEDURE READ_IFL (VAR I: INTEGER);                                       *     *     S
368 BEGIN
369   IF WORDS_IN = PAGELENGTH THEN BEGIN                                    322    94
370     IF PAGES_IN > FILE_LENGTH(INFILE) THEN FILE_LIMIT                    322   104   250   334
371     ELSE BEGIN
372       GET(INFILE, PAGES_IN, PAGE_IN);                                    102   250   322   322
373       PAGES_IN:= SUCC(PAGES_IN)                                          322     *   322
374     END;
375     WORDS_IN:= 0                                                         322
376   END;
377   WORDS_IN:= SUCC(WORDS_IN);                                             322   373   322
378   I:= PAGE_IN(.WORDS_IN.)                                                367   322   322
379 END;
380
381 PROCEDURE WRITE_IFL (I: INTEGER);                                          *     *     S
382 BEGIN
383   IF GENNINGINTFAC                                                       280
384   THEN BEGIN
385      IFWORDS_OUT := SUCC(IFWORDS_OUT);                                   324   373   324
386      IFPAGE_OUT[IFWORDS_OUT] := I;                                       324   324   381
387      IF IFWORDS_OUT = PAGELENGTH                                         324    94
388      THEN BEGIN
389         IF IFPAGES_OUT > FILE_LENGTH(INTERFACEFILE)                      324   104   250
390         THEN FILE_LIMIT                                    .             334
391         ELSE BEGIN
392            PUT (INTERFACEFILE, IFPAGES_OUT, IFPAGE_OUT);                 103   250   324   324
393            IFPAGES_OUT := SUCC(IFPAGES_OUT)                              324   373   324
394            END;
395         IFWORDS_OUT := 0                                                 324
396         END
397      END
398   ELSE BEGIN
399      WORDS_OUT:= SUCC(WORDS_OUT);                                        323   373   323
400      PAGE_OUT(.WORDS_OUT.):= I;                                          323   323   381
401      IF WORDS_OUT = PAGELENGTH                                           323    94
402      THEN BEGIN
403         IF PAGES_OUT > FILE_LENGTH(OUTFILE)                              323   104   250
404         THEN FILE_LIMIT                                                  334
405         ELSE BEGIN
406            PUT(OUTFILE, PAGES_OUT, PAGE_OUT);                            103   250   323   323
407            PAGES_OUT:= SUCC(PAGES_OUT)                                   323   373   323
408            END;
409         WORDS_OUT:= 0                                                    323
410         END
411      END                                                                        .
412 END;
413
414 PROCEDURE PRINTABS(ARG: INTEGER);                                          *     *     S
415 VAR X: ARRAY (.1..MAXDIGIT.) OF CHAR; REM, DIGIT, I: INTEGER;              *    26     S     *     *     *     S
416 BEGIN
417   REM:= ARG; DIGIT:= 0;                                                  415   414   415
418   REPEAT
419     DIGIT:= DIGIT + 1;                                                   415   415
```

```
420     T(.DIGIT.):= CHR(ABS(REM MOD 10) + ORD('0'));                  415  415   *    *   415   *
421     REM:= REM DIV 10;                                              415  415
422   UNTIL (REM=0) OR (DIGIT>=MAXDIGIT);                              415  415   26
423   FOR I:= DIGIT DOWNTO 1 DO WRITE(T(.I.));                         415  415   99  415  415
424   FOR I:= DIGIT + 1 TO MAXDIGIT DO WRITE(' ');                     415  415   26   99
425 END;
426
427 PROCEDURE PRINTEOL;                                                 *
428 BEGIN WRITE(EOL); PRINTED:= 0 END;                                 99   25  316
429
430 PROCEDURE PRINTFF;                                                  *
431 VAR I:INTEGER;                                                      *    S
432 BEGIN
433   PRINTEOL; FOR I:=1 TO 130 DO WRITE('6'); PRINTEOL                427  431   99  427
434 END;
435
436 PROCEDURE PRINTOP(OP: INTEGER);                                     *    *    S
437 BEGIN
438   IF PRINTED = PRINTLIMIT THEN PRINTEOL;                           316   27  427
439   WRITE('C'); PRINTABS(OP);                                         99  414  436
440   PRINTED:= PRINTED + 1;                                           316  316
441 END;
442
443 PROCEDURE PRINTARG(ARG: INTEGER);                                   *    *    S
444 BEGIN
445   IF PRINTED = PRINTLIMIT THEN PRINTEOL;                           316   27  427
446   IF ARG < 0 THEN WRITE('-') ELSE WRITE(' ');                      443   99   99
447   PRINTABS(ARG);                                                   414  443
448   PRINTED:= PRINTED + 1;                                           316  316
449 END;
450
451 "################"
452 "INPUT PROCEDURES"
453 "################"
454
455 PROCEDURE READ1ARG;                                                 *
456 BEGIN READ_IFL(ARG1) END;                                          367  308
457
458 PROCEDURE READ2ARG;                                                 *
459 BEGIN READ_IFL(ARG1); READ_IFL(ARG2) END;                          367  308  367  308
460
461 PROCEDURE READ3ARG;                                                 *
462 BEGIN READ_IFL(ARG1); READ_IFL(ARG2); READ_IFL(ARG3) END;          367  308  367  308  367  308
463
464 PROCEDURE READ4ARG;                                                 *
465 BEGIN
466   READ_IFL(ARG1); READ_IFL(ARG2);                                  367  308  367  308
467   READ_IFL(ARG3); READ_IFL(ARG4);                                  367  308  367  308
468 END;
469
470 PROCEDURE READ5ARG;                                                 *
471 BEGIN
472   READ_IFL(ARG1); READ_IFL(ARG2); READ_IFL(ARG3);                  367  308  367  308  367  308
473   READ_IFL(ARG4); READ_IFL(ARG5)                                   367  308  367  308
474 END;
475
476 "#################"
477 "OUTPUT PROCEDURES"
478 "#################"
479
```

```
  480 PROCEDURE ERROR (PASS, NUMBER: INTEGER); FORWARD;                        *    *    *    S    *
  481
  482 PROCEDURE UPDLOC (CODE: TYPEOFCODE; VALUE: INTEGER);                     *    *   269   *    S
  483 BEGIN  "UPDATE THE OUTPUT CODE LOCATION COUNTER."
  484 IF LOCATION < CODE_LIMIT                                               303  248
  485 THEN CASE CODE OF                                                      482
  486      OPTR, UB, SB, DB: LOCATION := LOCATION + HALFWORD;                269  269  269  269  303  303  245
  487      B: IF VALUE <= 127                                                269  482
  488         THEN LOCATION := LOCATION + HALFWORD                           303  303  245
  489         ELSE LOCATION := LOCATION + ONEWORD;                           303  303  246
  490      W: LOCATION := LOCATION + ONEWORD;                                269  303  303  246
  491      NOTME:                                                            269
  492      END
  493 ELSE BEGIN
  494    ERROR (THIS_PASS, CODE_ERROR);                                      480  249  256
  495    LOCATION := 0;                                                      303
  496    END;
  497 END;
  498
  499 "LOCATION COUNTER IS KEPT FOR THE CONCURRENT SEGMENT ONLY.
C 500  IF CODE OUTPUT ROUTINES ARE CALLED WHILE GENERATING AN INTERFACE
C 501  SEGMENT (GENNINGINTFAC = TRUE) THEN DON'T FOOL WITH CONCURRENT CODE
C 502  LOCATION COUNTER."
  503
  504 PROCEDURE WRITE1(OP: INTEGER);                                           *    *    S
  505 BEGIN
  506   IF TEST THEN PRINTOP(OP);                                            276  436  504
  507   WRITE_IFL(OP);                                                       381  504
  508   IF NOT GENNINGINTFAC THEN UPDLOC (OPTR, OP)                          280  482  269  504
  509 END;
  510
  511 PROCEDURE WRITE2(OP, ARG: INTEGER; ARGTYP: TYPEOFCODE);                  *    *    *    S    *   269
  512 BEGIN
  513   IF TEST                                                              276
  514   THEN BEGIN
  515      PRINTOP(OP); PRINTARG(ARG)                                        436  511  443  511
  516      END;
  517   WRITE_IFL(OP); WRITE_IFL(ARG);                                       381  511  381  511
  518   IF NOT GENNINGINTFAC                                                 280
  519   THEN BEGIN
  520      UPDLOC(OPTR, OP); UPDLOC(ARGTYP, ARG)                             482  269  511  482  511  511
  521      END;
  522 END;
  523
  524 PROCEDURE WRITE3(OP, ARG1, ARG2: INTEGER;                                *    *    *    *    S
  525                  ARG1TYP, ARG2TYP: TYPEOFCODE);                          *    *   269
  526 BEGIN
  527   IF TEST                                                              276
  528   THEN BEGIN
  529      PRINTOP(OP);                                                      436  524
  530      PRINTARG(ARG1); PRINTARG(ARG2);                                   443  524  443  524
  531      END;
  532   WRITE_IFL(OP);                                                       381  524
  533   WRITE_IFL(ARG1); WRITE_IFL(ARG2);                                    381  524  381  524
  534   IF NOT GENNINGINTFAC                                                 280
  535   THEN BEGIN
  536      UPDLOC(OPTR, OP);                                                 482  269  524
  537      UPDLOC(ARG1TYP, ARG1); UPDLOC(ARG2TYP, ARG2)                      482  525  524  482  525  524
  538      END
  539 END;
```

```
540
541 PROCEDURE WRITE4(OP, ARG1, ARG2, ARG3: INTEGER;                       *      *      *      *      *      S
542                      ARG1TYP, ARG2TYP, ARG3TYP: TYPEOFCODE);           *      *      *      269
543 BEGIN
544   IF TEST THEN                                                        276
545   BEGIN
546     PRINTOP(OP); PRINTARG(ARG1);                                      436    541    443    541
547     PRINTARG(ARG2); PRINTARG(ARG3);                                   443    541    443    541
548     END;
549   WRITE_IFL(OP); WRITE_IFL(ARG1);                                     381    541    381    541
550   WRITE_IFL(ARG2); WRITE_IFL(ARG3);                                   381    541    381    541
551   IF NOT GENNINGINTFAC                                                280
552   THEN BEGIN
553      UPDLOC(OPTR, OP); UPDLOC(ARG1TYP, ARG1);                         482    269    541    482    542    541
554      UPDLOC(ARG2TYP, ARG2); UPDLOC(ARG3TYP, ARG3)                     482    542    541    482    542    541
555      END;
556 END;
557
558 PROCEDURE WRITE5(OP, ARG1, ARG2, ARG3, ARG4: INTEGER;                 *      *      *      *      *      *      S
559                      ARG1TYP, ARG2TYP, ARG3TYP, ARG4TYP: TYPEOFCODE);  *      *      *      *      269
560 BEGIN
561   IF TEST                                                             276
562   THEN BEGIN
563      PRINTOP(OP);                                                     436    558
564      PRINTARG(ARG1); PRINTARG(ARG2);                                  443    558    443    558
565      PRINTARG(ARG3); PRINTARG(ARG4);                                  443    558    443    558
566      END;
567   WRITE_IFL(OP);                                                      381    558
568   WRITE_IFL(ARG1); WRITE_IFL(ARG2);                                   381    558    381    558
569   WRITE_IFL(ARG3); WRITE_IFL(ARG4);                                   381    558    381    558
570   IF NOT GENNINGINTFAC                                                280
571   THEN BEGIN
572      UPDLOC(OPTR, OP);                                                482    269    558
573      UPDLOC(ARG1TYP, ARG1); UPDLOC(ARG2TYP, ARG2);                    482    559    558    482    559    558
574      UPDLOC(ARG3TYP, ARG3); UPDLOC(ARG4TYP, ARG4)                     482    559    558    482    559    558
575      END
576 END;
577
578 PROCEDURE WRITEARG(ARG: INTEGER; ARGTYP: TYPEOFCODE);                 *      *      S      *      269
579 BEGIN
580   IF TEST THEN PRINTARG(ARG);                                         276    443    578
581   WRITE_IFL(ARG);                                                     381    578
582   IF NOT GENNINGINTFAC THEN UPDLOC(ARGTYP, ARG)                       280    482    578    578
583 END;
584
585 PROCEDURE WRITELOCATION;                                              *
586 BEGIN
587   IF TEST THEN PRINTARG(LOCATION);                                    276    443    303
588   WRITE_IFL(LOCATION);                                                381    303
589 END;
590
591 PROCEDURE ERROR;                                                      *
592 BEGIN
593   IF NOT AFTERERROR                                                   278
594   THEN BEGIN
595        AFTERERROR:= TRUE;                                             278    242
596        WRITEARG(MESSAGE_2, NOTME);                                    578    217    269
597        WRITEARG(PASS, NOTME);                                         578    480    269
598        WRITEARG(NUMBER, NOTME);                                       578    480    269
599        WRITEARG(LINE, NOTME);                                         578    306    269
```

```
600        GENERATE:= FALSE                                                        277   337
601        END
602 END;
603
604 "################################"
605 "PSUEDO RUN-TIME STACK PROCEDURES"
606 "################################"
607
608 PROCEDURE PUSHWORD;                                                             *
609 BEGIN
610   IF TEMP < STACK_LIMIT THEN TEMP:= TEMP + WORDLENGTH                          300   248   300   300    28
611     ELSE ERROR(THIS_PASS, STACK_ERROR);                                        591   249   256
612   IF TEMP > MAXTEMP THEN MAXTEMP:= TEMP;                                       300   301   301   300
613 END;
614
615 PROCEDURE POPWORD;                                                              *
616 BEGIN TEMP:= TEMP - WORDLENGTH ;                                               300   300    28
617 END;
618
619 PROCEDURE PUSHREAL;                                                             *
620 BEGIN
621   IF TEMP < STACK_LIMIT THEN TEMP:= TEMP + REALLENGTH                          300   248   300   300    29
622     ELSE ERROR(THIS_PASS, STACK_ERROR);                                        591   249   256
623   IF TEMP > MAXTEMP THEN MAXTEMP:= TEMP;                                       300   301   301   300
624 END;
625
626 PROCEDURE POPREAL;                                                              *
627 BEGIN TEMP:= TEMP - REALLENGTH END;                                            300   300    29
628
629 PROCEDURE PUSHSET;                                                              *
630 BEGIN
631   IF TEMP < STACK_LIMIT THEN TEMP:= TEMP + SETLENGTH                           300   248   300   300    30
632     ELSE ERROR(THIS_PASS, STACK_ERROR);                                        591   249   256
633   IF TEMP > MAXTEMP THEN MAXTEMP:= TEMP;                                       300   301   301   300
634 END;
635
636 PROCEDURE POPSET;                                                               *
637 BEGIN TEMP:= TEMP - SETLENGTH END;                                             300   300    30
638
639 PROCEDURE PUSH(LENGTH: INTEGER);                                                *     *     S
640 BEGIN
641   IF TEMP < STACK_LIMIT - LENGTH THEN TEMP:= TEMP + LENGTH                     300   248   639   300   300   639
642     ELSE ERROR(THIS_PASS, STACK_ERROR);                                        591   249   256
643   IF TEMP > MAXTEMP THEN MAXTEMP:= TEMP;                                       300   301   301   300
644 END;
645
646 PROCEDURE POP(LENGTH: INTEGER);                                                 *     *     S
647 BEGIN TEMP:= TEMP - LENGTH END;                                                300   300   646
648
649 "#####################"
650 " SHORT CONSTANT PUSH "
651 "#####################"
652
653 PROCEDURE PICK_PUSHCONST (VALUE: INTEGER);                                      *     *     S
654 BEGIN   "PICK 1 OF 3 OPERATORS WHICH PUSH IMMEDIATE CONSTANTS"
655 IF (VALUE >= 0) AND (VALUE <= 31)                                              653   653
656 THEN WRITE1(VALUE)   "THE OP IS AN SLDCXX_02"                                  504   653
657 ELSE IF (VALUE >= 32) AND (VALUE <= 255)                                       653   653
658      THEN WRITE2(LDCB_2, VALUE, UB)                                            511   191   653   269
659      ELSE WRITE2(LDCI_2, VALUE, W)                                             511   191   653   269
```

```
   660 END;
   661
   662 "####################"
   663 "VARIABLE PROCEDURES"
   664 "####################"
   665
   666 FUNCTION DISPL(MODE, ARG: INTEGER): INTEGER;                             *     *     *     S     S
   667 "HARTMANN'S VARIABLES HAVE NEGATIVE DISPLACEMENT, PARAMETERS HAVE
C  668  POSITIVE DISPLACEMENTS, BOTH IN BYTES & 2-BASED.
C  669  FOR MICROENGINE, BOTH VARIABLES AND PARAMETERS HAVE
C  670  POSITIVE WORD DISPLACEMENTS (1-BASED). VARS HAVE SMALLER DISPLS
C  671  SINCE THEY ARE CLOSER TO THE MSCW (PUSHED LATER) THAN PARMS."
   672 BEGIN
   673 CASE MODE OF                                                             666
   674      MODE0: "CONSTANT"                                                   139
   675         DISPL := (ARG + ONEWORD) DIV WORDLENGTH;                         666   666   246    28
   676
   677      "PROCEDURE, PROGRAM, CLASS ENTRY, MONITOR ENTRY, MGR ENTRY"
   678      MODE1, MODE2, MODE4, MODE5, MODE12:                                 140   141   143   144   151
   679         IF ARG < 0                                                       666
   680         THEN DISPL := (-ARG + TWOWORDS) DIV WORDLENGTH                   666   666   247   '28
   681         ELSE DISPL := (ARG + VARLENGTH + TWOWORDS) DIV WORDLENGTH;       666   666   297   247    28
   682
   683      MODE3: "PROCESS ENTRY"                                              142
   684      "INTERFACE PROCEDURE HAS NO LOCAL VARS (NOT EVEN LINE & OLD
C  685      GLOBAL BASE). HOWEVER, IT DOES HAVE AN EXTRA PARM (THE NO.
C  686      OF THE PREFIX PROCEDURE BEING CALLED) WHICH HAS NOT BEEN
C  687      TAKEN INTO ACCOUNT IN DISPLACEMENT CALCULATION TO THIS POINT.
C  688      IF ARG < 0 THEN REFERENCE IS TO A LOCAL VARIABLE IN THE ENTRY
C  689      PROCEDURE. OTHERWISE, REFERENCE IS TO PARM IN THE INTERFACE
C  690      PROCEDURE."
   691         IF ARG < 0                                                       666
   692         THEN DISPL := (-ARG + TWOWORDS) DIV WORDLENGTH                   666   666   247    28
   693         ELSE DISPL := (ARG + ONEWORD) DIV WORDLENGTH;                    666   666   246    28
   694
   695      "PROCESS, CLASS, MONITOR, MANAGER MODES"
   696      MODE6, MODE7, MODE8, MODE11:                                        145   146   147   150
   697     ."IN THESE MODES, REFERENCE IS TO 'SHARED' VARIABLES OR PARMS
C  698      TO THE INITIAL STATEMENT. THE RECORD CONTAINING THESE VARS
C  699      & PARMS IS STORED IN THE SHARED VARIABLE AREA OF THE INITIAL
C  700      PROCESS. SPACE FOR THE RECORD WAS ALLOCATED IN PRIOR PASSES.
C  701      THE GLOBAL RECORDS DO NOT CONTAIN TWO WORDS FOR LINE NO. &
C  702      OLD GLOBAL BASE."
   703         IF ARG < 0                                                       666
   704         THEN DISPL := (-ARG) DIV WORDLENGTH                              666   666    28
   705         ELSE DISPL := (ARG + VARLENGTH) DIV WORDLENGTH;                  666   666   297    28
   706
   707      MODE10: "UNDEFINED MODE"                                            149
   708 END
   709 END;
   710
   711 PROCEDURE PUSHVALUE(MODE, ARG: INTEGER);                                 *     *     *     S
   712 BEGIN
   713   CASE MODE OF                                                           711
   714
   715     "PROCEDURE, CLASS ENTRY, MONITOR ENTRY, MGR ENTRY MODES"
   716     MODE1, MODE4, MODE5, MODE12:                "MGR*"                   140   143   144   151
   717       WRITE2(LDL_2, DISPL(MODE, ARG), B);                                511   192   666   711   711   269
   718
   719     MODE2:   "PROGRAM MODE"                                              141
```

