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:
SPEC
COLL
LD
2668
R4
1981
L57
c. 2
# 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
## LIST OF FIGURES

1. Concurrent Pascal Compiler Geneoloy .................................. 5
2. Stack and Registers After Virtual Machine Function Call .... 9
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
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 routines are mapped to 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 GENEEOLOGY

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

\[ \text{MOV } @\text{(Q)+, P} \]

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
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.
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. PRIORITY (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 [MICH79]. 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 6. 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.
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, ENDPRECESS,
INITPROCESS, REALTIME, DELAYGATE, CONTGATE,
STOPJOB, WAIT, SYSERROR, IO);

SEGMENT PROCEDURE KCALLHDLR (OPTT: KERNOPTR;
   OPND1, OPND2, OPND3, OPND4: INTEGER);
CONST
  IGATERTN = ;
  EGATERTN = ;
  LGATERTN = ;
  EPROCRTN = ;
  IPROCRTN = ;
  RTIMERTN = ;
  DGATERTN = ;
  CGATERTN = ;
  STJOBRTN = ;
  WAITRTN = ;
  SYSEERRTN = ;
  IORTN = ;

KERNSEG = 0; "SEGMENT NUMBER OF KERNEL PROPER"
BEGIN
CASE OPTT 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;
  ENDPRECESS: PUSH OPERANDS; EXTERNAL CALL TO KERNSEG, 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;
  SYSEERROR: PUSH OPERANDS; EXTERNAL CALL TO KERNSEG, SYSEERRTN;
  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.
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 guaranteed 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 corresponding 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 15. Concurrent Pascal program on the Pascal Microengine.
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

```pascal
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

```pascal
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
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—these 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.
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;

3) 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 PROGRESS 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 compatibility 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 DELABEL1 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 DELABEL1( <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 DELABEL1 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 increased 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 $100\cdot[N/100]$ 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 WRITEM 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, PROGLNGTH 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 labels 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:

<table>
<thead>
<tr>
<th>Command File:</th>
<th>MCPASCAL .CSS</th>
<th>MEPASCAL .CSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver:</td>
<td>MCPASCAL</td>
<td>MEPASCAL</td>
</tr>
<tr>
<td>Compiler Passes:</td>
<td>MCPASSx</td>
<td>MEPASSx</td>
</tr>
<tr>
<td>Pass 1, no source:</td>
<td>MCPASS1N</td>
<td>MEPASS1N</td>
</tr>
<tr>
<td>Code Mnemonics:</td>
<td>MNEM</td>
<td>MNEM</td>
</tr>
</tbody>
</table>

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 JJP (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 JJP_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, DUITOS2, (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. DUITOS2 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 DUITOS1 and DUITOS2. DUITOS1 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 PASSLINKINTERFACE is of type POINTER merely to reserve a fullword (32 bits) of storage, the amount of space required for any pointer. In pass five PASSLINKINTERFACE 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). InterfaceINTERFACEPTR and INTERFACE 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 (INTERFACEPTR) 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 MEGASCA can be found in [MICR79, REG79] 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 and 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 constant 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
* The number of interfaces can be zero, leaving TEMP FILE 4 empty.
FIGURE 27. Structures left in the heap by MEPASS6. Although 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.

MEEPASS6 leaves five tables in the heap: Jumptable, Consttable, Xjptable, Exictable, and Datatable. 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. Exictable 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 Datatable, and the entries are the DATASIZE values for the routines (see section 3.2.2).

In the conversion of MCPASS6 to MEEPASS6 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, code-length, 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 during 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; much 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 ahead of 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).