```
720       WRITE2(LDO_2, DISPL(MODE, ARG), B);                                 511   192   666   711   711   269
721
722     MODE3:   "PROCESS ENTRY MODE"                                         142
723       IF ARG < 0                                                          711
724       THEN WRITE2(LDL_2, DISPL(MODE, ARG), B)                             511   192   666   711   711   269
725       ELSE WRITE3(LOD_2, 1, DISPL(MODE, ARG), DB, B);                     524   193   666   711   711   269   269
726
727     "PROCESS, CLASS, MONITOR, MANAGER MODES"
728     MODE6, MODE7, MODE8, MODE11:                       "MGR*"             145   146   147   150
729       WRITE2(LDO_2, DISPL(MODE, ARG), B);                                 511   192   666   711   711   269
730
731     MODE10:    "UNDEFINED MODE"                                           149
732   END;
733   PUSHWORD;                                                               608
734 END;
735
736 PROCEDURE PUSHADDRESS(MODE, ARG: INTEGER);                                 *     *     *     S
737 BEGIN
738   CASE MODE OF                                                            736
739     MODE0:     "CONSTANT MODE"                                            139
740       WRITE2(LCA_2, DISPL(MODE, ARG), B);                                 511   191   666   736   736   269
741
742     "PROCEDURE, CLASS ENTRY, MON. ENTRY, MGR. ENTRY"
743     MODE1, MODE4, MODE5, MODE12:  "MGR*"                                  140   143   144   151
744       WRITE2(LLA_2, DISPL(MODE, ARG), B);                                 511   192   666   736   736   269
745
746     MODE2:    "PROGRAM MODE"                                              141
747         WRITE2(LAO_2, DISPL(MODE, ARG), B);                               511   192   666   736   736   269
748
749     MODE3: "PROCESS ENTRY MODE"                                           142
750         IF ARG < 0                                                        736
751         THEN WRITE2(LLA_2, DISPL(MODE, ARG), B)                           511   192   666   736   736   269
752         ELSE WRITE3(LDA_2, 1, DISPL(MODE, ARG), DB, B);                   524   193   666   736   736   269   269
753
754     "PROCESS, CLASS, MONITOR, MANAGER MODES"
755     MODE6, MODE7, MODE8, MODE11:  "MGR*"                                  145   146   147   150
756       WRITE2(LAO_2, DISPL(MODE, ARG), B);                                 511   192   666   736   736   269
757
758     MODE10:    "UNDEFINED MODE"                                           149
759   END;
760   PUSHWORD;                                                               608
761 END;
762
763 PROCEDURE PUSHINDIRECT(VARTYPE: INTEGER);                                  *     *     S
764 BEGIN
765   CASE VARTYPE OF                                                         763
766     BYTETYPE:                                                             135
767       BEGIN   WRITE1(LDB_2);    POPWORD   END;                            504   200   615
768     WORDTYPE:                                                             135
769       WRITE1(SINDO_2);                                                    504   189
770     REALTYPE:                                                             135
771       BEGIN WRITE2(LDM_2, REALLENGTH DIV WORDLENGTH, UB);                 511   210    29    28   269
772         POPWORD; PUSHREAL;                                                615   619
773       END;
774     SETTYPE:                                                              135
775       BEGIN WRITE2(LDM_2, SETLENGTH DIV WORDLENGTH, UB);                  511   210    30    28   269
776         POPWORD; PUSHSET;                                                 615   629
777         PICK_PUSHCONST(SETLENGTH DIV WORDLENGTH);                         653    30    28
778         PUSHWORD;   "PUSH WORD LENGTH OF SET"                             608
779       END
```

```
780   END;
781 END;
782
783
784
785 "#################"
786 "TABLE PROCEDURES"
787 "#################"
788
789 PROCEDURE ALLOCATE(VAR T: TABLEPTR; ENTRIES: INTEGER);                        *     *    48     *     S
790 VAR PORTION: TABLEPTR; I: INTEGER;                                            *    48     *     S
791 BEGIN
792   NEW(T); PORTION:= T;                                                        *   789   790   789
793   I:= ENTRIES - MAXWORD;                                                    790   789    40
794   WHILE I > 0 DO                                                            790
795   WITH PORTION@ DO                                                          790
796   BEGIN
797     NEW(NEXTPORTION); PORTION:= NEXTPORTION;                                792    50   790    50
798     I:= I - MAXWORD;                                                        790   790    40
799   END;
800 END;
801
802 PROCEDURE ENTER(T: TABLEPTR; I, J: INTEGER);                                  *     *    48     *     *     S
803 VAR PORTION: TABLEPTR; K: INTEGER;                                            *    48     *     S
804 BEGIN
805   PORTION:= T; K:= I;                                                       803   802   803   802
806   WHILE K > MAXWORD DO                                                      803    40
807   BEGIN
808     PORTION:= PORTION@.NEXTPORTION;                                         803   803    50
809     K:= K - MAXWORD;                                                        803   803    40
810   END;
811   PORTION@.CONTENTS(.K.):= J;                                               803    51   803   802
812 END;
813
814 FUNCTION ENTR(T: TABLEPTR; I: INTEGER): INTEGER;                              *     *    48     *     S     S
815 VAR PORTION: TABLEPTR; J: INTEGER;                                            *    48     *     S
816 BEGIN
817   PORTION:= T; J:= I;                                                       815   814   815   814
818   WHILE J > MAXWORD DO                                                      815    40
819   BEGIN
820     PORTION:= PORTION@.NEXTPORTION;                                         815   815    50
821     J:= J - MAXWORD;                                                        815   815    40
822   END;
823   ENTR:= PORTION@.CONTENTS(.J.);                                            814   815    51   815
824 END;
825
826 "##########################################"
827 "INITIALIZATION AND TERMINATION PROCEDURES"
828 "##########################################"
829
830 PROCEDURE BEGINPASS;                                                          *
831 VAR I: INTEGER;                                                               *     S
832 BEGIN
833   WITH LINK@ DO                                                             349
834   BEGIN
835     SUMMARY:= SUMMARYOPTION IN OPTIONS;                                     276    38    71
836     TEST:= TESTOPTION IN OPTIONS;                                           276    38    71
837     CHECK:= CHECKOPTION IN OPTIONS;                                         276    38    71
838     VARNTCHECK:= VARNTCHECKOPTION IN OPTIONS;                   "VARNT*"    278    39    71
839     NUMBER:= NUMBEROPTION IN OPTIONS;                                       480    39    71
```

```
  840      GENERATE:= TRUE;                                                   277   242
  841      FOR I := 1 TO MAXINTFAC DO                                         831    61
  842         IFSEGSIZE[I] := INTERFACE@.INTERFACESIZES[I];                   286   831    75    66   831
  843      ALLOCATE(JUMPTABLE, LABELS);                                       789   282    72
  844      ALLOCATE(CONSTTABLE, CONSTANTS DIV WORDLENGTH);                    789   282   288    28
  845      ALLOCATE(XJPTABLE, XJP_OFFSETS DIV WORDLENGTH);                    789   282    72    28
  846      ALLOCATE(EXITICTABLE, BLOCKS);                                     789   283    72
  847      ALLOCATE(DATASIZETABLE, BLOCKS);                                   789   283    72
  848      MARK (TABLERESET);                                                 105   274
  849      ALLOCATE(STACKTABLE, BLOCKS);                                      789   282    72
  850    END;
  851    CONSTANTS:= 0;                                                       288
  852    XJPOFFSETPTR := 0;                                                   289
  853    POOLSIZE := LINK@.CONSTANTS + LINK@.XJP_OFFSETS;                     290   349   288   349    72
  854    LINE := 0;                                                           306
  855    LOCATION := ONEWORD + POOLSIZE;                                      303   246   290
  856    AFTERBEGIN := FALSE;                                                 277   337
  857    IFSEGNUM := 0;                                                       305
  858    GENNINGINTFAC := FALSE;                                              280   337
  859    "LABEL 1 (TARGET OF 1ST (DISCARDED) JUMP) IS NEVER RESOLVED"
  860    ENTER(JUMPTABLE, STARTPROC, -1);                                     802   282   255
  861    IF TEST THEN PRINTFF;                                                276   430
  862 END;
  863
  864 PROCEDURE ENDPASS;                                                        *
  865 BEGIN
  866      LINK@.CONSTANTS := CONSTANTS;                                      349   288   288
  867      LINK@.XJP_OFFSETS := XJPOFFSETPTR;                                 349    72   289
  868      WITH LINK@ DO                                                      349
  869        IF GENERATE THEN OPTIONS:= OPTIONS OR (.CODEOPTION.);            277    71    71    39
  870      RELEASE (TABLERESET);  "REMOVE TABLES NOT NEEDED IN PASS7"         106   274
  871      NEW(LINK@.TABLES);                                                 792   349    74
  872      WITH LINK@ DO                                                      349
  873         BEGIN
  874         "SEGDISTANCE IS LOC. CTR. + SPACE FOR PROCEDURE DICTIONARY.
C 875         'LOCATION' WILL THEN POINT TO LO ORDER BYTE IN THE
C 876         (LAST) WORD WHICH WILL CONTAIN NO. OF PROCS. IN SEG. &
C 877         SEGMENT NUMBER."
  878         TABLES@.SEGDISTANCE := LOCATION + 2*LINK@.BLOCKS;                74    54   303   349    72
  879         TABLES@.JUMPTABLE:=JUMPTABLE;                                    74   282   282
  880         TABLES@.CONSTTABLE:=CONSTTABLE;                                  74   282   282
  881         TABLES@.XJPTABLE := XJPTABLE;                                    74   282   282
  882         TABLES@.EXITICTABLE := EXITICTABLE;                              74   283   283
  883         TABLES@.DATASIZETABLE := DATASIZETABLE;                          74   283   283
  884         END
  885 END;
  886
  887 "%IF SRC_32"
  888 PROCEDURE SPLIT_CONST(WORD:UNIV SPLIT_INTEGER;                            *     *   261
  889             VAR HIWORD,LOWORD: INTEGER);                                  *     *     S
  890 BEGIN "SPLIT A 32-BIT INTEGER INTO TWO 16-BIT INTEGERS"
  891   HIWORD:=WORD[1]; LOWORD:=WORD[2];                                     889   888   889   888
  892 END;
  893 "%END"
  894
  895 PROCEDURE GEN_SAVELINE_SAVEGBASE;                                         *
  896 "PROLOGUE FOR ROUTINES"
  897 BEGIN
  898 IF NUMBER                                                               480
  899 THEN BEGIN
```

```
    900      WRITEARG (NEWLIN_2, NOTME); "CRUTCH FOR MNEMONICS PGM."        578   217   269
    901      PICK_PUSHCONST (LINE);           PUSHWORD;                     653   306   608
    902      WRITE2 (STL_2, FIXLCL_LINE, B); POPWORD                        511   200   238   269   615
    903      END;
    904 PICK_PUSHCONST (GBASE_REG);           PUSHWORD;                     653   234   608
    905 WRITE1 (LPR_2);                                                     504   198
    906 WRITE2 (STL_2, FIXLCL_OLDG, B);       POP(TWOWORDS)                 511   200   239   269   646   247
    907 END;
    908
    909 PROCEDURE GEN_EXIT;                                                  *
    910 "EPILOGUE FOR ROUTINES"
    911 BEGIN
    912    PICK_PUSHCONST (GBASE_REG);         PUSHWORD;                    653   234   608
    913    WRITE1 (SLDL02_2);                  PUSHWORD;                    504   180   608
    914    WRITE1 (SPR_2);                     POP(TWOWORDS);               504   210   646   247
    915    WRITE2 (RPU_2, POPLENGTH DIV WORDLENGTH, B);                     511   196   299    28   269
    916    POP(POPLENGTH + MSCWSIZE)                                        646   299   241
    917 END;
    918
    919
    920 "#################################
C   921  # GENERATE AN INTERFACE SEGMENT #
C   922  #################################"
    923
    924 PROCEDURE GEN_INTERFACE;                                             *
    925 TYPE
    926    TOSPTR = @STKENTRY;                                               *     *
    927    STKENTRY = RECORD                                                 *
    928       DATA: INTEGER;                                                 *     S
    929       NXTENTRY: TOSPTR  "POINTS TO THING PUSHED JUST BEFORE"         *   926
    930       END;
    931 CONST
    932    CONSEG = 1; "CONCURRENT SEGMENT IS SEGMENT 1"                     *
    933    HIORDER = 256; "SHIFTS SMALL INT. TO HI-ORDER BYTE IN 16-BIT WORD"  *
    934    MININDEX = 1; "MIN INDEX FOR CASE STMT. I.E., PREFIX RTNS ARE     *
C   935                   NUMBERED STARTING AT 1"
    936    DATASIZE = 0; "ROUTINE BEING GEN'D HAS NO LOCAL VAR SPACE"        *
    937    PARMWORDS = 1; "ROUTINE BEING GEN'D NEEDS 1 PARAMETER"            *
    938    XJPTABLE = 1; "TABLE FOR CASEJUMP INSTRUC STARTS IN WORD 1 OF SEG."  *
    939    PROCSGEND = 1; "INTERFACE SEGS CONSIST OF ONLY 1 PROCEDURE"       *
    940 VAR
    941    OLDHEAPTOP: FULLWORD;                                             *    18
    942    RTNS, "NO. OF ROUTINE LABELS IN THIS INTERFACE"                   *
    943    KASE, "LOOP VAR FOR RTNS & CASES IN CASE STMNT"                   *
    944    IFLABEL: INTEGER; "INTERFACE ROUTINE LABEL"                       *     S
    945    TOS: TOSPTR; "POINTS TO CURRENT TOP OF STACK"                     *   926
    946    NEWTOS: TOSPTR; "POINTS TO THING TO BE PUSHED NEXT"               *   926
    947 BEGIN                                                               *
    948 GENNINGINTFAC := TRUE; "FROM NOW TO END OF THIS PROC, SEND          280   242
C   949                        OUTPUT TO THE FILE OF INTERFACE SEGS"
    950 IFSEGNUM := SUCC(IFSEGNUM);                                         305   373   305
    951 RTNS := IFSEGSIZE[IFSEGNUM]; "GET NO. OF ROUTINES IN THIS INTFACE."  942   286   305
    952
    953 WRITEARG ( (RTNS*8+22) DIV 2, W); "WORDS IN SEG, LESS 1"            578   942   269
    954
    955 "GEN CONSTANT BLOCK. IT CONSISTS ONLY OF THE TABLE FOR CASE STMT."
    956 WRITEARG (MININDEX, W); "MIN INDEX OF CASE STMT"                    578   934   269
    957 WRITEARG (RTNS, W); "MAX INDEX OF CASE STMT"                        578   942   269
    958 FOR KASE := 0 TO RTNS-1 DO WRITEARG (KASE*6+4, W); "OFFSETS"        943   942   578   943   269
    959
    960 "GENERATE EXIT-IC FIELD. SEGMENT-RELATIVE POINTER, IN BYTES."
    961 WRITEARG (RTNS*8+18, W);                                          578   942   269
    962 WRITEARG (DATASIZE, W); "GEN DATASIZE FIELD"                      578   936   269
    963
    964 "GEN CODE TO USE THE NO. OF A PREFIX ROUTINE (THE CALLING
C   965  SEQUENTIAL PROGRAM MUST SUPPLY THIS AS A PARAMETER) AS THE INDEX
C   966  INTO CASE STMNT AND CALL CORRESPONDING CONCURRENT PROCESS
C   967  ENTRY ROUTINE."
    968 WRITE1 (SLDL01_2); "PUSH THE PARAMETER"                           504   180
    969 WRITE2 (XJP_2, XJPTABLE, B); "CASE JUMP"                          511   211   938   269
    970 WRITE1 (NOP_2); "GEN'D IN IMITATION OF WESTERN DIGITAL"           504   198
    971 WRITE2 (UJPL_2, 6*RTNS, W); "FOR THE CASE WHERE PARM IS OUT OF RANGE"   511   193   942   269
    972
    973 "THE INTERFACE ROUTINE LABELS COME INTO THIS PASS IN REVERSE ORDER.
C   974  THEY NEED TO BE IN FORWARD ORDER. STACK THEM AND POP THEM TO
C   975  MAKE THE CORRECTION."
    976
    977 MARK (OLDHEAPTOP);                                                105   941
    978 TOS := NIL; "THERE IS NO TOP OF STACK YET"                        945
    979 NEW (NEWTOS);                                                     792   946
    980 FOR KASE := 1 TO RTNS DO   "READ & STACK THE LABELS"              943   942
    981    BEGIN
    982    READ_IFL (IFLABEL);                                            367   944
    983    WITH NEWTOS@ DO                                                946
    984       BEGIN
    985       DATA :=IFLABEL; "SAVE THE LABEL IN A STACK NODE"            928   944
    986       NXTENTRY := TOS "TIE NODE TO EXISTING STACK (PUSH NODE)"    929   945
    987       END;
    988       TOS := NEWTOS; "KEEP TOS POINTER CURRENT"                   945   946
    989    NEW (NEWTOS);                                                  792   946
    990     "IF THE LABEL JUST HANDLED WAS NOT THE LAST ONE IN THIS INTERFACE,
C   991      THEN READ PAST THE 'PUSHLABEL' OPERATOR WHICH PRECEDES THE
C   992      NEXT LABEL. IF THIS WAS THE LAST LABEL, DON'T READ THE NEXT
C   993      OPERATOR, SINCE IT NEEDS TO BE HANDLED BY THE LARGE CASE
C   994      STATEMENT IN THE 'SCAN' ROUTINE."
    995    IF KASE <> RTNS                                                943   942
    996    THEN READ_IFL (OP)                                             367   558
    997    END;
    998
    999 FOR KASE := 1 TO RTNS DO   "GEN CODE FOR EACH CASE"               943   942
   1000    BEGIN
   1001    "GEN THE CALL TO THE ENTRY ROUTINE IN THE CONCURRENT PROCESS.
C  1002  MUST BE CALL EXTERNAL <LOCAL> SO THAT STATIC LINK IN PROCESS
C  1003  ENTRY PROCEDURE POINTS TO MARKSTACK FOR THIS (INTERFACE)
C  1004  PROCEDURE. WHEN REFERENCING PARMS, THE PROCESS ENTRY ROUTINE
C  1005  WILL USE <INTERMEDIATE> LOADS AND STORES. THIS ALLOWS THE
C  1006  ENTRY ROUTINE TO REFER TO PARMS IN THE INTERFACE PROCEDURE AS IF
C  1007  THEY HAD BEEN PASSED DIRECTLY TO THE ENTRY ROUTINE."
   1008    WRITE3 (CXL_2, CONSEG, TOS@.DATA, UB, UB);                     524   195   932   945   928   269   269
   1009    WRITE2 (UJPL_2, 6*(RTNS-KASE), W); "GEN JUMP OUT OF CASE CODE"   511   193   942   943   269
   1010    TOS := TOS@.NXTENTRY "POP THE STACK OF LABELS"                 945   945   929
   1011    END;
   1012
   1013 WRITE1 (NOP_2); "TO WORD-ALIGN"                                   504   198
   1014 WRITE2 (RPU_2, DATASIZE+PARMWORDS, B);                            511   196   936   937   269
   1015 RELEASE (OLDHEAPTOP);                                             106   941
   1016
   1017 WRITEARG (RTNS+3, W); "SEG-REL WORD ADDR OF DATASIZE FIELD"       578   942   269
   1018
   1019 "GEN A WORD IN WHICH THE HIGH ORDER BYTE CONTAINS THE NUMBER
```

```
C 1020  OF PROCEDURES IN THE SEGMENT JUST GEN'D, & THE LOW ORDER BYTE
C 1021  CONTAINS THE SEG NUMBER OF SEG JUST GEN'D. ASSUME THAT THE
C 1022  KERNEL IS SEGMENT 0, CONCURRENT PROGRAM IS SEGMENT 1, AND THAT
C 1023  INTERFACE SEGS ARE NUMBERED CONSECUTIVELY AFTER THAT. THEY WILL
C 1024  BE RENUMBERED BY THE KERNEL WHEN LOADED, BUT THESE NUMBERS
C 1025  SHOULD HELP IN TESTING SINCE WEST. DIG. SOFTWARE DOESN'T SUPPORT
C 1026  MORE THAN 16 SEGMENTS."
  1027 WRITEARG (PROCSGEND * HIORDER + (IFSEGNUM + CONSEG), W);             578   939   933   305   932   269
  1028
  1029 GENNINGINTFAC := FALSE "SEND OUTPUT TO CONCURRENT FILE"             280   337
  1030 END;
  1031
  1032
  1033
  1034 "#########"
  1035 "OPERATORS"
  1036 "#########"
  1037
  1038 PROCEDURE SCAN;                                                        *
  1039 VAR ARG1LO, ARG1HI: INTEGER;                                           *     *     S
  1040 BEGIN
  1041
  1042   READ_IFL (OP);     READ1ARG; "DISCARD JUMP TO INITIAL BLOCK"         367   558   455
  1043
  1044   DONE:=FALSE; "KEEP READING & TRANSLATING CODE UNTIL END-OF-MEDIUM."  278   337
  1045   REPEAT
  1046     READ_IFL(OP);                                                      367   558
  1047     CASE OP OF                                                         558
  1048
  1049 PUSHCONST1"(VALUE)":                                                   120
  1050   BEGIN
  1051   READ1ARG;  PICK_PUSHCONST(ARG1);  PUSHWORD;                          455   653   558   608
  1052   END;
  1053
  1054 PUSHVAR1"(TYPE, MODE, DISPL)":                                         120
  1055   BEGIN READ3ARG;                                                      461
  1056     IF ARG1 = WORDTYPE                                                 558   135
  1057       THEN PUSHVALUE(ARG2, ARG3)                                       711   558   558
  1058       ELSE BEGIN
  1059              PUSHADDRESS(ARG2, ARG3);                                  736   558   558
  1060              PUSHINDIRECT(ARG1);                                       763   558
  1061            END;
  1062   END;
  1063
  1064 PUSHIND1"(TYPE)":                                                      120
  1065   BEGIN READ1ARG; PUSHINDIRECT(ARG1) END;                              455   763   558
  1066
  1067 PUSHADDR1"(MODE, DISPL)":                                              120
  1068   BEGIN READ2ARG; PUSHADDRESS(ARG1, ARG2) END;                         458   736   558   558
  1069
  1070 FIELD1"(DISPL)":                                                       121
  1071   BEGIN READ1ARG;                                                      455
  1072     IF ARG1 <> 0                                                       558
  1073     THEN WRITE2(INC_2, (ARG1 DIV WORDLENGTH), B)                       511   215   558    28   269
  1074   END;
  1075
  1076 INDEX1"(MIN, MAX, LENGTH, TYPE)":                                      121
  1077   BEGIN   READ4ARG;                                                    464
  1078     PICK_PUSHCONST(ARG1);  PUSHWORD;                                   653   558   608
  1079     PICK_PUSHCONST(ARG2);  PUSHWORD;                                   653   558   608
```

```
    WRITE1(CHK_2);          POP(TWOWORDS);                               504  208  646  247
    PICK_PUSHCONST(ARG1);   PUSHWORD;                                    653  558  608
    WRITE1(SBI_2);          POPWORD;                                     504  199  615
    IF ARG4 <> BYTETYPE                                                  558  135
    THEN BEGIN
       WRITE2(IXA_2, ARG3 DIV WORDLENGTH, B);    POPWORD                 511  211  558   28  269  615
       END;
  END;

POINTER1:                                                                121
  BEGIN
  IF CHECK                                                               276
  THEN BEGIN
       WRITE1(DUP1_2);      PUSHWORD;                                    504  214  608
       WRITE1(SLDC00_2);    PUSHWORD;                                    504  172  608
       WRITE2(EFJ_2, CALLERRORDISPL, SB);    POP(TWOWORDS);              511  210  257  269  646  247
       "GENERATE CALL TO POINTER ERROR ROUTINE.  $$$"
       END
  END;

VARIANT1"(TAGSET, DISPL)":                                               121
"CHANGE PRIOR PASSES SO THAT TOS TAG IS <VALUE> OF
C BIT WHICH IS SET FOR TAG.    $$$"
  BEGIN    READ2ARG;                                                     458
  IF (VARNTCHECK)                                                        278
  THEN BEGIN
       WRITE1(DUP1_2);           PUSHWORD;                               504  214  608
       WRITE2(IND_2, ARG2 DIV WORDLENGTH, B);    "POPWORD; PUSHWORD;"    511  215  558   28  269
       WRITE2(LDCI_2, ARG1, W); PUSHWORD;                                511  191  558  269  608
       WRITE1(LAND_2);           "POPWORD;"                              504  199
       WRITE1(SLDC00_2);         "PUSHWORD;"                             504  172
       WRITE2(EFJ_2, CALLERRORDISPL, SB);  POP(TWOWORDS);                511  210  257  269  646  247
       "GENERATE CALL TO VARIANT ERROR ROUTINE.   $$$"
       END
  END;

RANGE1"(MIN, MAX)":                                                      122
  BEGIN READ2ARG;                                                        458
  IF CHECK                                                               276
  THEN BEGIN
       PICK_PUSHCONST(ARG1);   PUSHWORD;                                 653  558  608
       PICK_PUSHCONST(ARG2);   PUSHWORD;                                 653  558  608
       WRITE1(CHK_2);          POP(TWOWORDS);                            504  208  646  247
       END
  END;

ASSIGN1"(TYPE)":                                                         122
  BEGIN READ1ARG;                                                        455
    CASE ARG1 OF                                                         558
      BYTETYPE:                                                          135
        BEGIN WRITE1(STB_2); POPWORD END;                                504  208  615
      WORDTYPE:                                                          135
        BEGIN WRITE1(STO_2); POPWORD END;                                504  207  615
      REALTYPE:                                                          135
        BEGIN
        WRITE2(STM_2, REALLENGTH DIV WORDLENGTH, UB); POPREAL            511  194   29   28  269  626
        END;
      SETTYPE:                                                           135
        BEGIN
        WRITE2(FJP_2, 0, SB); POPWORD; "POP SET LENGTH VALUE"            511  211  269  615
  1140         WRITE2(STM_2, SETLENGTH DIV WORDLENGTH, UB); POPSET;          511   194    30    28   269   636
  1141         END
  1142      END;
  1143      POPWORD;                                                          615
  1144    END;
  1145
  1146 "ASSIGNTAG1 (LENGTH) :    ASSIGNTAG1 NEVER PRODUCED BY PASS 5.
C 1147    BEGIN  READ1ARG;
C 1148      IF ARG1 = 0 THEN WRITE1(COPYWORD2)
C 1149      ELSE WRITE2(COPYTAG2, ARG1 DIV WORDLENGTH);
C 1150      POPWORD; POPWORD;
C 1151    END; "
  1152
  1153 COPY1"(LENGTH)":                                                      122
  1154    BEGIN READ1ARG;                                                    455
  1155      WRITE2(MOV_2, ARG1 DIV WORDLENGTH, B);    POP(TWOWORDS);         511   207   558    28   269   646   247
  1156    END;
  1157
  1158 NEW1"(LENGTH, INITIALIZE)":   "$$$"                                   123
  1159    BEGIN READ2ARG;                                                    458
  1160      "IF (ARG2 = 1) & CHECK
C 1161        THEN WRITE3(NEWINIT2, BLOCK, ARG1)
C 1162        ELSE WRITE3(NEW2, BLOCK, ARG1);"
  1163      POPWORD;                                                         615
  1164    END;
  1165
  1166 NOT1:                                                                 123
  1167    WRITE1(LNOT_2);                                                    504   215
  1168
  1169 AND1"(TYPE)":                                                         123
  1170    BEGIN READ1ARG;                                                    455
  1171      IF ARG1 = WORDTYPE                                               558   135
  1172      THEN BEGIN WRITE1(LAND_2); POPWORD END                           504   199   615
  1173      ELSE BEGIN WRITE1(INT_2); POPWORD; POPSET END;                   504   213   615   636
  1174    END;
  1175
  1176 OR1"(TYPE)":                                                          123
  1177    BEGIN READ1ARG;                                                    455
  1178      IF ARG1 = WORDTYPE                                               558   135
  1179      THEN BEGIN WRITE1(LOR_2); POPWORD END                            504   199   615
  1180      ELSE BEGIN WRITE1(UNI_2); POPWORD; POPSET END;                   504   212   615   636
  1181    END;
  1182
  1183 NEG1"(TYPE)":                                                         124
  1184    BEGIN READ1ARG;                                                    455
  1185      IF ARG1 = WORDTYPE                                               558   135
  1186      THEN WRITE1(NGI_2)                                               504   214
  1187      ELSE WRITE1(NGR_2);                                              504   215
  1188    END;
  1189
  1190 ADD1"(TYPE)":                                                         124
  1191    BEGIN READ1ARG;                                                    455
  1192      IF ARG1 = WORDTYPE                                               558   135
  1193      THEN BEGIN WRITE1(ADI_2); POPWORD END                            504   199   615
  1194      ELSE BEGIN WRITE1(ADR_2); POPREAL END;                           504   206   626
  1195    END;
  1196
  1197 SUB1"(TYPE)":                                                         124
  1198    BEGIN READ1ARG;                                                    455
  1199      CASE ARG1 OF                                                     558
```

```
        WORDTYPE: BEGIN WRITE1(SBI_2); POPWORD END;                135  504  199  615
        REALTYPE: BEGIN WRITE1(SBR_2); POPREAL END;                135  504  206  626
        SETTYPE:  BEGIN WRITE1(DIF_2); POPWORD; POPSET END         135  504  213  615  636
    END;
  END;

MUL1"(TYPE)":                                                      124
  BEGIN READ1ARG;                                                  455
    IF ARG1 = WORDTYPE                                             558  135
    THEN BEGIN WRITE1(MPI_2); POPWORD END                          504  194  615
    ELSE BEGIN WRITE1(MPR_2); POPREAL END;                         504  206  626
  END;

DIV1"(TYPE)":                                                      125
  BEGIN READ1ARG;                                                  455
    IF ARG1 = WORDTYPE                                             558  135
    THEN BEGIN WRITE1(DVI_2); POPWORD END                          504  194  615
    ELSE BEGIN WRITE1(DVR_2); POPREAL END;                         504  206  626
  END;

MOD1"(TYPE)":                                                      125
  BEGIN READ1ARG; WRITE1(MODI_2); POPWORD END;                     455  504  194  615

"(NOT USED)"

"(NOT USED)"

FUNCTION1"(STANDARDFUNC, TYPE)":                                   126
  BEGIN READ2ARG;                                                  458
    IF (ARG1 >= MIN_FUNC) AND (ARG1 <= MAX_FUNC) THEN              558  162  558  162
    CASE ARG1 OF                                                   558
      TRUNC1:                                                      160
        BEGIN WRITE1(TNC_2); POPREAL; PUSHWORD END;                504  205  626  608
      ABS1:                                                        160
        IF ARG2 = WORDTYPE                                         558  135
        THEN WRITE1(ABI_2)                                         504  214
        ELSE WRITE1(ABR_2);                                        504  214
      SUCC1:                                                       160
        BEGIN
        WRITE1(SLDCO1_2);   PUSHWORD;                              504  172  608
        WRITE1(ADI_2);      POPWORD                                504  199  615
        END;
      PRED1:                                                       160
        BEGIN
        WRITE1(SLDCO1_2);    PUSHWORD;                             504  172  608
        WRITE1(SBI_2);       POPWORD;                              504  199  615
        END;
      CONV1:                                                       161
        BEGIN WRITE1(FLT_2); POPWORD; PUSHREAL END;                504  209  615  619
      EMPTY1:                                                      161
        BEGIN
        WRITE1(SLDCO0_2);   PUSHWORD;                              504  172  608
        WRITE1(EQUI_2);     POPWORD;                               504  202  615
        END;
      ATTRIBUTE1,   "$$$"                                          161
        "WRITE1(ATTRIBUTE2);"
      REALTIME1:    "$$$"                                          161
        "BEGIN WRITE1(REALTIME2); PUSHWORD END"
    END;
  END;
  1260
  1261 BUILDSET1:                                                          126
  1262   BEGIN
  1263   WRITE1(SLDC00_2);   PUSHWORD; "PUSH LOWER LEGAL SET BOUND."       504   172   608
  1264   "PUSH UPPER LEGAL SET BOUND-- EXPECTED TO BE 127."
  1265   PICK_PUSHCONST((SETLENGTH * 8 "BITS IN BYTE") - 1); PUSHWORD;     653    30   608
  1266   WRITE1(CHK_2);        POP(TWOWORDS);                              504   208   646   247
  1267   WRITE1(DUP1_2);       PUSHWORD;                                   504   214   608
  1268   WRITE1(SRS_2);        PUSHSET;  PUSHWORD;                         504   205   629   608
  1269      "THE SUBRANGE SET (OF 1 ITEM) JUST BUILT MAY BE LESS THAN
C 1270       THE FIXED SET SIZE OF 16 BYTES. CALL TO 'PUSHSET' WILL
C 1271       INCREASE CALCULATED SIZE OF RUNTIME STACK BY 16, REGARDLESS
C 1272       OF ACTUAL SET SIZE."
  1273   WRITE1(UNI_2);        POPWORD;  POPSET                            504   212   615   636
  1274   END;
  1275
  1276 COMPARE1"(COMPARISON, TYPE)":                                       126
  1277   BEGIN READ2ARG;                                                   458
  1278   CASE ARG2 OF                                                      558
  1279        WORDTYPE: BEGIN                                              135
  1280           CASE ARG1 OF                                              558
  1281               LESS:        BEGIN WRITE1(GEQI_2); WRITE1(LNOT_2); END;  155   504   202   504   215
  1282               EQUAL:             WRITE1(EQUI_2);                   155   504   202
  1283               GREATER:     BEGIN WRITE1(LEQI_2); WRITE1(LNOT_2); END;  155   504   202   504   215
  1284               NOTLESS:           WRITE1(GEQI_2);                   155   504   202
  1285               NOTEQUAL:          WRITE1(NEQI_2);                   156   504   202
  1286               NOTGREATER:        WRITE1(LEQI_2)                    156   504   202
  1287               END;
  1288           POPWORD;                                                  615
  1289           END;
  1290
  1291        REALTYPE: BEGIN                                              135
  1292           CASE ARG1 OF                                              558
  1293               LESS:     BEGIN WRITE1(GEQREAL_2); WRITE1(LNOT_2); END;  155   504   209   504   215
  1294               EQUAL:          WRITE1(EQUREAL_2);                   155   504   209
  1295               GREATER:  BEGIN WRITE1(LEQREAL_2); WRITE1(LNOT_2); END;  155   504   209   504   215
  1296               NOTLESS:        WRITE1(GEQREAL_2);                   155   504   209
  1297               NOTEQUAL: BEGIN WRITE1(EQUREAL_2); WRITE1(LNOT_2); END;  156   504   209   504   215
  1298               NOTGREATER:   . WRITE1(LEQREAL_2)                    156   504   209
  1299               END;
  1300           POPREAL; POPREAL; PUSHWORD;                               626   626   608
  1301           END;
  1302
  1303        SETTYPE: BEGIN                                               135
  1304           CASE ARG1 OF                                              558
  1305               EQUAL:      WRITE1(EQUPWR_2);                         155   504   203
  1306               NOTLESS:    WRITE1(GEQPWR_2);                         155   504   204
  1307               NOTEQUAL:   BEGIN WRITE1(EQUPWR_2); WRITE1(LNOT_2); END;  156   504   203   504   215
  1308               NOTGREATER: WRITE1(LEQPWR_2);                         156   504   203
  1309               INSET:      WRITE1(INN_2)                             156   504   212
  1310               END;
  1311           POPWORD;  POPSET;                                         615   636
  1312           IF ARG1 <> INSET THEN                                     558   156
  1313           BEGIN   "POPWORD;"    POPSET;   "PUSHWORD"   END;         636
  1314           END;
  1315        END;
  1316   END;
  1317
  1318 COMPSTRUC1"(COMPARISON, LENGTH)":                                   126
  1319 BEGIN READ2ARG;                                                     458
  CASE ARG1 OF                                                            558
    LESS:        BEGIN WRITE2(GEQBYT_2, ARG2, B); WRITE1(LNOT_2); END;   155  511  204  558  269  504  215
    EQUAL:             WRITE2(EQUBYT_2, ARG2, B);                        155  511  204  558  269
    GREATER:     BEGIN WRITE2(LEQBYT_2, ARG2, B); WRITE1(LNOT_2); END;   155  511  204  558  269  504  215
    NOTLESS:           WRITE2(GEQBYT_2, ARG2, B);                        155  511  204  558  269
    NOTEQUAL:    BEGIN WRITE2(EQUBYT_2, ARG2, B); WRITE1(LNOT_2); END;   156  511  204  558  269  504  215
    NOTGREATER:        WRITE2(LEQBYT_2, ARG2, B)                         156  511  204  558  269
  END;
  POPWORD;                                                                615
END;

FUNCVALUE1"(MODE, TYPE)":                                                127
  BEGIN READ2ARG;                                                         458
     CASE ARG1 OF                                                         558

     MODE1, MODE3:   "PROCEDURE MODE, PROCESS ENTRY MODE"                 140  142
        IF ARG2 = WORDTYPE                                                558  135
        THEN BEGIN WRITE1(SLDCOO_2); PUSHWORD END                         504  172  608
        ELSE BEGIN WRITE1(SLDCOO_2); WRITE1(SLDCOO_2); PUSHREAL END;      504  172  504  172  619

     MODE4, MODE5, MODE12: "ENTRY MODES: CLASS, MONITOR, MGR"             143  144  151
        IF ARG2 = WORDTYPE                                                558  135
        THEN BEGIN
             WRITE1(SLDCOO_2);   WRITE1(SWAP_2);   PUSHWORD               504  172  504  205  608
             END
        ELSE BEGIN
             WRITE1(SLDCOO_2);   WRITE1(SWAP_2);                          504  172  504  205
             WRITE1(SLDCOO_2);   WRITE1(SWAP_2);   PUSHREAL               504  172  504  205  619
             END
     END
  END;

DEFLABEL1"(LABEL)":                                                      127
  BEGIN READ1ARG;                                                         455
    ENTER(JUMPTABLE, ARG1, LOCATION);                                     802  282  558  303
    IF NUMBER                                                             480
    THEN BEGIN
         PICK_PUSHCONST(LINE);    PUSHWORD;                               653  306  608
         WRITE2(STL_2, FIXLCL_LINE, B);     POPWORD;                      511  200  238  269  615
         END;
  END;

JUMP1"(LABEL)":                                                          127
  BEGIN READ1ARG;                                                         455
    WRITE2(UJPL_2, ARG1, W); WRITELOCATION;                               511  193  558  269  585
  END;

FALSEJUMP1"(LABEL)":                                                     127
  BEGIN READ1ARG;                                                         455
    WRITE2(FJPL_2, ARG1, W); WRITELOCATION;                               511  211  558  269  585
    POPWORD;                                                              615
  END;