EXITCTABLE 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 concurrent 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 operator. Length is always one byte;
- **UB**—An **Unsigned Byte** operand. Length is always one byte;
- **SB**—A **Signed Byte** operand. Length is always one byte;
- **DB**—A "**Don't care** Byte" operand. Length is always one byte;
- **B**—A "**Big**" operand. Length is one or two bytes, depending on the value;
- **W**—A **Word** operand. Length is always two bytes;
- **NOTME**—The output data item is **NOT** a MicroEngine 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 WR ITEARG 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 UDLOC. 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 (DISPLacemnet) 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 MEGASS6 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 variables is the same as for class and monitor ENTRY routines. Parameters, 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 but the stack marker in that position, and parameter displacements could be calculated as

\[(\text{BYTE OFFSET} + \text{TWOWORDS} + \text{FOURWORDS} + \text{ONETHEMATICAL}) \div \text{WORDLENGTH}\]

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

\[ \text{LOCATION} + (2 \times \text{Routines}) \].

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_SAVEBASE 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 COMPSTRUCT1 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
<table>
<thead>
<tr>
<th>VIRTUAL OPERATOR</th>
<th>RESULT OR ACTION</th>
<th>MICROENGINE INSTRUCTION(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSWORD</td>
<td>absolute value of TOS</td>
<td>ABI</td>
</tr>
<tr>
<td>ADDWORD</td>
<td>sum of TOS, TOS-1</td>
<td>ADI</td>
</tr>
<tr>
<td>ANDWORD</td>
<td>logical AND of TOS, TOS-1</td>
<td>LAND</td>
</tr>
<tr>
<td>CONVWORD</td>
<td>real-value equivalent of TOS</td>
<td>FLT</td>
</tr>
<tr>
<td>COPYWORD</td>
<td>store TOS indirect through TOS-1</td>
<td>STO</td>
</tr>
<tr>
<td>DECRWORD</td>
<td>decrement word indirect through TOS-1</td>
<td>DUP1, LDM (1), SLDCL01, SBI, STO</td>
</tr>
<tr>
<td>DIVWORD</td>
<td>integer quotient of TOS-1/TOS</td>
<td>DVI</td>
</tr>
<tr>
<td>EQWORD</td>
<td>boolean value of (TOS-1 = TOS)</td>
<td>EQUI</td>
</tr>
<tr>
<td>GHWORD</td>
<td>boolean value of (TOS-1 &gt; TOS)</td>
<td>LEQI, LNOT</td>
</tr>
<tr>
<td>INCRWORD</td>
<td>increment word indirect through TOS-1</td>
<td>DUP1, LDM (1), SLDCL01, ADI, STO</td>
</tr>
</tbody>
</table>

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 the stack. Instructions marked '*' do not push a result.
<table>
<thead>
<tr>
<th>VIRTUAL OPERATOR</th>
<th>RESULT OR ACTION</th>
<th>MICROENGINE INSTRUCTION(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LESSWORD</td>
<td>boolean value of (TOS-1 &lt; TOS)</td>
<td>GEQI</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LNOT</td>
</tr>
<tr>
<td>MODWORD</td>
<td>TOS-1 MOD TOS</td>
<td>MODI</td>
</tr>
<tr>
<td>MULWORD</td>
<td>product of TOS, TOS-1</td>
<td>MPI</td>
</tr>
<tr>
<td>NEGWORD</td>
<td>2's complement of TOS</td>
<td>NGI</td>
</tr>
<tr>
<td>NEWORD</td>
<td>boolean value of (TOS-1 &lt;&gt; TOS)</td>
<td>NEQI</td>
</tr>
<tr>
<td>NGWORD</td>
<td>boolean value of (TOS-1 &lt;= TOS)</td>
<td>LEQI</td>
</tr>
<tr>
<td>NLWORD</td>
<td>boolean value of (TOS-1 &gt;= TOS)</td>
<td>GEQI</td>
</tr>
<tr>
<td>ORWORD</td>
<td>logical OR of TOS, TOS-1</td>
<td>LOR</td>
</tr>
<tr>
<td>PREDWORD</td>
<td>TOS minus 1</td>
<td>SLDC01</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SBI</td>
</tr>
<tr>
<td>SUBWORD</td>
<td>TOS-1 minus TOS</td>
<td>SBI</td>
</tr>
<tr>
<td>SUCCEED</td>
<td>TOS plus 1</td>
<td>SLDC01</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADI</td>
</tr>
</tbody>
</table>