CASEJUMP1"(MIN, MAX, LABELS)":                                           128
     "GENERATE RANGE CHECK ON CASE SELECTOR VALUE (TOS)
C     AND PUT CASEJUMP OPERATOR INTO CODE STREAM.
C     OPERAND TO XJP IS OFFSET (IN WORDS, FROM START
C     OF SEGMENT) INTO CONSTANT POOL WHERE CASE TABLE
C     WILL BE FOUND AT RUN TIME. CASE TABLE CONSISTS
C     OF MIN. CASE INDEX (1 WORD), MAX. CASE INDEX
```

```
C 1380       (1 WORD), AND JUMP DISPLACEMENTS (1 WORD EACH)."
  1381    BEGIN READ2ARG;                                                    458
  1382      IF ARG2-ARG1 > 0                                                 558   558
  1383      THEN BEGIN
  1384           PICK_PUSHCONST (ARG1);    PUSHWORD;   "PUSH MIN"            653   558   608
  1385           PICK_PUSHCONST (ARG2);    PUSHWORD;   "PUSH MAX"            653   558   608
  1386           WRITE1 (CHK_2);           POP(TWOWORDS);   "RANGECHECK"     504   208   646   247
  1387           " 'LINK@.CONSTANTS' IS AMOUNT OF SPACE NEEDED IN CONSTANT
C 1388           POOL FOR LONG CONSTANTS ONLY (DOES NOT INCLUDE SPACE
C 1389           NEEDED FOR FOR CASE TABLES)."
  1390           " 'XJPOFFSETPTR' POINTS TO WORD LAST FILLED."
  1391           WRITE2 (XJP_2, ((ONEWORD + LINK@.CONSTANTS) DIV WORDLENGTH) +  511   211   246   349   288    28
  1392                          XJPOFFSETPTR, B);      POPWORD;              289   269   615
  1393
  1394           "BUILD CASE TABLE"
  1395           XJPOFFSETPTR := SUCC(XJPOFFSETPTR);                         289   373   289
  1396           ENTER (XJPTABLE, XJPOFFSETPTR, ARG1); "MIN"                 802   938   289   558
  1397           XJPOFFSETPTR := SUCC(XJPOFFSETPTR);                         289   373   289
  1398           ENTER (XJPTABLE, XJPOFFSETPTR, ARG2); "MAX"                 802   938   289   558
  1399           "REMOVE CASE LABELS FROM CODE STREAM AND PUT
C 1400            CORRESPONDING OFFSETS IN CASE TABLE."
  1401           ARG3 := ARG1;                                               558   558
  1402           REPEAT
  1403              READ1ARG;      "READ A CASE LABEL"                      455
  1404              ARG4 := ENTR(JUMPTABLE, ARG1) - LOCATION;                558   814   282   558   303
  1405              XJPOFFSETPTR := SUCC(XJPOFFSETPTR);                      289   373   289
  1406              ENTER (XJPTABLE, XJPOFFSETPTR, ARG4);                    802   938   289   558
  1407              ARG3 := ARG3 + 1;                                        558   558
  1408           UNTIL ARG3 > ARG2;                                          558   558
  1409           END;
  1410    END;
  1411
  1412 "INITVAR1 (LENGTH) :       NEVER PRODUCED BY PASS 5.
C 1413    BEGIN READ1ARG;
C 1414      IF CHECK THEN WRITE2(INITVAR2, ARG1 DIV WORDLENGTH);
C 1415    END;  "
  1416
  1417 CALL1"(MODE, LABEL, PARAMLENGTH)":     "$$$"                          128
  1418    BEGIN READ3ARG;                                                    461
  1419      IF ARG1 = MODE3      "IF PROCESS ENTRY"                          558   142
  1420      THEN BEGIN
  1421           PICK_PUSHCONST(ARG2);     PUSHWORD; "PREFIX ROUTINE NO."    653   558   608
  1422           WRITE3(CXL_2, INTFSEG, STARTPROC, UB, UB);                  524   195   254   255   269   269
  1423           ARG1:= WORDLENGTH                                           558    28
  1424           END
  1425      ELSE BEGIN
  1426           WRITE2(CPL_2, ARG2, UB);    PUSH(MSCWSIZE);                 511   195   558   269   639   241
  1427           IF CONCURRENT                                               243
  1428           THEN ARG1:= ENTR(STACKTABLE, ARG2)                          558   814   282   558
  1429           ELSE ARG1:= WORDLENGTH;                                     558    28
  1430           END;
  1431      PUSH(ARG1); POP(ARG1 + ARG3);                                    639   558   646   558   558
  1432    END;
  1433
  1434 ENTER1"(MODE, LABEL, PARAMLENGTH, VARLENGTH, TEMPLENGTH)":            128
  1435    BEGIN READ5ARG;                                                    470
  1436      BLOCK := ARG2;                                                   302   558
  1437      PARAMLENGTH := ARG3;                                             298   558
  1438      VARLENGTH := ARG4;                                               297   558
  1439      STACKLENGTH := ARG5;                                             291   308
  1440     AFTERBEGIN := TRUE;                                                    277   242
  1441     TEMP := 0;                                                             300
  1442     MAXTEMP := 0;                                                          301
  1443     LOCATION := LOCATION + TWOWORDS; "EXIT-IC & DATASIZE"                  303   303   247
  1444     CASE ARG1 OF                                                           558
  1445       MODE1:   "PROCEDURE"                                                 140
  1446          BEGIN
  1447          ENTER (DATASIZETABLE, BLOCK, (VARLENGTH+TWOWORDS)                 802   283   302   297   247
  1448                                               DIV WORDLENGTH);              28
  1449          POPLENGTH := PARAMLENGTH + VARLENGTH + TWOWORDS;                  299   298   297   247
  1450          PICK_PUSHCONST(LINE);                PUSHWORD;                    653   306   608
  1451          WRITE2(STL_2, FIXLCL_LINE, B);       POPWORD;                     511   200   238   269   615
  1452          END;
  1453       MODE2:   "PROGRAM"                                                   141
  1454          BEGIN
  1455          "BUMP 'JOB' "            "$$$"
  1456          GEN_SAVELINE_SAVEGBASE;                                           895
  1457          "MAKE GLOBAL BASE SAME AS PRESENT LOCAL BASE"
  1458          PICK_PUSHCONST(GBASE_REG);         PUSHWORD;                      653   234   608
  1459          PICK_PUSHCONST(LBASE_REG);         PUSHWORD;                      653   233   608
  1460          WRITE1 (LPR_2);                                                   504   198
  1461          WRITE1 (SPR_2);                    POP (TWOWORDS)                 504   210   646   247
  1462          END;
  1463       MODE3:  "PROCESS ENTRY"                                              142
  1464          BEGIN
  1465          ENTER (DATASIZETABLE, BLOCK, (VARLENGTH+TWOWORDS)                 802   283   302   297   247
  1466                                     DIV WORDLENGTH);                        28
  1467          POPLENGTH := VARLENGTH + TWOWORDS;                                299   297   247
  1468          GEN_SAVELINE_SAVEGBASE;                                           895
  1469          "RE-ESTABLISH OLD GLOBAL BASE"
  1470          PICK_PUSHCONST (GBASE_REG);          PUSHWORD;                    653   234   608
  1471          WRITE1 (SLDOO2_2);                PUSHWORD;                       504   184   608
  1472          WRITE1 (SPR_2);                      POP (TWOWORDS);              504   210   646   247
  1473          "SET 'JOB'  :=  0  "             "$$$"
  1474          END;
  1475       MODE4,  "CLASS ENTRY"                                                143
  1476       MODE7:  "CLASS"                                                      146
  1477               "  'POPLENGTH' MUST INCLUDE THE LOCATION CONTAINING
C 1478                THE ADDRESS OF THE CLASS VARIABLE.
C 1479                THE NEW GLOBAL BASE REG. MUST POINT A FEW WORDS
C 1480                BELOW FIRST GLOBAL VARIABLE. THE 'FEW WORDS'
C 1481                ARE THE MSCW. HARDWARE TAKES THE SIZE OF THE
C 1482                MSCW INTO ACCOUNT WHEN ACCESSING VARIABLES."
  1483          BEGIN
  1484          IF ARG1 = MODE4                                                   558   143
  1485          THEN BEGIN  "CLASS ENTRY"
  1486               ENTER (DATASIZETABLE, BLOCK, (VARLENGTH+TWOWORDS)            802   283   302   297   247
  1487                                          DIV WORDLENGTH);                   28
  1488               POPLENGTH := PARAMLENGTH + ONEWORD +                         299   298   246
  1489                                          VARLENGTH + TWOWORDS;             297   247
  1490               END
  1491          ELSE BEGIN  "CLASS INITIAL STATEMENT"
  1492               ENTER (DATASIZETABLE, BLOCK, TWOWORDS DIV WORDLENGTH);       802   283   302   247    28
  1493               POPLENGTH := ONEWORD + TWOWORDS;                             299   246   247
  1494               END;
  1495          GEN_SAVELINE_SAVEGBASE;                                           895
  1496          "SET NEW GLOBAL BASE"
  1497          PICK_PUSHCONST (GBASE_REG);          PUSHWORD;                    653   234   608
  1498          WRITE2 (LDL_2, POPLENGTH DIV WORDLENGTH, B);  PUSHWORD;           511   192   299    28   269   608
  1499          PICK_PUSHCONST (MSCWSIZE);           PUSHWORD;                    653   241   608
  1500          WRITE1 (SBI_2);                   POPWORD;                   504   199   615
  1501          WRITE1 (SPR_2);                   POP (TWOWORDS)             504   210   646   247
  1502          END;
  1503        MODE5,  "MONITOR ENTRY"                                        144
  1504        MODE12: "MANAGER ENTRY"                                        151
  1505          BEGIN
  1506          ENTER (DATASIZETABLE, BLOCK, (VARLENGTH + TWOWORDS)          802   283   302   297   247
  1507                                         DIV WORDLENGTH);               28
  1508          POPLENGTH := PARAMLENGTH + ONEWORD + VARLENGTH + TWOWORDS;   299   298   246   297   247
  1509          GEN_SAVELINE_SAVEGBASE;                                      895
  1510          "NEW GLOBAL BASE"
  1511          PICK_PUSHCONST (GBASE_REG);  PUSHWORD;                       653   234   608
  1512          WRITE2 (LDL_2, POPLENGTH DIV WORDLENGTH, B);  PUSHWORD;      511   192   299    28   269   608
  1513          PICK_PUSHCONST (MSCWSIZE);   PUSHWORD;                       653   241   608
  1514          WRITE1 (SBI_2);              POPWORD;                        504   199   615
  1515          WRITE1 (SPR_2);              POP (TWOWORDS);                 504   210   646   247
  1516          "KERNEL CALL TO ENTER GATE"         "$$$"
  1517          END;
  1518        MODE6:  "PROCESS"                                              145
  1519          BEGIN
  1520          ENTER (DATASIZETABLE, BLOCK, TWOWORDS DIV WORDLENGTH);       802   283   302   247    28
  1521          POPLENGTH := TWOWORDS;                                       299   247
  1522          PICK_PUSHCONST (LINE);            PUSHWORD;                  653   306   608
  1523          WRITE2 (STL_2, FIXLCL_LINE, B);   POPWORD                    511   200   238   269   615
  1524          END;
  1525        MODE8,  "MONITOR"                                              147
  1526        MODE11: "MANAGER"                "MGR""                        150
  1527          BEGIN
  1528          ENTER (DATASIZETABLE, BLOCK, TWOWORDS DIV WORDLENGTH);       802   283   302   247    28
  1529          POPLENGTH := ONEWORD + TWOWORDS;                             299   246   247
  1530          GEN_SAVELINE_SAVEGBASE;                                      895
  1531          "SET NEW GLOBAL BASE"
  1532          PICK_PUSHCONST (GBASE_REG);       PUSHWORD;                  653   234   608
  1533          WRITE2 (LDL_2, POPLENGTH DIV WORDLENGTH, B);   PUSHWORD;     511   192   299    28   269   608
  1534          PICK_PUSHCONST (MSCWSIZE);        PUSHWORD;                  653   241   608
  1535          WRITE1 (SBI_2);                   POPWORD;                   504   199   615
  1536          WRITE1 (SPR_2);                   POP (TWOWORDS);            504   210   646   247
  1537          "KERNEL CALL TO INITIALIZE GATE"        "$$$"
  1538          END;
  1539        MODE10:  "UNDEFINED"                                           149
  1540      END;
  1541    END;
  1542
  1543 RETURN1"(MODE)":                                                      129
  1544    BEGIN READ1ARG;                                                    455
  1545    "CURRENT LOCATION IS THIS ROUTINE'S 'EXIT IC' "
  1546    ENTER (EXITICTABLE, BLOCK, LOCATION);                              802   283   302   303
  1547      CASE ARG1 OF                                                     558
  1548        MODE1:  BEGIN     "PROCEDURE MODE"                             140
  1549                WRITE2(RPU_2, POPLENGTH DIV WORDLENGTH, B);            511   196   299    28   269
  1550                POP (POPLENGTH + MSCWSIZE)                             646   299   241
  1551                END;
  1552        MODE2: BEGIN   "PROGRAM MODE"      "$$$"                       141
  1553               ""PUSH 'CONTINUE'"
C 1554               WRITE1(SLDC00_2);
C 1555               WRITE2(NFJ_2, "DISPLACEMENT TO CONO:", SB);
C 1556               WRITE1(SLDC00_2);
C 1557               "STORE IN 'RESULT'"
C 1558               "CONO:"
C 1559               "PUSH POINTER TO 'LINE'"
```

```
C 1560              WRITE1(SLDL01_2);
C 1561              WRITE1(STO_2);
C 1562              "PUSH 'JOB'"
C 1563              WRITE1(SLDC00_2);
C 1564              WRITE2(EFJ_2, "DISPLACEMENT TO SEQL:", SB);
C 1565              "KERNEL CALL--SYSTEM ERROR"
C 1566              WRITE2(UJP_2, "DISPLACEMENT TO DONE:", SB);
C 1567              "SEQL:"
C 1568              WRITE1(SLDC06_2);
C 1569              WRITE1(SLD002_2);
C 1570              WRITE1(SPR);
C 1571              "CLEAR 'JOB'"
C 1572              "DONE:""
  1573              WRITE2(RPU_2, POPLENGTH DIV WORDLENGTH, B)               511   196   299    28   269
  1574              END;
  1575       MODE3: BEGIN    "PROCESS ENTRY MODE"                            142
  1576          GEN_EXIT;     " 'JOB' := SUCC ('JOB');"    "$$$"            909
  1577          END;
  1578       MODE4:  GEN_EXIT;    "CLASS ENTRY MODE"                         143   909
  1579       MODE5,               "MONITOR ENTRY"                            144
  1580       MODE12: BEGIN        "MANAGER ENTRY MODE"                       151
  1581          "KERNEL CALL TO LEAVE GATE"     "$$$"
  1582          GEN_EXIT                                                     909
  1583          END;
  1584       MODE6:           "PROCESS MODE"                                 145
  1585          WRITE2(RPU_2, POPLENGTH DIV WORDLENGTH, B);                  511   196   299    28   269
  1586       MODE7:  GEN_EXIT;    "CLASS MODE"                               146   909
  1587       MODE8,               "MONITOR MODE"                             147
  1588       MODE11:  BEGIN       "MANAGER MODE"                             150
  1589          "KERNEL CALL TO LEAVE GATE"     "$$$"
  1590          GEN_EXIT                                                     909
  1591          END;
  1592       MODE10:     "UNDEFINED MODE"                                    149
  1593     END;
  1594     IF (STACKLENGTH + MAXTEMP + VARLENGTH) < STACK_LIMIT              291   301   297   248
  1595     THEN STACKLENGTH := STACKLENGTH + MAXTEMP + VARLENGTH             291   291   301   297
  1596     ELSE ERROR (THIS_PASS, STACK_ERROR);                              591   249   256
  1597     ENTER (STACKTABLE, BLOCK, STACKLENGTH);                           802   282   302   291
  1598     AFTERBEGIN := FALSE;                                              277   337
  1599     IF (LOCATION MOD 2) = 1   "IF NOT ON WORD BOUNDARY"               303
  1600     THEN WRITE1 (NOP_2)       "THEN PAD TO A WORD BOUNDARY"           504   198
  1601   END;
  1602
  1603 POP1"(LENGTH)":                                                     129
  1604   BEGIN READ1ARG;                                                   455
  1605   WHILE ARG1 >= TWOWORDS DO                                         558   247
  1606      BEGIN
  1607      WRITE2(NFJ_2, 0, SB);   POP(TWOWORDS);                         511   210   269   646   247
  1608      ARG1 := ARG1 - TWOWORDS;                                       558   558   247
  1609      END;
  1610   IF ARG1 = ONEWORD                                                 558   246
  1611   THEN BEGIN WRITE2(FJP_2, 0, SB);   POPWORD;   END;                511   211   269   615
  1612   END;
  1613
  1614 NEWLINE1"(NUMBER)":                                                 129
  1615   "LOCAL WORD 1 ALWAYS RESERVED FOR LINE #."
  1616   BEGIN READ1ARG;                                                   455
  1617     LINE := ARG1;                                                   306   558
  1618     AFTERERROR := FALSE;                                            278   337
  1619     IF (NUMBER AND AFTERBEGIN)                                      480   277
```

```
    THEN BEGIN
         WRITEARG (NEWLIN_2, NOTHE); "CRUTCH FOR MNEMONICS PROGRAM"      578  217  269
         PICK_PUSHCONST(ARG1);   PUSHWORD;                              653  558  608
         WRITE2(STL_2, FIXLCL_LINE, B);    POPWORD                       511  200  238  269  615
         END;
  END;

ERROR1:                                                                  129
  GENERATE:= FALSE;                                                      277  337

CONSTANT1 "(LENGTH, VALUE)":                                             130
 "REMOVE LONG CONSTANT FROM CODE STREAM & PUT IN CONSTANT POOL."
  BEGIN READ1ARG;                                                        455
        ARG2 := ARG1;                                                    558  558
    FOR ARG3 := 1 TO                                                     558
        (ARG1+SOURCE_WORD_LENGTH-1) DIV SOURCE_WORD_LENGTH DO            558   36   36
      BEGIN
      CONSTANTS:= CONSTANTS + 1 "WORD";                                  288  288
      READ1ARG;                                                          455

      "$IF SRC_16"
      ENTER(CONSTTABLE, CONSTANTS, ARG1);                                802  282  288  558
      "$END"

       "$IF SRC_32"
       SPLIT_CONST(ARG1, ARG1HI, ARG1LO);                                888  558 1039 1039
       ENTER(CONSTTABLE, CONSTANTS, ARG1HI);                             802  282  288 1039
       IF (((ARG3-1)*SOURCE_WORD_LENGTH) + WORDLENGTH) < ARG2            558   36   28  558
       THEN BEGIN
            CONSTANTS := CONSTANTS + 1;                                  288  288
            ENTER(CONSTTABLE, CONSTANTS, ARG1LO)                         802  282  288 1039
            END
       "$END"

      END;
  END;

MESSAGE1"(PASS, ERROR)":                                                 130
  BEGIN READ2ARG;                                                        458
    ERROR(ARG1, ARG2)                                                    591  558  558
  END;

INCREMENT1:                                                              130
  BEGIN
  WRITE1(DUP1_2);         PUSHWORD;                                      504  214  608
  WRITE2(LDM_2, 1, UB);   PUSHWORD;                                      511  210  269  608
  WRITE1(SLDC01_2);       PUSHWORD;                                      504  172  608
  WRITE1(ADI_2);          POPWORD;                                       504  199  615
  WRITE1(STO_2);          POP(TWOWORDS);                                 504  207  646  247
  END;