FIGURE 30. (continued from previous page)
<table>
<thead>
<tr>
<th>VIRTUAL OPERATOR</th>
<th>RESULT OR ACTION</th>
<th>MICROENGINE INSTRUCTION(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSREAL</td>
<td>absolute value of TOS</td>
<td>ABR</td>
</tr>
<tr>
<td>ADDREAL</td>
<td>sum of TOS, TOS-1</td>
<td>ADR</td>
</tr>
<tr>
<td>COPYREAL</td>
<td>store TOS indirect through TOS-1 *</td>
<td>STM (2)</td>
</tr>
<tr>
<td>DIVREAL</td>
<td>quotient of TOS-1/TOS</td>
<td>DVR</td>
</tr>
<tr>
<td>EQREAL</td>
<td>boolean of (TOS-1 = TOS)</td>
<td>EQUREAL</td>
</tr>
<tr>
<td>GRREAL</td>
<td>boolean of (TOS-1 &gt; TOS)</td>
<td>LEQREAL LNOT</td>
</tr>
<tr>
<td>LSREAL</td>
<td>boolean of (TOS-1 &lt; TOS)</td>
<td>GEQREAL LNOT</td>
</tr>
<tr>
<td>MULREAL</td>
<td>product of TOS-1, TOS</td>
<td>MPR</td>
</tr>
<tr>
<td>NEGREAL</td>
<td>negation of TOS</td>
<td>NGR</td>
</tr>
<tr>
<td>NEREAL</td>
<td>boolean of (TOS-1 &lt;&gt; TOS)</td>
<td>EQUREAL LNOT</td>
</tr>
<tr>
<td>NGREAL</td>
<td>boolean of (TOS-1 &lt;= TOS)</td>
<td>LEQREAL</td>
</tr>
<tr>
<td>NLREAL</td>
<td>boolean of (TOS-1 &gt;= TOS)</td>
<td>GEQREAL</td>
</tr>
<tr>
<td>SUBREAL</td>
<td>TOS-1 minus TOS</td>
<td>SBR</td>
</tr>
<tr>
<td>TRUNCREAL</td>
<td>TOS, truncated and converted to integer</td>
<td>TNC</td>
</tr>
</tbody>
</table>

**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.
<table>
<thead>
<tr>
<th>VIRTUAL OPERATOR</th>
<th>RESULT OR ACTION</th>
<th>MICROENGINE INSTRUCTION(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ANDSET</strong></td>
<td>intersection of TOS, TOS-1</td>
<td>INT</td>
</tr>
<tr>
<td><strong>BUILDSET</strong></td>
<td>TOS set with a new member; New member number is given by integer TOS.</td>
<td>SLDC00, LDCB (127), CHK, DUP, SRS, UNI</td>
</tr>
<tr>
<td><strong>COPYSET</strong></td>
<td>store TOS set indirect through TOS-1 *</td>
<td>FJP (0), STM (8)</td>
</tr>
<tr>
<td><strong>EQSET</strong></td>
<td>boolean value of set compare (TOS-1 = TOS)</td>
<td>EQUFWR</td>
</tr>
<tr>
<td><strong>INSET</strong></td>
<td>boolean value of test for inclusion of TOS-1 member number (integer) in TOS set</td>
<td>INW</td>
</tr>
<tr>
<td><strong>NESET</strong></td>
<td>boolean value of set comparison (TOS-1 &lt;&gt; TOS)</td>
<td>EQUFWR, LNOT</td>
</tr>
<tr>
<td><strong>NGSET</strong></td>
<td>boolean value of subset test (TOS-1 &lt;= TOS)</td>
<td>LEQPW</td>
</tr>
<tr>
<td><strong>NLSET</strong></td>
<td>boolean value of superset test (TOS-1 &gt;= TOS)</td>
<td>GEQPW</td>
</tr>
<tr>
<td><strong>ORSET</strong></td>
<td>union of TOS, TOS-1 sets</td>
<td>UNI</td>
</tr>
<tr>
<td><strong>SUBSET</strong></td>
<td>TOS-1 set less members of TOS-1 set</td>
<td>DIF</td>
</tr>
</tbody>
</table>