DECREMENT1:                                                              130
  BEGIN
  WRITE1(DUP1_2);         PUSHWORD;                                      504  214  608
  WRITE2(LDM_2, 1, UB);   PUSHWORD;                                      511  210  269  608
  WRITE1(SLDC01_2);       PUSHWORD;                                      504  172  608
  WRITE1(SBI_2);          POPWORD;                                       504  199  615
  WRITE1(STO_2);          POP(TWOWORDS);                                 504  207  646  247
  END;

  1680 PROCEDURE1"(STANDARDPROCEDURE)":    "$$$"                      131
  1681   BEGIN READ1ARG;                                                  455
  1682     "IF (ARG1 >= MIN_PROC) AND (ARG1 <= MAX_PROC) THEN
C 1683     CASE ARG1 OF
C 1684       DELAY1:
C 1685         BEGIN WRITE1(DELAY2); POPWORD END;
C 1686       CONTINUE1:
C 1687         BEGIN WRITE1(CONTINUE2); POPWORD END;
C 1688       IO1:
C 1689         BEGIN WRITE1(IO2); POP(THREEWORDS) END;
C 1690       START1:
C 1691         WRITE1(START2);
C 1692       STOP1:
C 1693         BEGIN WRITE1(STOP2); POP(TWOWORDS) END;
C 1694       SETHEAP1:
C 1695         BEGIN WRITE1(SETHEAP2); POPWORD END;
C 1696       WAIT1:
C 1697         WRITE1(WAIT2)
C 1698     END; "
  1699   END;
  1700
  1701 INIT1"(MODE, LABEL, PARAMLENGTH, VARLENGTH)":                     131
  1702   BEGIN READ4ARG;                                                  464
  1703   IF ARG1 = MODE6     "IF PROCESS MODE"                            558   145
  1704   THEN BEGIN
  1705      "KERNEL CALL TO INIT PROCESS"    "$$$"
  1706      WRITE2 (FJP_2, 0, SB);     POPWORD                            511   211   269   615
  1707      END
  1708   ELSE BEGIN   "CLASS OR MONITOR"
  1709      WRITE2 (STM_2, ARG3 DIV WORDLENGTH, UB);                      511   194   558    28   269
  1710      POP ((ARG3 DIV WORDLENGTH) + 1);                              646   558    28
  1711      WRITE2 (CPL_2, ARG2, UB);                                     511   195   558   269
  1712      PUSH (MSCWSIZE);                                              639   241
  1713      END
  1714   END;
  1715
  1716 PUSHLABEL1"(LABEL)":  GEN_INTERFACE;                               131   924
  1717
  1718 CALLPROG1:                                                         131
  1719   BEGIN
  1720     "ADDRESS OF SEQ'L PROGRAM VARIABLE ON TOS"
  1721     "PUT IT IN SEQ'L PGM'S SIB"
  1722     WRITE1(SLDC01_2); PUSHWORD; "GET @ OF CURRENTLY RUNNING TIB"   504   172   608
  1723     WRITE1(NGI_2);                                                 504   214
  1724     WRITE1(LPR_2);                                                 504   198
  1725     WRITE2(IND_2, SIBS_OFFSET, B);  "GET ADDRESS OF SIB VECTOR"    511   215   252   269
  1726     WRITE2(INC_2, 5, B);    "SEGBASE FIELD IN SIB FOR SEGMENT      511   215   269
C 1727                             129 (SEQ'L PGM) IS 5 WORDS FROM
C 1728                             START OF SIB VECTOR."
  1729
  1730     "AFTER EXECUTION OF THE 'INC (5)', @ SEQ'L PROGRAM VARIABLE
C 1731      IS IN THE TOS-1 WORD, AND @ OF THE SEGBASE FIELD IN SIB
C 1732      NO. 129 IS IN TOS WORD."
  1733     WRITE1(SWAP_2);                                                504   205
  1734     WRITE1(STO_2);  POP(TWOWORDS);  "PUT @ PGM VAR IN SIB"         504   207   646   247
  1735     WRITE3(CXL_2, SEQL_SEG, STARTPROC, UB, UB); "CALL SEQ'L PGM"   524   195   253   255   269   269
  1736     READ_IFL(OP); READ1ARG  "DISPOSE OF 'POP' VIRTUAL OPERATOR     367   558   455
C 1737                             AND ITS OPERAND... NOT NEEDED FOR
C 1738                             MICROENGINE."
  1739   END;
```

```
1740
1741 EOM1"(VARLENGTH)":                                                  132
1742   BEGIN
1743     DONE:=TRUE;                                                     278   242
1744     READ1ARG; VARLENGTH:=ARG1;                                      455   297   558
1745     ENTER (DATASIZETABLE, STARTPROC,                                802   283   255
1746                 (VARLENGTH + TWOWORDS) DIV WORDLENGTH);             297   247    28
1747     WRITEARG(EOM_2, NOTME);                                         578   217   269
1748   END;
1749
1750
1751 DUPTOS1: BEGIN WRITE1(DUP1_2); PUSHWORD END                         132   504   214   608
1752
1753
1754
1755     END
1756   UNTIL DONE                                                        278
1757 END;
1758
1759 BEGIN
1760   INIT_PASS(LINK);                                                  340   349
1761   BEGINPASS;                                                        830
1762   SCAN;                                                            1038
1763   ENDPASS;                                                          864
1764   NEXT_PASS(LINK);                                                  349   349
1765 END.
```

```
CROSS REFERENCE    * IS DEF    = IS ASG

 -A-
  ABI_2         214* 1235
  ABR_2         214* 1236
  ABS           420
  ABS1          160* 1233
  ADD1          124* 1190
  ADI_2         199* 1193  1240  1667
  ADJ_2         207*
  ADR_2         206* 1194
  AFTERBEGIN    277*  856= 1440= 1598= 1619
  AFTERERROR    278*  593   595= 1618=
  ALLOCATE      789*  843   844   845   846   847   849
  AND1          123* 1169
  ARG           414*  417   443*  446   447   511*  515   517   520   578*  580   581   582   666*  675   679   680   681
                691   692   693   703   704   705   711*  717   720   723   724   725   729   736*  740   744   747   750
                751   752   756
  ARG1          308*  456   459   462   466   472   524*  530   533   537   541*  546   549   553   558*  564   568   573
               1051  1056  1060  1065  1068  1072  1073  1078  1081  1108  1120  1128  1155  1171  1178  1185  1192  1199
               1208  1215  1229  1229  1230  1280  1292  1304  1312  1320  1333  1354  1364  1369  1382  1384  1396  1401
               1404  1419  1423= 1428= 1429= 1431  1431  1444  1484  1547  1605  1608= 1608  1610  1617  1622  1633  1635
               1641  1645  1659  1703  1744
  ARG1HI       1039* 1645  1646
  ARG1LO       1039* 1645  1650
  ARG1TYP       525*  537   542*  553   559*  573
  ARG2          308*  459   462   466   472   524*  530   533   537   541*  547   550   554   558*  564   568   573  1057
               1059  1068  1079  1107  1121  1234  1278  1321  1322  1323  1324  1325  1326  1336  1341  1382  1385  1398
               1408  1421  1426  1428  1436  1633= 1647  1659  1711
  ARG2TYP       525*  537   542*  554   559*  573
  ARG3          308*  462   467   472   541*  547   550   554   558*  565   569   574  1057  1059  1085  1401= 1407= 1407
               1408  1431  1437  1634= 1647  1709  1710
  ARG3TYP       542*  554   559*  574
  ARG4          308*  467   473   558*  565   569   574  1083  1404= 1406  1438
  ARG4TYP       559*  574
  ARG5          308*  473  1439
  ARGLIST        91*  108
  ARGTAG         78*   82
  ARGTYP        511*  520   578*  582
  ARGTYPE        81*   91
  ASSIGN1       122* 1126
  ASSIGNTAG1    122*
  ATTRIBUTE1    161* 1254

 -B-
  B             269   487   717   720   724   725   729   740   744   747   751   752   756   902   906   915   969  1014
               1073  1085  1107  1155  1321  1322  1323  1324  1325  1326  1358  1392  1451  1498  1512  1523  1533  1549
               1573  1585  1623  1725  1726
  BEGINPASS     830* 1761
  BLOCK         102*  103*  302* 1436= 1447  1465  1486  1492  1506  1520  1528  1546  1597
  BLOCKS         72*  846   847   849   878
  BOOL           83   360=
  BOOLEAN        83   278   280   318
  BOOLTYPE       79    83   360
  BPT_2         198*
  BUILDSET1     126* 1261
  BYTETYPE      135*  766  1083  1129

 -C-
 C              98*   99*
 CALL1         128*  1417
 CALLERRORDIS  257*  1095  1111
 CALLPROG1     131*  1718
 CASEJUMP1     128*  1373
 CHAR           45     92    98    99   415
 CHECK         276*   837=  1091  1118
 CHECKOPTION    38*   837
 CHK_2         208*  1080  1122  1266  1386
 CHR           420
 CODE          482*   485
 CODEOPTION     39*   869
 CODE_ERROR    256*   494
 CODE_LIMIT    248*   484
 COMPARE1      126*  1276
 COMPSTRUC1    126*  1318
 CONCURRENT    243*  1427
 CONSEG        932*  1008  1027
 CONSTANT1     130*  1630
 CONSTANTS      72*   288*   844   851=  853   866=  866  1391  1637= 1637  1641  1646  1649= 1649  1650
 CONSTTABLE     55*   282*   844   880=  880  1641  1646  1650
 CONTENTS       51*   811=   823
 CONTINUE1     166*
 CONV1         161*  1247
 COPY1         122*  1153
 CPF_2         196*
 CPG_2         195*
 CPI_2         195*
 CPL_2         195*  1426  1711
 CXG_2         196*
 CXI_2         196*
 CXL_2         195*  1008  1422  1735

-D-
 DATA          928*   985=  1008
 DATASIZE      936*   962  1014
 DATASIZETABL   56*   283*   847   883=  883  1447  1465  1486  1492  1506  1520  1528  1745
 DB            269    486   725   752
 DECREMENT1    130*  1671
 DEFLABEL1     127*  1352
 DELAY1        166*
 DIF_2         213*  1202
 DIGIT         415*   417=  419=  419   420   422   423   424
 DISPL         666*   675=  680=  681=  692=  693=  704=  705=  717   720   724   725   729   740   744   747   751   752
              756
 DIV1          125*  1213
 DONE          278*  1044= 1743= 1756
 DUP1_2        214*  1093  1106  1267  1664  1673  1751
 DUP2_2        207*
 DUPTOS1       132*  1751
 DVI_2         194*  1216
 DVR_2         206*  1217

-E-
 EFJ_2         210*  1095  1111
 EMPTY1        161*  1249
 ENDPASS       864*  1763
 ENTER         802*   860  1354  1396  1398  1406  1447  1465  1486  1492  1506  1520  1528  1546  1597  1641  1646  1650
             1745
 ENTER1        128*  1434
 ENTR           814*  823= 1404  1428
 ENTRIES        789*  793
 EOL             25*  329   331   428
 EOM             25*
 EOM1           132* 1741
 EOM_2          217* 1747
 EQUAL          155* 1282  1294  1305  1322
 EQUBYT_2       204* 1322  1325
 EQUI_2         202* 1252  1282
 EQUPWR_2       203* 1305  1307
 EQUREAL_2      209* 1294  1297
 ERROR          480*  494   591*  611   622   632   642  1596  1659
 ERROR1         129* 1627
 EXITICTABLE     56*  283*  846   882=  882  1546

-F-
 F              102*  103*  104*
 FALSE          337   600   856   858  1029  1044  1598  1618  1628
 FALSEJUMP1     127* 1367
 FF              25*
 FIELD1         121* 1070
 FILE            42*  102   103   104
 FILE_LENGTH    104*  352   356   370   389   403
 FILE_LIMIT     334*  353   357   370   390   404
 FIVEWORDS      247*
 FIXGBL_OLDG    240*
 FIXLCL_LINE    238*  902  1358  1451  1523  1623
 FIXLCL_OLDG    239*  906
 FJPL_2         211* 1369
 FJP_2          211* 1139  1611  1706
 FLAG_REG       229*
 FLT_2          209* 1248
 FORWARD        480*
 FOURWORDS      247*
 FULLWORD        18*   73   105   106   274   941
 FUNCTION1      126* 1227
 FUNCVALUE1     127* 1331

-G-
 GBASE_REG      234*  904   912  1458  1470  1497  1511  1532
 GENERATE       277*  600=  840=  869  1628=
 GENNINGINTFA   280*  383   508   518   534   551   570   582   858=  948= 1029=
 GEN_EXIT       909* 1576  1578  1582  1586  1590
 GEN_INTERFAC   924* 1716
 GEN_SAVELINE   895* 1456  1468  1495  1509  1530
 GEQBYT_2       204* 1321  1324
 GEQI_2         202* 1281  1284
 GEQPWR_2       204* 1306
 GEQREAL_2      209* 1293  1296
 GET            102*  372
 GEUSW_2        203*
 GREATER        155* 1283  1295  1323

-H-
 HALFWORD       245*  486   488
 HIORDER        933* 1027
 HIWORD         889*  891=

-I-
 I              327*  330=  330   367*  378=  381*  386   400   415*  423=  423   424=  431*  433=  790*  793=  794   798=
                 798   802*  805   814*  817   831*  841=  842   842
 ID               85
 IDENTIFIER       45*   85
 IDLENGTH         44*   45
 IDTYPE           79    85*
 IFINFO           63    64*
 IFLABEL         944*  982   985
 IFPAGES_OUT     324*  346=  356   358   389   392   393=  393
 IFPAGE_OUT      324*  358   386=  392
 IFPTR            63*   75
 IFSEGNUM        305*  857=  950=  950   951   1027
 IFSEGSIZE       286*  842=  951
 IFWORDS_OUT     324*  346=  355   385=  385   386   387   395=
 INCREMENT1      130*  1662
 INC_2           215*  1073  1726
 INDEX1          121*  1076
 IND_2           215*  1107  1725
 INFILE          250*  370   372
 INIT1           131*  1701
 INITVAR1        128*
 INIT_PASS       340*  1760
 INN_2           212*  1309
 INSET           156*  1309  1312
 INT              84   364=
 INTEGER          18    19*   47    51    54    65    66    72    84    96   102   103   104   286   309   316   322   323
                 324   327   367   381   414   415   431   436   443   480   482   504   511   524   541   558   578   639
                 646   653   666   666   711   736   763   789   790   802   803   814   814   815   831   889   928   944
                1039
 INTERFACE        75*  842
 INTERFACEFIL    250*  356   358   389   392
 INTERFACES       65*
 INTERFACESIZ     66*  842
 INTFSEG         254*  1422
 INTTYPE          79    84*  364
 INT_2           213*  1173
 IO1             166*
 IXA_2           211*  1085
 IXP_2           212*

-J-
 J               802*  811   815*  817=  818   821=  821   823
 JUMP1           127*  1362
 JUMPTABLE        55*  282*  843   860   879=  879   1354  1404

-K-
 K               803*  805=  806   809=  809   811
 KASE            943*  958=  958   980=  995   999=  1009

-L-
 LABELS           72*  843
 LAE_2           197*
 LAND_2          199*  1109  1172
 LAO_2           192*  747   756
 LBASE_REG       233*  1459
 LCA_2           191*  740
 LDA_2           193*  752
 LDB_2           200*  767
 LDCB_2          191*  658
 LDCI_2          191*  659   1108
 LDCN_2          197*
```

```
LDC_2          191*
LDE_2          197*
LDL_2          192*  717   724  1498  1512  1533
LDM_2          210*  771   775  1665  1674
LDO_2          192*  720   729
LDP_2          208*
LENGTH         639*  641   641   646*  647
LEQBYT_2       204* 1323  1326
LEQI_2         202* 1283  1286
LEQPWR_2       203* 1308
LEQREAL_2      209* 1295  1298
LESS           155* 1281  1293  1321
LEUSH_2        203*
LINE           306*  599   854=  901  1357  1450  1522  1617=
LINK           273*  340*  342=  349*  362   833   853   853   866   867   868   871   872   878  1391  1760  1764
LISTOPTION      38*   59
LLA_2          192*  744   751
LNOT_2         215* 1167  1281  1283  1293  1295  1297  1307  1321  1323  1325
LOCATION       303*  484   486=  486   488=  488   489=  489   490=  490   495=  587   588   855=  878  1354  1404  1443=
              1443  1546  1599
LOD_2          193*  725
LOR_2          199* 1179
LOWORD         889*  891=
LPR_2          198*  905  1460  1724
LSL_2          197*
LSPLIM         230*

-M-
MAIN           108*
MARK           105*  848   977
MAXARG          89*   91
MAXDIGIT        26*  415   422   424
MAXINTFAC       61*   66   286   841
MAXTEMP        301*  612   612=  623   623=  633   633=  643   643= 1442= 1594  1595
MAXWORD         40*   51   793   798   806   809   818   821
MAX_FUNC       162* 1229
MAX_PROC       168*
MESSAGE1       130* 1657
MESSAGE_2      217*  596
MININDEX       934*  956
MIN_FUNC       162* 1229
MIN_PROC       168*
MOD1           125* 1220
MODE           666*  673   711*  713   717   720   724   725   729   736*  738   740   744   747   751   752   756
MODE0          139*  674   739
MODE1          140*  678   716   743  1335  1445  1548
MODE10         149*  707   731   758  1539  1592
MODE11         150*  696   728   755  1526  1588
MODE12         151*  678   716   743  1340  1504  1580
MODE2          141*  678   719   746  1453  1552
MODE3          142*  683   722   749  1335  1419  1463  1575
MODE4          143*  678   716   743  1340  1475  1484  1578
MODE5          144*  678   716   743  1340  1503  1579
MODE6          145*  696   728   755  1518  1584  1703
MODE7          146*  696   728   755  1476  1586
MODE8          147*  696   728   755  1525  1587
MODE9          148*
MODI_2         194* 1221
MOV_2          207* 1155
MPI_2          194* 1209
```

```
 MPR_2           206* 1210
 MSCWSIZE        241*  916  1426  1499  1513  1534  1550  1712
 MUL1            124* 1206

-N-
 NEG1            124* 1183
 NEQI_2          202* 1285
 NEW             792   797   871   979   989
 NEW1            123* 1158
 NEWLINE1        129* 1614
 NEWLIN_2        217*  900  1621
 NEWTOS          946*  979   983   988   989
 NEXTPORTION      50*  797   797   808   820
 NEXT_PASS       349* 1764
 NFJ_2           210* 1607
 NGI_2           214* 1186  1723
 NGR_2           215* 1187
 NILTYPE          79    83
 NOP_2           198*  970  1013  1600
 NOT1            123* 1166
 NOTEQUAL        156* 1285  1297  1307  1325
 NOTGREATER      156* 1286  1298  1308  1326
 NOTLESS         155* 1204  1296  1306  1324
 NOTME           269   491   596   597   598   599   900  1621  1747
 NOTUSED1        100*
 NOTUSED2        101*
 NUMBER          277*  480*  598   839=  898  1355  1619
 NUMBEROPTION     39*  839
 NXTENTRY        929*  986= 1010

-O-
 OK              318*  337=  343=  360
 OLDHEAPTOP      941*  977  1015
 ONEWORD         246*  489   490   675   693   855  1391  1488  1493  1508  1529  1610
 OP              307*  436*  439   504*  506   507   508   511*  515   517   520   524*  529   532   536   541*  546   549
                 553   558*  563   567   572   996  1042  1046  1047  1736
 OPTION           59*   71
 OPTIONS          71*  835   836   837   838   839   869=  869
 OPTR            269   486   508   520   536   553   572
 OR1             123* 1176
 ORD             420
 OUTFILE         250*  352   354   403   406