| **COPYSTRUC**    | move TOS® to TOS-1® * | MOV (size) |
| **EQSTRUC**      | boolean value of (TOS-1® = TOS®) | EQUFW (size) |
| **CRSTRUC**      | boolean value of (TOS-1® > TOS®) | LEQFW (size), LNOT |
| **LSSTRUC**      | boolean value of (TOS-1® < TOS®) | GEQFW (size), LNOT |
| **NESTRUC**      | boolean value of (TOS-1® <> TOS®) | EQUFW (size), LNOT |
| **NGSTRUC**      | boolean value of (TOS-1® <= TOS®) | LEQFW (size) |
| **NLSTRUC**      | boolean value of (TOS-1® >= TOS®) | GEQFW (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 EQU 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 heap top 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:

<table>
<thead>
<tr>
<th>COPYTAG</th>
<th>EOM</th>
<th>FALSEJUMP</th>
<th>FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCVALUE</td>
<td>INITVAR</td>
<td>IO</td>
<td>JUMP</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>NEWLINE</td>
<td>NOT</td>
<td>POINTER</td>
</tr>
<tr>
<td>POP</td>
<td>RANGE</td>
<td>REALTIME</td>
<td>VARIANT</td>
</tr>
</tbody>
</table>

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 heap top
SPR            set heap pointer to new heap top

next: SLDC02
LPR            push new heap top
GEQI           object scan pointer >= new heap top?
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 (always one word long) for which access is being requested. The tag field contains the bit number 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 does not alter the stack, leaving the record address in place. Use of the bit number for the
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"
EPJ (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 accessible (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 equivalent 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 NEWLIN2 virtual instruction into the code stream. This is a crutch for the intermediate code mnemonics program MEMNEM. NEWLIN2 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:

<table>
<thead>
<tr>
<th>WORD-TYPE</th>
<th>REAL-TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLDC00</td>
<td>SLDC00</td>
</tr>
<tr>
<td>SWAP</td>
<td>SWAP</td>
</tr>
</tbody>
</table>

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 [REG78]. 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 MCPPASS5, 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 length 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               add base address and offset,
                   yielding element address,
```

where the IXA (Index Array) is not 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 follows the code for the cases. This makes for a good deal of wild branching during execution, but it also guarantees that pass six 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 BEGIN
0004 CASE SELCTR OF
0005  3:  B := 2;
0006  8, 11:  C := 3;
0007  9:  D := 4;
0008  12: BEGIN
0009  15:  E := 5;
0010 END;
0011  6:  F := 6
0012 END;
0013 F := 200;
0014 END.

77777777777777777777777777777777777777777777
77777777777777777777777777777777777777777777
JUMP(2)  BEGNPRCS(4)
LINE  5  GLOELADD(-2)  PUSHCONS(100)  COPYWORD
LINE  6  PUSHGGLB(-14)  JUMP(138)
LINE  7  GLOELADD(-4)  PUSHCONS(2)  COPYWORD  JUMP(146)
LINE  8  GLOELADD(-6)  PUSHCONS(3)  COPYWORD  JUMP(124)
LINE  9  GLOELADD(-8)  PUSHCONS(4)  COPYWORD  JUMP(102)
LINE 10  GLOELADD(-10)  PUSHCONS(5)  COPYWORD
LINE 11  GLOELADD(-12)  PUSHCONS(5)  COPYWORD
LINE 12  JUMP(58)
LINE 13  GLOELADD(-12)
LINE 14  PUSHCONS(6)  COPYWORD  JUMP(32)
LINE 15  CASEJUMP(3, 9, -142, 18, 16, -38, 12, -130, -110, 6, -136, -94)LINE 15
LINE 16  GLOELADD(-12)  PUSHCONS(200)  COPYWORD
LINE 17  ENDPRECS

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 MEEPASS6) 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 before 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 minimum 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 instructions
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

\[\text{--- > varaddr --- > FIELD(disp) --- > arg list --- ~}
\]
\[
\text{------------------- ~}
\]
\[
\text{--- > INIT(mode, label, parm length, var length) --- > .}
\]

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

\[\text{--- > varaddr --- > FIELD(disp) --- > DUPTOS --- > arg list --- ~}
\]
\[
\text{------------------- ~}
\]
\[
\text{--- > INIT(mode, label, parm length, var length) --- > ,}
\]

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

\[
\text{STM (paramsize) pop "paramsize" words into class variable}
\]
\[
\text{CPL (routine no.) invoke initial routine of class.}
\]

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
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: SQL_CODE_TYPE);
ENTRY OPEN, CLOSE, GET, PUT, ACCEPT, DISPLAY;
```