-P-
 P               102*  103*
 PAGE             96*  102   103   322   323   324
 PAGELENGTH       94*   96   344   369   387   401
 PAGES_IN        322*  344=  370   372   373=  373
 PAGES_OUT       323*  345=  352   354   364   403   406   407=  407
 PAGE_IN         322*  372   378
 PAGE_OUT        323*  354   400=  406
 PARAM           108*  342   359   361   363
 PARAMLENGTH     298* 1437= 1449  1488  1508
 PARMWORDS       937* 1014
 PASS            480*  597
 PASSLINK         69    70*
 PASSPTR          69*   86   273   340   349
 PC_REG          235*
 PDP11           242*
 PICK_PUSHCON    653*  777   901   904   912  1051  1078  1079  1081  1120  1121  1265  1357  1384  1385  1421  1450  1458
```

```
                 1459  1470  1497  1499  1511  1513  1522  1532  1534  1622
 POINTER           47*
 POINTER1         121* 1089
 POOLSIZE         290*  853=  855
 POP              646*  906   914   916  1080  1095  1111  1122  1155  1266  1386  1431  1461  1472  1501  1515  1536  1550
                 1607  1668  1677  1710  1734
 POP1             129* 1603
 POPLENGTH        299*  915   916  1449= 1467= 1488= 1493= 1498  1508= 1512  1521= 1529= 1533  1549  1550  1573  1585
 POPREAL          626* 1135  1194  1201  1210  1217  1232  1300  1300
 POPSET           636* 1140  1173  1180  1202  1273  1311  1313
 POPWORD          615*  767   772   776   902  1082  1085  1130  1132  1139  1143  1163  1172  1173  1179  1180  1193  1200
                 1202  1209  1216  1221  1240  1245  1248  1252  1273  1288  1311  1328  1358  1370  1392  1451  1500  1514
                 1523  1535  1611  1623  1667  1676  1706
 PORTION          790*  792=  795   797=  803*  805=  808=  808   811   815*  817=  820=  820   823
 PRED1            160* 1242
 PRINTABS         414*  439   447
 PRINTARG         443*  515   530   530   546   547   547   564   564   565   565   580   587
 PRINTED          316*  428=  438   440=  440   445   448=  448
 PRINTEOL         427*  433   433   438   445
 PRINTFF          430*  861
 PRINTLIMIT        27*  438   445
 PRINTOP          436*  506   515   529   546   563
 PRINT_TEXT       326*  336
 PRIORITY_REQ     228*
 PROCEDURE1       131* 1680
 PROCSGEND        939* 1027
 PTR               86   342   362=
 PTRTYPE           79    86*  362
 PUSH             639* 1426  1431  1712
 PUSHADDR1        120* 1067
 PUSHADDRESS      736* 1059  1068
 PUSHCONST1       120* 1049
 PUSHIND1         120* 1064
 PUSHINDIRECT     763* 1060  1065
 PUSHLABEL1       131* 1716
 PUSHREAL         619*  772  1248  1338  1347
 PUSHSET          629*  776  1268
 PUSHVALUE        711* 1057
 PUSHVAR1         120* 1054
 PUSHWORD         608*  733   760   778   901   904   912   913  1051  1078  1079  1081  1093  1094  1106  1108  1120  1121
                ·1232  1239  1244  1251  1263  1265  1267  1268  1300  1337  1343  1357  1384  1385  1421  1450  1458  1459
                 1470  1471  1497  1498  1499  1511  1512  1513  1522  1532  1533  1534  1622  1664  1665  1666  1673  1674
                 1675  1722  1751
 PUT              103*  354   358   392   406

-Q-
 QLINK_REQ        226*

-R-
 RANGE1           122* 1116
 RBP_2            198*
 READ              98*
 READ1ARG         455* 1042  1051  1065  1071  1127  1154  1170  1177  1184  1191  1198  1207  1214  1221  1353  1363  1368
                 1403  1544  1604  1616  1632  1638  1681  1736  1744
 READ2ARG         458* 1068  1103  1117  1159  1228  1277  1319  1332  1381  1658
 READ3ARG         461* 1055  1418
 READ4ARG         464* 1077  1702
 READ5ARG         470* 1435
 READYQ_REQ       223*
 READ_IFL         367*  456   459   459   462   462   462   466   466   467   467   472   472   472   473   473   982   996
```

```
                1042  1046  1736
REALLENGTH        29*  621   627   771  1135
REALTIME1        161* 1256
REALTYPE         135*  770  1133  1201  1291
RELEASE          106*  870  1015
REM              415*  417=  420   421=  421   422
RESETPOINT        73*
RETURN1          129* 1543
RND_2            205*
RPU_2            196*  915  1014  1549  1573  1585
RTN3             942*  951=  953   957   958   961   971   980   995   999  1009  1017

-S-
SB               269   486  1095  1111  1139  1607  1611  1706
SBI_2            199* 1082  1200  1245  1500  1514  1535  1676
SDR_2            206* 1201
SCAN            1038* 1762
SEGDICT_REG      224*
SEGDISTANCE       54*  878=
SEG_REG          236*
SEQL_SEG         253* 1735
SETHEAP1         167*
SETLENGTH         30*  631   637   775   777  1140  1265
SETTYPE          135*  774  1137  1202  1303
SHORTINTEGER      19   261
SIBS_OFFSET      252* 1725
SIGNAL_2         213*
SIND0_2          189*  769
SIND1_2          189*
SIND2_2          189*
SIND3_2          189*
SIND4_2          190*
SIND5_2          190*
SIND6_2          190*
SIND7_2          190*
SLDC00_2         172* 1094  1110  1251  1263  1337  1338  1338  1343  1346  1347
SLDC01_2         172* 1239  1244  1666  1675  1722
SLDC02_2         172*
SLDC03_2         172*
SLDC04_2         173*
SLDC05_2         173*
SLDC06_2         173*
SLDC07_2         173*
SLDC08_2         174*
SLDC09_2         174*
SLDC10_2         174*
SLDC11_2         174*
SLDC12_2         175*
SLDC13_2         175*
SLDC14_2         175*
SLDC15_2         175*
SLDC16_2         176*
SLDC17_2         176*
SLDC18_2         176*
SLDC19_2         176*
SLDC20_2         177*
SLDC21_2         177*
SLDC22_2         177*
SLDC23_2         177*
SLDC24_2         178*
```

```
SLDC25_2        178*
SLDC26_2        178*
SLDC27_2        178*
SLDC28_2        179*
SLDC29_2        179*
SLDC30_2        179*
SLDC31_2        179*
SLDL01_2        180*  968
SLDL02_2        180*  913
SLDL03_2        180*
SLDL04_2        180*
SLDL05_2        181*
SLDL06_2        181*
SLDL07_2        181*
SLDL08_2        181*
SLDL09_2        182*
SLDL10_2        182*
SLDL11_2        182*
SLDL12_2        182*
SLDL13_2        183*
SLDL14_2        183*
SLDL15_2        183*
SLDL16_2        183*
SLDO01_2        184*
SLDO02_2        184* 1471
SLDO03_2        184*
SLDO04_2        184*
SLDO05_2        185*
SLDO06_2        185*
SLDO07_2        185*
SLDO08_2        185*
SLDO09_2        186*
SLDO10_2        186*
SLDO11_2        186*
SLDO12_2        186*
SLDO13_2        187*
SLDO14_2        187*
SLDO15_2        187*
SLDO16_2        187*
SOURCE_WORD_     36* 1635  1635  1647
SPLITLENGTH     244*
SPLIT_CONST     888* 1645
SPLIT_INTEGE    261*  888
SPR_2           210*  914  1461  1472  1501  1515  1536
SRO_2           200*
SRS_2           205* 1268
STACKLENGTH      54*  291* 1439= 1594  1595= 1595  1597
STACKPTR_REG    232*
STACKTABLE      282*  849  1428  1597
STACK_ERROR     256*  611   622   632   642  1596
STACK_LIMIT     248*  610   621   631   641  1594
START1          166*
STARTPROC       255*  860  1422  1735  1745
STB_2           208* 1130
STE_2           212*
STKENTRY        926   927*
STL_2           200*  902   906  1358  1451  1523  1623
STM_2           194* 1135  1140  1709
STOP1           167*
STO_2           207* 1132  1668  1677  1734
```

```
 STP_2          208*
 STR_2          200*
 SUB1           124* 1197
 SUCC           373   377   385   393   399   407   950  1395  1397  1405
 SUCC1          160* 1237
 SUMMARY        276*  835=
 SUMMARYOPTIO    38*  835
 SWAP_2         205* 1343  1346  1347  1733

-T-
 T              415*  420=  423   789*  792   792   802*  805   814*  817
 TABLE           48    49*
 TABLEPART       53*   58
 TABLEPTR        48*   50    56   283   789   790   802   803   814   815
 TABLERESET     274*  848   870
 TABLES          74*  871   878   879   880   881   882   883
 TABLESPTR       58*   74
 TAG             82*  360=  362=  364=
 TASK_REG       225*
 TEMP           300*  610   610=  610   612   612   616=  616   621   621=  621   623   623   627=  627   631   631=  631
                633   633   637=  637   641   641=  641   643   643   647=  647  1441=
 TEST           276*  506   513   527   544   561   580   587   836=  861
 TESTOPTION      38*  836
 TEXT           326*  330
 TEXT_LENGTH     90*   92   330
 TEXT_TYPE       92*  326
 THIS_PASS      249*  494   611   622   632   642  1596
 THREEWORDS     247*
 TNC_2          205* 1232
 TOP            105*  106*
 TOS            945*  978=  986   988= 1008  1010= 1010
 TOSPTR         926*  929   945   946
 TRUE           242   243   343   595   840   948  1440  1743
 TRUNC1         160* 1231
 TWOWORDS       247*  680   681   692   906   914  1080  1095  1111  1122  1155  1266  1386  1443  1447  1449  1461  1465
               1467  1472  1486  1489  1492  1493  1501  1506  1508  1515  1520  1521  1528  1529  1536  1605  1607  1608
               1668  1677  1734  1746
 TYPEOFCODE     269*  482   511   525   542   559   578

-U-
 UB             269   486   658   771   775  1008  1008  1135  1140  1422  1422  1426  1665  1674  1709  1711  1735  1735
 UJPL_2         193*  971  1009  1364
 UJP_2          193*
 UNI_2          212* 1180  1273
 UPDLOC         482*  508   520   520   536   537   537   553   553   554   554   572   573   573   574   574   582
 USPLIM         231*

-V-
 VALUE          482*  487   653*  655   655   656   657   657   658   659
 VARIANT1       121* 1100
 VARLENGTH      297*  681   705  1438= 1447  1449  1465  1467  1486  1489  1506  1508  1594  1595  1744= 1746
 VARNTCHECK     278*  838= 1104
 VARNTCHECKOP    39*   59   838
 VARTYPE        763*  765

-W-
 W              269   490   659   953   956   957   958   961   962   971  1009  1017  1027  1108  1364  1369
 WAIT1          167*
 WAIT_2         213*
 WORD           888*  891   891
 WORDLENGTH      28#  610  616  675  680  681  692  693  704  705  771  775  777  844  845  915 1073 1085
               1107 1135 1140 1155 1391 1423 1429 1448 1466 1487 1492 1498 1507 1512 1520 1528 1533 1549
               1573 1585 1647 1709 1710 1746
 WORDS_IN       322# 344= 369  375= 377= 377  378
 WORDS_OUT      323# 345= 351  399= 399  400  401  409=
 WORDTYPE       135# 768 1056 1131 1171 1178 1185 1192 1200 1208 1215 1234 1279 1336 1341
 WRITE           99# 329  330  331  423  424  428  433  439  446  446
 WRITE1         504# 656  767  769  905  913  914  968  970 1013 1080 1082 1093 1094 1106 1109 1110 1122
               1130 1132 1167 1172 1173 1179 1180 1186 1187 1193 1194 1200 1201 1202 1209 1210 1216 1217
               1221 1232 1235 1236 1239 1240 1244 1245 1248 1251 1252 1263 1266 1267 1268 1273 1281 1281
               1282 1283 1283 1284 1285 1286 1293 1293 1294 1295 1295 1296 1297 1297 1298 1305 1306 1307
               1307 1308 1309 1321 1323 1325 1337 1338 1338 1343 1343 1346 1346 1347 1347 1386 1460 1461
               1471 1472 1500 1501 1514 1515 1535 1536 1600 1664 1666 1667 1668 1673 1675 1676 1677 1722
               1723 1724 1733 1734 1751
 WRITE2         511# 658  659  717  720  724  729  740  744  747  751  756  771  775  902  906  915  969
                971 1009 1014 1073 1085 1095 1107 1108 1111 1135 1139 1140 1155 1321 1322 1323 1324 1325
               1326 1358 1364 1369 1391 1426 1451 1498 1512 1523 1533 1549 1573 1585 1607 1611 1623 1665
               1674 1706 1709 1711 1725 1726
 WRITE3         524# 725  752 1008 1422 1735
 WRITE4         541#
 WRITE5         558#
 WRITEARG       578# 596  597  598  599  900  953  956  957  958  961  962 1017 1027 1621 1747
 WRITELOCATIO   585# 1364 1369
 WRITE_IFL      381# 507  517  517  532  533  533  549  549  550  550  567  568  568  569  569  581  588

-X-
 XJPOFFSETPTR   289# 852= 867 1392 1395= 1395 1396 1397= 1397 1398 1405= 1405 1406
 XJPTABLE        55# 282# 845  881= 881  938# 969 1396 1398 1406
 XJP_2          211# 969 1391
 XJP_OFFSETS     72# 845  853  867=

END XREF  538 IDENTIFIERS  2678 TOTAL REFERENCES
 745 COLLISIONS.
```

APPENDIX C

SOURCE CODE FOR MEPASCAL COMPILER DRIVER: MEPASCAL

```
     1
     2 "KANSAS STATE UNIVERSITY"
     3 "DEPARTMENT OF COMPUTER SCIENCE"
     4 "DRIVER PROGRAM FOR MEPASCAL COMPILER"
     5
     6 "############
C    7 #  PREFIX  #
C    8 ############"
     9
    10 TYPE FULLWORD = INTEGER;                                                    *     S
    11      INTEGER  = SHORTINTEGER;                                               S     *
    12
    13 CONST NL = '(:10:)';  FF = '(:12:)';  CR = '(:13:)';  EM = '(:25:)';        *     *     *     *
    14
    15 CONST PAGELENGTH = 512;                                                     *
    16 TYPE PAGE = ARRAY (.1..PAGELENGTH.) OF CHAR;                                *     15    S
    17
    18 CONST LINELENGTH = 132;                                                     *
    19 TYPE LINE = ARRAY (.1..LINELENGTH.) OF CHAR;                                *     18    S
    20
    21 CONST IDLENGTH = 12;                                                        *
    22 TYPE IDENTIFIER = ARRAY (.1..IDLENGTH.) OF CHAR;                            *     21    S
    23
    24 TYPE FILE = 1..2;                                                           *
    25
    26 TYPE FILEKIND = (EMPTY, SCRATCH, ASCII, SEQCODE, CONCODE);                  *     *     *     *     *     *
    27
    28 TYPE FILEATTR = RECORD                                                      *
    29                   KIND: FILEKIND;                                           *     26
    30                   ADDR: INTEGER;                                            *     S
    31                   PROTECTED: BOOLEAN;                                       *     S
    32                   NOTUSED: ARRAY (.1..5.) OF INTEGER                        *     S
    33                 END;
    34
    35 TYPE IODEVICE =                                                             *
    36   (TYPEDEVICE, DISKDEVICE, TAPEDEVICE, PRINTDEVICE, CARDDEVICE);            *     *     *     *     *
    37
    38 TYPE IOOPERATION = (INPUT, OUTPUT, MOVE, CONTROL);                          *     *     *     *     *
    39
    40 TYPE IOARG = (WRITEEOF, REWIND, UPSPACE, BACKSPACE);                        *     *     *     *     *
    41
    42 TYPE IORESULT =                                                             *
    43   (COMPLETE, INTERVENTION, TRANSMISSION, FAILURE,                           *     *     *     *
    44    ENDFILE, ENDMEDIUM, STARTMEDIUM);                                        *     *     *
    45
    46 TYPE IOPARAM = RECORD                                                       *
    47                  OPERATION: IOOPERATION;                                    *     38
    48                  STATUS: IORESULT;                                          *     42
    49                  ARG: IOARG                                                 *     40
    50                END;
    51
    52 TYPE TASKKIND = (INPUTTASK, JOBTASK, OUTPUTTASK);                           *     *     *     *
    53
    54 TYPE ARGTAG =                                                               *
    55   (NILTYPE, BOOLTYPE, INTTYPE, IDTYPE, PTRTYPE);                            *     *     *     *     *
    56
    57 TYPE POINTER = @INTEGER;                                                    *     S
    58
    59 TYPE PASSPTR = @PASSLINK;                                                   *     *
```

```
 60
 61 TYPE OPTION = (LISTOPTION,SUMMARYOPTION,TESTOPTION,CHECKOPTION,             *     *     *     *     *
 62        CODEOPTION,NUMBEROPTION,XREFOPTION,DUMPOPTION);                      *     *     *     *
 63
 64 TYPE PASSLINK = RECORD                                                      *
 65   OPTIONS: SET OF OPTION;                                                   *     61
 66   LABELS, BLOCKS, CONSTANTS. XJP_OFFSETS: INTEGER;                          *     *     *     *     S
 67   RESETPOINT: POINTER;                                                      *     57
 68   TABLES: POINTER;                                                          *     57
 69   INTERFACE: POINTER;                                                       *     57
 70   END;
 71
 72
 73 TYPE ARGTYPE = RECORD                                                       *
 74                  CASE TAG: ARGTAG OF                                        *     54
 75                    NILTYPE, BOOLTYPE: (BOOL: BOOLEAN);                      55    55    *     S
 76                    INTTYPE: (INT: INTEGER);                                 *     *     S
 77                    IDTYPE: (ID: IDENTIFIER);                                *     *     22
 78                    PTRTYPE: (PTR: PASSPTR)                                  *     *     59
 79                END;
 80
 81 CONST MAXARG = 10;                                                          *
 82 TYPE ARGLIST = ARRAY (.1..MAXARG.) OF ARGTYPE;                              *     81    73
 83
 84 TYPE ARGSEQ = (INP, OUT);                                                   *     *     *
 85
 86 TYPE PROGRESULT =                                                           *
 87   (TERMINATED, OVERFLOW, POINTERERROR, RANGEERROR, VARIANTERROR,            *     *     *     *     *
 88    HEAPLIMIT, STACKLIMIT, CODELIMIT, TIMELIMIT, CALLERROR);                 *     *     *     *     *
 89
 90 PROCEDURE READ(VAR C: CHAR);                                                *     *     S
 91 PROCEDURE WRITE(C: CHAR);                                                   *     *     S
 92
 93 PROCEDURE OPEN(F: FILE; ID: IDENTIFIER; VAR FOUND: BOOLEAN);                *     *     24    *     22    *     S
 94 PROCEDURE CLOSE(F: FILE);                                                   *     *     24
 95 PROCEDURE GET(F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);                   *     *     24    *     S     *     16
 96 PROCEDURE PUT(F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);                   *     *     24    *     S     *     16
 97 FUNCTION LENGTH(F: FILE): INTEGER;                                          *     *     24    S
 98
 99 PROCEDURE MARK(VAR TOP: INTEGER);                                           *     *     S
100 PROCEDURE RELEASE(TOP: INTEGER);                                            *     *     S
101
102 PROCEDURE IDENTIFY(HEADER: LINE);                                           *     *     19
103 PROCEDURE ACCEPT(VAR C: CHAR);                                              *     *     S
104 PROCEDURE DISPLAY(C: CHAR);                                                 *     *     S
105
106 PROCEDURE NOTUSED;                                                          *
107 PROCEDURE NOTUSED2;                                                         *
108 PROCEDURE NOTUSED3;                                                         *
109 PROCEDURE NOTUSED4;                                                         *
110 PROCEDURE NOTUSED5;                                                         *
111 PROCEDURE NOTUSED6;                                                         *
112
113 PROCEDURE NOTUSED7;                                                         *
114
115 PROCEDURE NOTUSED8;                                                         *
116
117 PROCEDURE NOTUUED9;                                                         *
118
119 PROCEDURE NOTUSED10;                                                        *
```

```
   120
   121 PROCEDURE RUN(ID: IDENTIFIER; VAR PARAM: ARGLIST;                        *    *    22    *    82
   122               VAR LINE: INTEGER; VAR RESULT: PROGRESULT);                *    S    *    86
   123
   124 PROCEDURE BREAKPNT(LINE: INTEGER);                                       *    *    S
   125
   126 PROGRAM P(PARAM: LINE);                                                  *    *    124
   127
   128
   129
   130
   131 "################################################################
C  132 #  NCPASCAL(VAR OK: BOOLEAN; SOURCE, DEST, OBJECT: IDENTIFIER) #
C  133 ################################################################"
   134
   135
   136 "INSERT PREFIX HERE"
   137
   138 "THE PARAMETERS OF THE COMPILER PASSES
C  139 HAVE THE FOLLOWING MEANING:
C  140
C  141   LIST(.1.) : BOOLEAN     (COMPILATION OK)
C  142   LIST(.2.) : POINTER     (HEAP POINTER)
C  143   LIST(.3.) : INTEGER     (CODE LENGTH) "
   144
   145 TYPE CHARSET = SET OF CHAR;                                              *    S
   146
   147 TYPE PASSES = 0..127;                                                    *
   148
   149 VAR
   150
   151 OK. PAST_EOM: BOOLEAN; SOURCE, DEST, OBJECT: ARGTYPE;                    *    *    S    *    *    *    73
   152 CODELENGTH: INTEGER;                                                     *    S
   153 WHERE: (NOWHERE, ONDISK);                                                *    *    *
   154 REPORT: (MAIN, OUTP);                                                    *    *    *
   155 I: INTEGER;                                                              *    S
   156
   157 LIST: ARGLIST;                                                           *    82
   158
   159 PARAMPTR: 1..LINELENGTH;                                                 *    18
   160 TESTPASS: SET OF PASSES;                                                 *    147
   161 NOSOURCE: BOOLEAN;                                                       *    S
   162
   163
   164 PROCEDURE INITWRITE;                                                     *
   165 BEGIN
   166   IDENTIFY('PASCAL:(:10:)');                                           102
   167   REPORT:= MAIN;                                                       154  154
   168 END;
   169
   170 PROCEDURE WRITECHAR(C: CHAR);                                            *    *    S
   171 BEGIN
   172   IF REPORT <> MAIN                                                    154  154
   173     THEN WRITE(C);                                                      91  170
   174   DISPLAY(C);                                                          104  170
   175 END;
   176
   177 PROCEDURE WRITETEXT(TEXT: LINE);                                         *    *    124
   178 CONST NUL = '(:0:)';                                                     *
   179 VAR I: INTEGER; C: CHAR;                                                 *    S    *    S
```

```
180 BEGIN
181   I:= 1; C:= TEXT(.1.);                                                179   179   177
182   WHILE C <> NUL DO                                                    179   178
183   BEGIN
184     WRITECHAR(C);                                                      170   179
185     I:= I + 1; C:= TEXT(.I.);                                          179   179   179   177   179
186   END;
187 END;
188
189 PROCEDURE WRITEINT(INT, LENGTH: INTEGER);                                *     *     *     S
190 VAR NUMBER: ARRAY (.1..6.) OF CHAR;                                      *     S
191   DIGIT, REM, I: INTEGER;                                                *     *     *     S
192 BEGIN
193   DIGIT:= 0; REM:= INT;                                                191   191   189
194   REPEAT
195     DIGIT:= DIGIT + 1;                                                 191   191
196     NUMBER(.DIGIT.):=                                                  190   191
197       CHR(ABS(REM MOD 10) + ORD('0'));                                   *     *   191     *
198     REM:= REM DIV 10;                                                  191   191
199   UNTIL REM = 0;                                                       191
200   FOR I:= 1 TO LENGTH - DIGIT - 1 DO                                   191   189   191
201     WRITECHAR(' ');                                                    170
202   IF INT < 0 THEN WRITECHAR('-')                                       189   170
203                ELSE WRITECHAR(' ');                                    170
204   FOR I:= DIGIT DOWNTO 1 DO                                            191   191
205     WRITECHAR(NUMBER(.I.));                                            170   190   191
206 END;
207
208 PROCEDURE WRITEID(ID: IDENTIFIER);                                       *     *    22
209 VAR I: INTEGER; C: CHAR;                                                 *     S     *     S
210 BEGIN
211   FOR I:= 1 TO IDLENGTH DO                                             209    21
212   BEGIN
213     C:= ID(.I.);                                                       209   208   209
214     IF C <> ' ' THEN WRITECHAR(C);                                     209   170   209
215   END;
216 END;
217
218 PROCEDURE CONVRESULT(RESULT: PROGRESULT; VAR ID: IDENTIFIER);            *     *    86     *    22
219 BEGIN
220   CASE RESULT OF                                                       218
221     TERMINATED:    ID:= 'TERMINATED  ';                                 87   218
222     OVERFLOW:      ID:= 'OVERFLOW    ';                                 87   218
223     POINTERERROR:  ID:= 'POINTERERROR';                                 87   218
224     RANGEERROR:    ID:= 'RANGEERROR  ';                                 87   218
225     VARIANTERROR:  ID:= 'VARIANTERROR';                                 87   218
226     HEAPLIMIT:     ID:= 'HEAPLIMIT   ';                                 88   218
227     STACKLIMIT:    ID:= 'STACKLIMIT  ';                                 88   218
228     CODELIMIT:     ID:= 'CODELIMIT   ';                                 88   218
229     TIMELIMIT:     ID:= 'LOADERROR   ';                                 88   218
230     CALLERROR:     ID:= 'CALLERROR   '                                  88   218
231   END;
232 END;
233
234 PROCEDURE WRITERESULT                                                    *
235   (ID: IDENTIFIER; LINE: INTEGER; RESULT: PROGRESULT);                   *    22     *     S     *    86
236 VAR ARG: IDENTIFIER;                                                     *    22
237 BEGIN
238   WRITECHAR(NL);                                                       170    13
239   WRITEID(ID);                                                         208   235
```

```
  240   WRITETEXT(': LINE (:0:)');                                            177
  241   WRITEINT(LINE, 4);                                                    189   235
  242   WRITECHAR(' ');                                                       170
  243   CONVRESULT(RESULT, ARG);                                              218   235   236
  244   WRITEID(ARG);                                                         208   236
  245   WRITECHAR(NL);                                                        170    13
  246   OK:= (RESULT = TERMINATED);                                           151   235    87
  247 END;
  248
  249 PROCEDURE ERROR(TEXT: LINE);                                              *     *   235
  250 BEGIN
  251   INITWRITE;                                                            164
  252   WRITETEXT(TEXT);                                                      177   249
  253   OK:= FALSE;                                                           151     *
  254 END;
  255
  256 FUNCTION NEXTCHAR: CHAR;                                                  *     S
  257 BEGIN
  258   PARAMPTR:=PARAMPTR+1;                                                 159   159
  259   NEXTCHAR:=PARAM[PARAMPTR];                                            256   126   159
  260 END;
  261
  262 PROCEDURE SCANPARMS;                                                      *
  263 VAR SKIPS,ALPHAS,NUMERICS: SET OF CHAR;                                   *     *     *     S
  264     ID: IDENTIFIER;                                                       *    22
  265     C: CHAR;                                                              *     S
  266     N,IDPTR: INTEGER;                                                     *     *     S
  267 BEGIN
  268   SKIPS:=[',',' '];                                                     263
  269   ALPHAS:=[]; FOR C:='A' TO 'Z' DO ALPHAS:=ALPHAS OR [C];               263   265   263   263   265
  270   NUMERICS:=[]; FOR C:='0' TO '9' DO NUMERICS:=NUMERICS OR [C];         263   265   263   263   265
  271   C:=PARAM[1]; PARAMPTR:=1;                                             265   126   159
  272   WHILE C<>NL DO BEGIN                                                  265    13
  273     WHILE C IN SKIPS DO C:=NEXTCHAR;                                    265   263   265   256
  274     IF C IN ALPHAS THEN BEGIN                                           265   263
  275       ID:='            '; IDPTR:=1;                                     264   266
  276       WHILE C IN (ALPHAS OR NUMERICS) DO                                265   263   263
  277         IF IDPTR<=IDLENGTH THEN BEGIN                                   266    21
  278           ID[IDPTR]:=C; IDPTR:=IDPTR+1; C:=NEXTCHAR;                    264   266   265   266   266   265   256
  279         END ELSE C:=NEXTCHAR;                                           265   256
  280       "IF ID[1]='N' THEN OPTIONS:=OPTIONS-[ID[2]]
C 281         ELSE OPTIONS:=OPTIONS OR [ID[1]];"
  282       IF ID='NS          ' THEN NOSOURCE:=TRUE;                         264   161     *
  283       IF ID='ALL         '                                              264
  284       THEN FOR N := 1 TO 7 DO TESTPASS := TESTPASS OR [N];              266   160   160   266
  285       END
  286     ELSE IF C IN NUMERICS THEN BEGIN                                    265   263
  287       N:=0;                                                             266
  288       WHILE C IN NUMERICS DO BEGIN                                      265   263
  289         N:=N*10+ORD(C)-ORD('0'); C:=NEXTCHAR;                           266   266   197   265   197   265   256
  290       END;
  291       TESTPASS:=TESTPASS OR [N];                                        160   160   266
  292     END
  293     ELSE IF C<>NL THEN C:=NEXTCHAR;                                     265    13   265   256
  294   END "WHILE";
  295 END;
  296
  297 PROCEDURE CHECKIO;                                                        *     .
  298 VAR C:CHAR;                                                               *     S
  299 BEGIN
```

```
300   "COMPLETE SOURCE TEXT INPUT/OUTPUT:"
301   IF NOT PAST_EOM THEN REPEAT READ(C) UNTIL C=EM;                151   90   298   298   13
302   WRITE(EM);                                                      91   13
303 END;
304
305 PROCEDURE INITIALIZE;                                             *
306 BEGIN
307   WRITECHAR (NL);                                                170   13
308   TESTPASS:=[]; NOSOURCE:=FALSE;                                 160  161   253
309   WITH LIST(.1.) DO                                              157
310   BEGIN TAG:= BOOLTYPE; BOOL:= FALSE END;                         74   55    75   253
311   WITH LIST(.2.) DO                                              157
312   BEGIN TAG:=PTRTYPE; PTR:=NIL END;                               74   78    78
313   WITH LIST(.3.) DO                                              157
314   BEGIN TAG:= INTTYPE; INT:= 0 END;                               74   76   189
315   SCANPARMS;                                                     262
316   INITWRITE;                                                     164
317   WRITETEXT (                                                    177
318   'MEPASCAL COMPILER (MICROENGINE P-CODE) (:10:)(:0:)');
319 END;
320
321 PROCEDURE CALLPASS(PASSNO: INTEGER; ID: IDENTIFIER);               *    *     S     *    22
322 VAR LINE: INTEGER; RESULT: PROGRESULT;                             *    S     *    86
323 BEGIN
324   LIST(.1.).BOOL:= FALSE;                                        157   75   253
325   RUN(ID, LIST, LINE, RESULT);                                   121  321   157   322  322
326   IF RESULT <> TERMINATED THEN                                   322   87
327   BEGIN
328     REPORT:= OUTP;                                               154  154
329     WRITERESULT(ID, LINE, RESULT);                               234  321   322   322
330   END ELSE
331   BEGIN
332     OK:= LIST(.1.).BOOL;                                         151  157    75
333     CODELENGTH:= LIST(.3.).INT;                                  152  157   189
334     IF NOT OK THEN                                               151
335       ERROR('COMPILATION ERRORS(:10:)(:0:)');                    249
336     IF OK                                                        151
337       THEN BEGIN
338         WRITEID(ID); WRITETEXT('  OK(:10:)(:0:)');               208  321   177
339       END;
340   END;
341 END;
342
343
344 PROCEDURE RUN_MNEM (PASS: INTEGER);                                *    *     S
345 VAR
346     L:INTEGER;                                                     *    S
347     R: PROGRESULT;                                                 *   86
348 BEGIN
349     LIST[10].TAG := INTTYPE;                                     157   74    76
350     LIST[10].INT := PASS;                                        157  189   344
351     IF (PASS = 5) OR                                             344
352        (PASS = 6)                                                344
353     THEN RUN ('MEMNEM      ', LIST, L, R);                       121  157   346   347
354     CASE PASS OF                                                 344
355        1: WRITETEXT ('NO MEMNEM 1(:10:)(:0:)');                  177
356        2: WRITETEXT ('NO MEMNEM 2(:10:)(:0:)');                  177
357        3: WRITETEXT ('NO MEMNEM 3(:10:)(:0:)');                  177
358        4: WRITETEXT ('NO MEMNEM 4(:10:)(:0:)');                  177
359        5: WRITETEXT ('MEMNEM  5  DONE(:10:)(:0:)');              177
```

```
        6: WRITETEXT ('MEMNEM  6  DONE(:10:)(:0:)');                   177
        7: WRITETEXT ('NO MEMNEM 7(:10:)(:0:)');                       177
     END;
 END;


 BEGIN
       INITIALIZE;                                                     305
       IF NOSOURCE THEN CALLPASS(1,'MEPASS1N    ')                     161   321
       ELSE CALLPASS(1,'MEPASS1     ');                                321
       PAST_EOM:= LIST[5].BOOL;                                        151   157    75
       IF (OK AND (1 IN TESTPASS))                                     151   160
       THEN RUN_MNEM (1);                                              344

       IF OK THEN CALLPASS(2,'MEPASS2     ');                          151   321
       IF (OK AND (2 IN TESTPASS))                                     151   160
       THEN RUN_MNEM(2);                                               344

       IF OK THEN CALLPASS(3,'MEPASS3     ');                          151   321
       IF (OK AND (3 IN TESTPASS))                                     151   160
       THEN RUN_MNEM(3);                                               344

       IF OK THEN CALLPASS(4,'MEPASS4     ');                          151   321
       IF (OK AND (4 IN TESTPASS))                                     151   160
       THEN RUN_MNEM(4);                                               344

       IF OK THEN CALLPASS(5,'MEPASS5     ');                          151   321
       IF (OK AND (5 IN TESTPASS))                                     151   160
       THEN RUN_MNEM(5);                                               344

       IF OK THEN CALLPASS(6,'MEPASS6     ');                          151   321
       IF (OK AND (6 IN TESTPASS))                                     151   160
       THEN RUN_MNEM(6);                                               344

       IF OK THEN CALLPASS(7,'MEPASS7     ');                          151   321
       IF (OK AND (7 IN TESTPASS))                                     151   160
       THEN RUN_MNEM(7);                                               344
 END.
```

```
CROSS REFERENCE   * IS DEF   = IS ASG

 -A-
  ABS            197
  ACCEPT         103*
  ADDR            30*
  ALPHAS         263*  269=  269=  269   274   276
  ARG             49*  236*  243   244
  ARGLIST         82*  121   157
  ARGSEQ          84*
  ARGTAG          54*   74
  ARGTYPE         73*   82   151
  ASCII           26

 -B-
  BACKSPACE       40
  BLOCK           95*   96*
  BLOCKS          66*
  BOOL            75   310=  324=  332   370
  BOOLEAN         31    75    93   151   161
  BOOLTYPE        55    75   310
  BREAKPNT       124*

 -C-
  C               90*   91*  103*  104*  170*  173   174   179*  181=  182   184   185=  209*  213=  214   214   265*  269=
                 269   270=  270   271=  272   273   273=  274   276   278   278=  279=  286   288   289   289=  293   293=
                 298*  301   301
  CALLERROR       88   230
  CALLPASS       321*  368   369   374   378   382   386   390   394
  CARDDEVICE      36
  CHAR            16    19    22    90    91   103   104   145   170   179   190   209   256   263   265   298
  CHARSET        145*
  CHECKIO        297*
  CHECKOPTION     61
  CHR            197
  CLOSE           94*
  CODELENGTH     152*  333=
  CODELIMIT       88   228
  CODEOPTION      62
  COMPLETE        43
  CONCODE         26
  CONSTANTS       66*
  CONTROL         38
  CONVRESULT     218*  243
  CR              13*

 -D-
  DEST           151*
  DIGIT          191*  193=  195=  195   196   200   204
  DISKDEVICE      36
  DISPLAY        104*  174
  DUMPOPTION      62

 -E-
  EM              13*  301   302
  EMPTY           26
  ENDFILE         44
  ENDMEDIUM       44
  ERROR          249*  335
```

```
-F-
 F              93*   94*   95*   96*   97*
 FAILURE        43
 FALSE         253   308   310   324
 FF             13*
 FILE           24*   93    94    95    96    97
 FILEATTR       28*
 FILEKIND       26*   29
 FOUND          93*
 FULLWORD       10*

-G-
 GET            95*

-H-
 HEADER        102*
 HEAPLIMIT      88   226

-I-
 I             155*  179*  181=  185=  185   185   191*  200=  204=  205   209*  211=  213
 ID             77    93*  121*  208*  213   218*  221=  222=  223=  224=  225=  226=  227=  228=  229=  230=  235*  239
               264*  275=  278=  282   283   321*  325   329   338
 IDENTIFIER     22*   77    93   121   208   218   235   236   264   321
 IDENTIFY      102*  166
 IDLENGTH       21*   22   211   277
 IDPTR         266*  275=  277   278   278=  278
 IDTYPE         55    77*
 INITIALIZE    305*  367
 INITWRITE     164*  251   316
 INP            84
 INPUT          38
 INPUTTASK      52
 INT            76   189*  193   202   314=  333   350=
 INTEGER        10    11*   30    32    57    66    76    95    96    97    99   100   122   124   152   155   179   189
               191   209   235   266   321   322   344   346
 INTERFACE      69*
 INTERVENTION   43
 INTTYPE        55    76*  314   349
 IOARG          40*   49
 IODEVICE       35*
 IOOPERATION    38*   47
 IOPARAM        46*
 IORESULT       42*   48

-J-
 JOBTASK        52

-K-
 KIND           29*

-L-
 L             346*  353
 LABELS         66*
 LENGTH         97*  189*  200
 LINE           19*  102   122*  124*  126   177   235*  241   249   322*  325   329
 LINELENGTH     18*   19   159
 LIST          157*  309   311   313   324   325   332   333   349   350   353   370
 LISTOPTION     61
```

```
-M-
 MAIN          154   167   172
 MARK           99*
 MAXARG         81*   82
 MOVE           38

-N-
 N             266*  284=  284   287=  289=  289   291
 NEXTCHAR      256*  259=  273   278   279   289   293
 NILTYPE        55    75
 NL             13*  238   245   272   293   307
 NOSOURCE      161*  282=  308=  368
 NOTUSED        32*  106*
 NOTUSED10     119*
 NOTUSED2      107*
 NOTUSED3      108*
 NOTUSED4      109*
 NOTUSED5      110*
 NOTUSED6      111*
 NOTUSED7      113*
 NOTUSED8      115*
 NOTUUED9      117*
 NOWHERE       153
 NUL           178*  182
 NUMBER        190*  196=  205
 NUMBEROPTION   62
 NUMERICS      263*  270=  270=  270   276   286   288