then the sequential prefix must list those routines in the same order (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 MEGASS6 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 body. 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 outside 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 segment) 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, VARLENGTH 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 source 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 MEEPAS6 is modified to pass STACKTABLE on to MEETPASS 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 consistency—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 POLENGTH 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 POLENGTH 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 concurrent (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 names 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
SLD002     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 instructions 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 concurrent/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 SQLPGM, 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.

5555555555555555555555555555555555555555555555555555555555555555
5 MEXPASIAL INTERMEDIATE CODE PASS 5
5555555555555555555555555555555555555555555555555555555555555555
JUMP(1) LNOCONST(16,HEX-VAL:0000,0000,0000,0000,0000,0000,0000,0000,0000,0000)

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)
PUSHCKST(100) PUSHCKST(200) PUSHCKST(300)
PUSHADDR(6,-2006) CALLPROG POP(6)
LINE 17 RETURN(6)
LINE 21 ENTER(6,1,0,2,0)
LINE 22 PUSHADDR(6,-2) DUPITOS 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 MEXPASS5. 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
data size (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)  out: NOP
UJPL (out)            no operation, for word alignment

RFU (1)               return to caller (sequential program)
@ dat size
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:
   SLDDL01, XJP, NOP, and UJFL-- 7
8) code for each case:
   CXL and UJFL-- 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

\[ (2 + 2 + 2 + 2*RTNS + 2 + 2 + 7 + 6*RTNS + \\
3 + 2 + 1 + 1) \text{ DIV 2 "bytes per word"}) - 1, \]

or more concisely,

\[ (8*RTNS + 22) \text{ DIV 2}. \]

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 UIJPL after XJP) plus the size of the cases between the UIJPL 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:

\[
\text{KASE} \times 6 \text{ "bytes per case" } + 4.
\]

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 RFU instruction. By a calculation similar to the one above, the pointer value is

\[
\text{RTNS} \times 8 + 18.
\]

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 (SLLD01—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 from 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
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 consists 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, MESPASCAL 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 MEAPASCAL 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.


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.


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 label 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 and 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 straightforward, 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.
"HCPASCAL NUM - NEMPHIC TEST OUTPUT FORMATTING"

"I CONCURRENT := TRUE"

固定宽度字体

```pascal
VAR
   PAGELENGTH = 512 DIV 2;
   LINELENGTH = 70;
   IDLENGTH = 12;
   FILENAMES = ARRAY[1..5] OF STRING;
   FILEAT = RECORD
      KIND: STRING;
      AGD: INTEGER;
      PROT: BOOLEAN;
      NOTUSED: ARRAY[1..5] OF INTEGER;
   END;
   IOFILE = RECORD
      OPERATION: IOFILE;
      STATUS: IORESULT;
      ARG: IOARG;
   END;
   TASKFILE = RECORD
      IOPARAM: IOFILE;
      ARGPARAM: ARGPARAM;
      FILENAME: STRING;
   END;
   ARGTYPE = RECORD
      CARG TAG: ARGTAG OF
         NILTYPE, BOOLTYPE: (BOOL: BOOLEAN);
         INTTYPE: (INT: INTEGER);
      END;
END;
```
60 IDTYPE: (ID: IDENTIFIER);
61 PTRTYPE: (PTR: POINTER)
62 END;
63
64 CONST MAXARG = 10;
65 TYPE ARGLIST = ARRAY (...) OF ARGTYPE;
66
67 TYPE ARGSEQ = (INF, OUT);
68
69 TYPE PROGRESS =
70 (TERMINATED, OVERFLOW, POINTERERROR, RANGEERROR, VARIANTERROR,
71 HEADLIMIT, STACKLIMIT, CODELIMIT, TIMELIMIT, CALLERROR);
72
73 PROCEDURE READ(VAR C: CHAR);
74 PROCEDURE WRITE(VAR C: CHAR);
75
76 PROCEDURE OPEN(F: FILE; ID: IDENTIFIER; VAR FOUND: BOOLEAN);
77 PROCEDURE CLOSE(F: FILE);
78 PROCEDURE GET(F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);
79 PROCEDURE PUT(F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);
80 FUNCTION LENGTH(F: FILE; P: INTEGER);
81 PROCEDURE MARK(VAR TOP: INTEGER);
82 PROCEDURE RELEASE(TOP: INTEGER);
83 PROCEDURE IDENTIFY(HEADER: LINE);
84 PROCEDURE ACCEPT(VAR C: CHAR);
85 PROCEDURE DISPLAY(C: CHAR);
86
87 PROCEDURE HEADPAGE(VAR BLOCK: UNIV PAGE; VAR EOF: BOOLEAN);
88 PROCEDURE WRITEPAGE(BLOCK: UNIV PAGE; EOF: BOOLEAN);
89 PROCEDURE HEADLINE(VAR TEXT: UNIV LINE);
90 PROCEDURE WRITELINE(TEXT: UNIV LINE);
91 PROCEDURE READARG(5: ARGSEQ; ARG: ARGTYPE);
92 PROCEDURE WRITEARG(5: ARGSEQ; ARG: ARGTYPE);
93 PROCEDURE LOOKUP(ID: IDENTIFIER; VAR ATTR: FILEATTR;
94 VAR FOUND: BOOLEAN);
95
96 PROCEDURE IOTRANSFER
97 (DEVICE: IODEVICE; VAR PARAM: IOPARAM; VAR BLOCK: UNIV PAGE);
98 PROCEDURE IONSAVE(DEVICE: IODEVICE; VAR PARAM: IOPARAM);
99 PROCEDURE IONSAVE(DEVICE: IODEVICE; VAR PARAM: IOPARAM);
100 FUNCTION TASK: TASKID;
101 PROCEDURE RUN(ID: IDENTIFIER; VAR PARAM: ARGLIST;
102 VAR LINE: INTEGER; VAR RESULT: PROGRESS);
103
104 PROGRAM MAIN(VAR PARAM: ARGLIST);
105 "SIMPLE TYPES AND CONSTANTS"
106 CONST LINE_LENGTH = 70;
107 "OUTPUT DEV LINE LENGTH"
108
109 TYPE CHAR8 = ARRAY [1..8] OF CHAR;
110
111 TYPE CHAR = CHAR8;
112 const MIN_KIND = 0; MAX_KIND = 0;
VAR WORDS_IN: INTEGER;  "WORD IN INFL BUFFER"  # S
IN_FILE: INTEGER;  "INPUT FILE"  # S
PAGES_IN: INTEGER;  "CURRENT PAGE NUMBER"  # S
PAGE_IN: PAGE;  "INPUT BUFFER"  # S
PAGE_NO: INTEGER;  "COMPILER PAGE NUMBER"  # S
OUT_COL_PTR: INTEGER;  "POS ON OUTPUT LINE FOR BUF"  # S
OUT_BUF_PTR: INTEGER;  "POS IN BUFFER"  # S
OUT_BUF: STRING;  "OPERATOR OUTPUT BUFFER"  # S
FIRST_COL: INTEGER;  "FIRST COL TO USE FOR NON-WHM.IN"  # S
MIN_COL_SEP: INTEGER;  "MIN BLANKS BETWEEN OP"  # S
COL_SEP: INTEGER;  "SEP BETWEEN COL BOUNDARIES"  # S
END_OF: SHORTINTEGER;  "ROM OPERATOR"  # S
LINE_OP: SHORTINTEGER;  "LINE NUMBER OPERATOR"  # S
LCONST_OP: SHORTINTEGER;  "LCONST OPERATOR"  # S
OP: SHORTINTEGER;  "CURRENT INPUT OPERATOR"  # S
CODE_LENGTH: SHORTINTEGER;  # S
CODE_READ: SHORTWORD;  # S
I: INTEGER;  # S