-O-
 OBJECT        151*
 OK            151*  246=  253=  332=  334   336   371   374   375   378   379   382   383   386   387   390   391   394
               395
 ONDISK        153
 OPEN           93*
 OPERATION      47*
 OPTION         61*   65
 OPTIONS        65*
 ORD           197   289   289
 OUT            84
 OUTP          154   328
 OUTPUT         38
 OUTPUTTASK     52
 OVERFLOW       87   222

-P-
 P              95*   96*  126*
 PAGE           16*   95    96
 PAGELENGTH     15*   16
 PARAM         121*  126*  259   271
 PARAMPTR      159*  258=  258   259   271=
 PASS          344*  350   351   352   354
 PASSES        147*  160
 PASSLINK       59    64*
 PASSNO        321*
 PASSPTR        59*   78
 PAST_EOM      151*  301   370=
 POINTER        57*   67    68    69
 POINTERERROR   87   223
 PRINTDEVICE    36
 PROGRESULT     86*  122   218   235   322   347
 PROTECTED       31*
 PTR             78   312=
 PTRTYPE         55    78*  312
 PUT             96*

-R-
 R              347*  353
 RANGEERROR      87   224
 READ            90*  301
 RELEASE        100*
 REM            191*  193=  197   198=  198   199
 REPORT         154*  167=  172   328=
 RESETPOINT      67*
 RESULT         122*  218*  220   235*  243   246   322*  325   326   329
 REWIND          40
 RUN            121*  325   353
 RUN_MNEM       344*  372   376   380   384   388   392   396

-S-
 SCANPARMS      262*  315
 SCRATCH         26
 SEQCODE         26
 SHORTINTEGER    11
 SKIPS          263*  268=  273
 SOURCE         151*
 STACKLIMIT      88   227
 STARTMEDIUM     44
 STATUS          48*
 SUMMARYOPTIO    61

-T-
 TABLES          68*
 TAG             74*  310=  312=  314=  349=
 TAPEDEVICE      36
 TASKKIND        52*
 TERMINATED      87   221   246   326
 TESTOPTION      61
 TESTPASS       160*  284=  284   291=  291   308=  371   375   379   383   387   391   395
 TEXT           177*  181   185   249*  252
 TIMELIMIT       88   229
 TOP             99*  100*
 TRANSMISSION    43
 TRUE           282
 TYPEDEVICE      36

-U-
 UPSPACE         40

-V-
 VARIANTERROR    87   225

-W-
 WHERE          153*
 WRITE           91*  173   302
 WRITECHAR      170*  184   201   202   203   205   214   238   242   245   307
 WRITEEOF        40
 WRITEID        208*  239   244   338
 WRITEINT       189*  241
 WRITERESULT    234*  329
 WRITETEXT      177*  240   252   317   338   355   356   357   358   359   360   361
 -X-
  XJP_OFFSETS     66*
  XREFOPTION      62

END XREF  191 IDENTIFIERS   605 TOTAL REFERENCES
  25 COLLISIONS.
```

APPENDIX D

SOURCE CODE CHANGES TO MCPASCAL PASSES 1-5

```
CONST IDLENGTH = 12;
TYPE IDENTIFIER = ARRAY (.1..IDLENGTH.) OF CHAR;
TYPE LINE = ARRAY(.1..132.) OF CHAR;
TYPE   POINTER = @ INTEGER;
OPTION = LISTOPTION..VARNTCHECKOPTION;              "VARNT*"
PASSPTR = @PASSLINK;
PASSLINK =
  RECORD
   OPTIONS: SET OF OPTION;
              "MICROENGINE"
    LABELS, BLOCKS, CONSTANTS, XJP_OFFSETS: INTEGER;
    RESETPOINT: FULLWORD;
    TABLES: POINTER;
    INTERFACE: POINTER    "MICROENGINE"
  END;
```

--- PASS 1 CODE MODIFICATIONS ---

```
CONST IDLENGTH = 12;
TYPE IDENTIFIER = ARRAY (.1..IDLENGTH.) OF CHAR;

TYPE   POINTER = @ INTEGER;
OPTION = LISTOPTION..VARNTCHECKOPTION;         "VARNT*"
PASSPTR = @PASSLINK;
PASSLINK =
  RECORD
    OPTIONS: SET OF OPTION;
           "MICROENGINE"
    LABELS, BLOCKS, CONSTANTS, XJP_OFFSETS: INTEGER;
    RESETPOINT: FULLWORD;
    TABLES: POINTER;
    INTERFACE: POINTER    "MICROENGINE"
  END;
```

--- PASS 2 CODE MODIFICATIONS ---

```
 35     CONST IDLENGTH = 12;
 36     TYPE IDENTIFIER = ARRAY (.1..IDLENGTH.) OF CHAR;
 37
 38     TYPE   POINTER = @ INTEGER;
 39     OPTION = LISTOPTION..VARNTCHECKOPTION;          "VARNT*"
 40     PASSPTR = @PASSLINK;
 41     PASSLINK =
 42       RECORD
 43         OPTIONS: SET OF OPTION;
                        "MICROENGINE"
 44         LABELS, BLOCKS, CONSTANTS, XJP_OFFSETS: INTEGER;
 45         RESETPOINT: FULLWORD;
 46         TABLES: POINTER;
 47         INTERFACE: POINTER   "MICROENGINE"
 48       END;
```

```
145     SUB2=81;         INDEX2=82;       REAL2=83;        STRING2=84;
146     LCONST2=85;      MESSAGE2=86;     NEW_LINE2=87;    FWD_DEF2=88;
147     CHK_TYPE2=89;    PROCF_DEF2=90;   UNDEF2=91;       PEND2=92;
148     CASE_JUMP2=93;
149     "MANAGER:"                                      "MGR*"
150     MANAGER2=94;        FROM2=95;          REFERS2=96;
151     NIL2=97;                                       "MGR*"
152     VTAG_DEF2=98;    VPART_END2=99;   VVARNT_END2=100;   "VARNT*"
153     VVARIANT2=101;   "VARNT*"         DUPTOS2=102; "MICROENGINE"
154     "DUPTOS2 DUPLICATES TOP-OF-STACK WORD"
155
156     "OTHER CONSTANTS"
157
158     SPLIT_SET_LENGTH = "%IF SRC_16" 8; "%END" "WORDS/SPLIT SET"
159                        "%IF SRC_32" 4; "%END"
160     NOUN_MAX=999;
```

```
1881     FUNCTION1: FUNCTION_;
1882     GE1: BINARY(GE2);
1883     GT1: BINARY(GT2);
1884     INCLUDE1: BINARY(INCLUDE2);
1885     INIT_NAME1:BEGIN INIT_NAME; PUTO(DUPTOS2) END;"MICROENGINE"
1886     INITS_DEF1: INITS_DEF;
1887     INITS_END1: POP_LEVEL;
1888     INIT1: CALL(INIT2);
1889     INTEGER1: INDEX(XINTEGER);
1890     INTF_ID1: INTF_ID;
```

--- PASS 3 CODE MODIFICATIONS ---

```
38      CONST IDLENGTH = 12;
39      TYPE IDENTIFIER = ARRAY (.1..IDLENGTH.) OF CHAR;
40
41      TYPE   POINTER = @ INTEGER;
42      OPTION = LISTOPTION..VARNTCHECKOPTION;          "VARNT*"
43      PASSPTR = @PASSLINK;
44      PASSLINK =
45        RECORD
46          OPTIONS: SET OF OPTION;
                    "MICROENGINE"
47          LABELS, BLOCKS, CONSTANTS, XJP_OFFSETS: INTEGER;
48          RESETPOINT: FULLWORD;
49          TABLES: POINTER;
50          INTERFACE: POINTER     "MICROENGINE"
51        END;
```

```
130     CASE_ JUMP1=93;
131     MANAGER1=94;      FROM1=95;       REFERS1=96;      "MGR*"
132     NIL1=97;                                   "MGR*"
133     VTAG_DEF1=98; VPART_END1=99; VVARNT_END1=100; VVARIANT1=101;
134     DUPTOS1 = 102;   "MICROENGINE"
135
136     "OUTPUT OPERATORS"
137
138     EOM2=1;           BODY2=2;          BODY_END2=3;     ADDRESS2=4;
139     RESULT2=5;        STORE2=6;         CALL_PROC2=7;    CONSTPARM2=8;
```

```
150     VCOMP2=49;        RCOMP2=50;        SUB2=51;         LCONST2=52;
151     MESSAGE2=53;      NEW_LINE2=54;     CHK_TYPE2=55;    SAVEPARM2=56;
152     CALL_GEN2=57;     NOT2=58;          UNDEF2=59;       RANGE2=60;
153     REFERS2=61;          "MGR*"
154     VVARIANT2=62;     "VARNT*"             DUPTOS2=63;"MICROENGINE"
155
156     "STANDARD SPELLING/NOUN INDICES"
157
158     XUNDEF=0;         XFALSE=1;         XTRUE=2;         XINTEGER=3;
159     XBOOLEAN=4;       XCHAR=5;          XQUEUE=6;        XABS=7;
```

--- PASS 4 CODE MODIFICATIONS ---

```
PROCEDURE BODY;
BEGIN
  WITH STACK(.T.)@ DO BEGIN
    VAR_SIZE:=CURRENT_DISP;
    IF INITIAL_ENTRY THEN BEGIN
      INITIAL_ENTRY:=FALSE;
      COMPVAR_LENGTH:=CURRENT_DISP "SAVE COMP VAR LENGTH;
      CURRENT_DISP:=0 "INITIAL STMNT IS VARIABLE-LESS";
      "REMOVED FOR MICROENGINE"
      "PUT5(BODY2,RMODE,RDISP,0,0,STACK_SIZE)"
    END;   "ELSE"  "REMOVED FOR MICROENGINE"
    PUT5(BODY2,RMODE,RDISP,PARM_SIZE,VAR_SIZE,STACK_SIZE)
  END
END;
```

```
CHK_TYPE1: BEGIN PUTO(CHK_TYPE2); PUT_TYPE END;
CLASS1: COMP_DEF(PACKED_CLASS);
CPARMLIST1: CPARM_LIST;
DEF_LABEL1: IGNORE1(DEF_LABEL2);
DIV1: PUTO(DIV2);
DUPTOS1: PUTO(DUPTOS2);   "MICROENGINE"
EMPTY_SET1: PUTO(EMPTY_SET2);
EOM1: EOM;
ENUM_DEF1: ENUM_DEF;
EQ1: PUTO(EQ2);
```

```
"************** MICROENGINE ********************"
CONST MAXINTFAC = 14; "14 INTERFACE SEGMENTS MAX."
TYPE
   IFPTR = @IFINFO;   "INTERFACE INFORMATION POINTER"
   IFINFO = RECORD    "INTERFACE INFORMATION RECORD"
      INTERFACES: INTEGER; "NO. OF INTERFACE SEGMENTS"
      INTERFACESIZES: "PROCESS ENTRY RTNS IN EACH INTERFACE"
         ARRAY[1..MAXINTFAC] OF INTEGER;
      END;
"***************************************************"

PASSPTR = @PASSLINK;
PASSLINK =
  RECORD
    OPTIONS: SET OF OPTION;
             "MICROENGINE"
    LABELS, BLOCKS, CONSTANTS, XJP_OFFSETS: INTEGER;
    RESETPOINT: FULLWORD;
    TABLES: POINTER;
    INTERFACE: IFPTR      "MICROENGINE"
  END;
```

```
VCOMP1=49;      RCOMP1=50;        SUB1=51;          LCONST1=52;
MESSAGE1=53;    NEW_LINE1=54;     CHK_TYPE1=55;     SAVEPARM1=56;
CALL_GEN1=57;   NOT1=58;          UNDEF1=59;        RANGE1=60;
REFERS1=61;                         "MGR*"
VVARIANT1=62;   "VARNT*"              DUPTOS1=63;   "MICROENGINE"

"OUTPUT OPERATORS"

PUSHCONST2=0;   PUSHVAR2=1;       PUSHIND2=2;       PUSHADDR2=3;
FIELD2=4;       INDEX2=5;         POINTER2=6;       VARIANT2=7;
```

```
CASEJUMP2=32;   INITVAR2=33;      CALL2=34;         ENTER2=35;
RETURN2=36;     POP2=37;          NEWLINE2=38;      ERR2=39;
LCONST2=40;     MESSAGE2=41;      INCREMENT2=42; DECREMENT2=43;
PROCEDURE2=44;  INIT2=45;         PUSHLABEL2=46;    CALLPROG2=47;
EOM2=48;            DUPTOS2=49;            "MICROENGINE"

"CONTEXT"

FUNC_RESULT=1;  ENTRY_VAR=2;      VARIABLE=3;       VAR_PARM=4;
UNIV_VAR=5;     CONST_PARM=6;     UNIV_CONST=7;     FIELD=8;
```

--- PASS 5 CODE MODIFICATIONS ---

```

  VARNT_TAGSET, VARNT_DISP: INTEGER;      "VARNT*"

  INTFACEPTR: IFPTR;     "MICROENGINE"
  INTFACE: IFINFO;       "MICROENGINE"


"############################"
"COMMON TEST OUTPUT MECHANISM"
"############################"
```

```
PROCEDURE NEXT_PASS (VAR LINK: PASSPTR);
BEGIN
  LINK@.INTERFACE := INTFACEPTR;     "MICROENGINE"
  IF WORDS_OUT > 0
  THEN IF PAGES_OUT > FILE_LENGTH(OUTFILE)
       THEN FILE_LIMIT
       ELSE PUT(OUTFILE, PAGES_OUT, PAGE_OUT);
  WITH PARAM(.1.) DO
    BEGIN TAG:= BOOLTYPE; BOOL:=OK END;
  WITH PARAM(.2.) DO
    BEGIN TAG:= PTRTYPE; PTR:= LINK END;
  WITH PARAM(.4.) DO
    BEGIN TAG:= INTTYPE;  INT:= PAGES_OUT  END;
END;
```

```
  PROCEDURE INITIALIZE;
  BEGIN
    DONE:=FALSE;
    ACTIVE_VARNT:= FALSE;                   "VARNT*"
    INIT_PASS(INTER_PASS_PTR);
    WITH INTER_PASS_PTR@ DO BEGIN
      DEBUG:=TESTOPTION IN OPTIONS;
      IF DEBUG THEN PRINTFF;
      XJP_OFFSETS := 0     "MICROENGINE"
    END;

"****************** MICROENGINE *********************"
    NEW(INTFACEPTR); "GET SPACE FOR INTERFACE INFO RECORD"
    MARK(INTER_PASS_PTR@.RESETPOINT); "GET @ NEW HEAPTOP"
    INTFACEPTR@.INTERFACES := 0; "NO INTERFACES DEFINED YET"
"*************************************************"

    ARITHMETIC:=(.INT_KIND,REAL_KIND.);
    INDEXS:=(.INT_KIND,BOOL_KIND,CHAR_KIND,ENUM_KIND.);
```

```
PROCEDURE ADDRESS;
BEGIN
  WITH T@ DO
    IF CLAS=VALUE THEN BEGIN
      CASE STATE OF
        DIRECT: BEGIN
          IF MODE=SCONST_MODE THEN ADDR_ERROR
          ELSE PUT2(PUSHADDR2,MODE,DISP);
          "NEXT 'IF...THEN' DELETED FOR MICROENGINE"
          "IF (KIND=SYSCOMP_KIND) & (CONTEXT<> FROM_VAR)"
            "THEN PUT1(FIELD2,LENGTH)" "OFFSET"
        END;
        INDIRECT: PUT3(PUSHVAR2,WORD_TYP,MODE,DISP);
        ADDR: ;
        EXPRESSION: ADDR_ERROR
      END;
      STATE:=ADDR
    END ELSE ADDR_ERROR
END;
```

```
PROCEDURE CASE_LIST;
VAR I,MIN,MAX:INTEGER; L:DISPLACEMENT;
BEGIN
  POP "SELECTOR";
  DEF_LABEL;
  READ_IFL(MIN); READ_IFL(MAX); PUT2(CASEJUMP2,MIN,MAX);

  "MICROENGINE"
  "COUNT HOW MANY BYTES THIS CASE JUMP WILL ADD TO CNSTNT
   BLOCK. MIN, MAX, OFFSET FOR EACH CASE, ARE 1 WORD EACH"
  WITH INTER_PASS_PTR@ DO
     XJP_OFFSETS:=XJP_OFFSETS+((MAX-MIN+1)+2)*WORDLENGTH;

  FOR I:=MIN TO MAX DO BEGIN
    READ_IFL(L); PUT_ARG(L)
  END;
  DEF_LABEL
END;
```

--- PASS 5 CODE MODIFICATIONS ---

```
  PROCEDURE PROG_CALL;
  VAR INTF_LENGTH:INTEGER;
  BEGIN
    READ_IFL(INTF_LENGTH);

"******************* MICROENGINE *******************"
    IF INTF_LENGTH <> 0
    THEN WITH INTFACEPTR@ DO
       BEGIN
       INTERFACES := SUCC(INTERFACES); "FOUND AN INTERFACE"
       "SAVE NUMBER OF ENTRY POINTS FOR THIS INTERFACE"
       INTERFACESIZES[INTERFACES] := INTF_LENGTH DIV
                                                 WORDLENGTH;
       END;
"************************************************"

    PUTO(CALLPROG2);  PUT1(POP2,INTF_LENGTH);
    POP
  END;
```

```
  PROCEDURE SUB;
  VAR MIN,MAX,SIZE: INTEGER;
  BEGIN
    "SUBSCRIPT" VALUE_;
    READ_IFL(MIN); READ_IFL(MAX); READ_IFL(SIZE);

"*************** MICROENGINE ********************"
    "INDEX OPERATOR CHANGED TO ACCOMODATE MICROENGINE
     BYTE POINTERS. TYPE ARGUMENT (LAST ARGUMENT) WAS ADDED
     SO PASS 6 KNOWS TO GEN BYTE POINTERS (2 WORDS ON STACK)
     INSTEAD OF WORD POINTERS (1 WORD ON STACK). "
    PUT4(INDEX2,MIN,MAX,SIZE,T@.KIND);
"************************************************"

    PUSH; T@:=UNDEF_EXPR; "INDEX" TYPE_;
    IF COMPATIBLE THEN "OK";
    POP; POP;
    "ELEMENT" TYPE_;
    WITH T@ DO
      IF KIND=SYSCOMP_KIND THEN PUT1(FIELD2,LENGTH) "OFFSET"
  END;
```

```
CASE_LIST1: CASE_LIST;
CHK_TYPE1: CHK_TYPE;
CONSTPARM1: CONSTPARM(FALSE);
DEF_LABEL1: DEF_LABEL;
DIV1: DIV_MOD(DIV2);
DUPTOS1: PUTO(DUPTOS2);      "MICROENGINE"
EMPTY_SET1: EMPTY_SET;
EOM1: EOM;
EQ1: EQUALITY(EQUAL);
FALSEJUMP1: FALSE_JUMP;
```

ON THE ADAPTABILITY OF MULTIPASS PASCAL COMPILERS TO VARIANTS OF (PASCAL) P-CODE MACHINE ARCHITECTURES

by

MARK A. LITTEKEN

B. S., Pittsburg State University, Pittsburg, Kansas, 1979

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1981

ABSTRACT

This report is a description of an effort to modify the code-generation portion of a multipass Concurrent Pascal (CPascal) compiler so that it will produce object code for a different machine. The original compiler generated P-code (Pascal stack machine code) for a virtual machine. The modified version will generate code for the Pascal Microengine, a microcomputer whose instruction set is similar to virtual P-code. The similarity of instruction sets made it appear that the required modifications would be straightforward, and that the high-level language constructs of Concurrent Pascal (monitors, for example) would easily map onto the Microengine's architecture. However, as the project progressed the architectural differences became a significant obstacle. This report contains a description of the organization and architecture of the two machines and detail on the changes made to the original Concurrent Pascal compiler. It also contains a description of the compilation problems caused by the differences between the machines and how those difficulties were resolved.