PROCEDURE READ_INFL (VAR ARG: SHORTINTEGER);  # 2
BEGIN
IF WORDS_IN = PAGELNGTH THEN BEGIN
GET(IN_FILE,PAGES_IN,PAGE_IN);
PAGES_IN:=PAGES_IN+1; WORDS_IN:=0;
END;
END;

PROCEDURE NEXT_COL;  # S
VAR I, J: INTEGER;
BEGIN "PAD TO START OF NEXT OP COLUMN"
I:=FIRST_COL;
I:=(OUT_COL_PTR+MIN_COL_SEP-FIRST_COL-1)*COL_SEP;
BEGIN I=0 THEN I:=0;
I:=I-FIRST_COL;
BEGIN I=OUT_BUF_PTR-1 > LINE_LENGTH THEN BEGIN "WONT FIT"
WRITENL(); I:=FIRST_COL; OUT_COL_PTR:=1;
END;
FOR J:=OUT.COL_PTR TO I-1 DO WRITE(' '*);
END;
OUT.COL_PTR:=I;
END;
PROCEDURE WRITE_BUF (ALIGN: BOOLEAN);  # S
VAR I: INTEGER;  # S
BEGIN
IF ALIGN THEN NEXT_COL;
120 MIN_TPK = 0; MAX_TPK = 0;  #
121 MIN_NODE = 0; MAX_NODE = 0;  #
122 MIN_CONTEXT = 0; MAX_CONTEXT = 0;  #
123
124 "VARIABLES"
125
126 VAR WORDS_IN: INTEGER;  "WORD IN INFL BUFFER"  # S
127 IN_FILE: INTEGER;  "INPUT FILE"  # S
128 PAGES_IN: INTEGER;  "CURRENT PAGE NUMBER"  # S
129 PAGE_IN: PAGE;  "INPUT BUFFER"  # S
130 PAGE_NO: INTEGER;  "COMPILER PAGE NUMBER"  # S
131 OUT_COL_PTR: INTEGER;  "POS ON OUTPUT LINE FOR BUF"  # S
132 OUT_BUF_PTR: INTEGER;  "POS IN BUFFER"  # S
133 OUT_BUF: STRING;  "OPERATOR OUTPUT BUFFER"  # S
134 FIRST_COL: INTEGER;  "FIRST COL TO USE FOR NON-WHM.IN"  # S
135 MIN_COL_SEP: INTEGER;  "MIN BLANKS BETWEEN OP"  # S
136 COL_SEP: INTEGER;  "SEP BETWEEN COL BOUNDARIES"  # S
137 END_OF: SHORTINTEGER;  "ROM OPERATOR"  # S
138 LINE_OP: SHORTINTEGER;  "LINE NUMBER OPERATOR"  # S
139 LCONST_OP: SHORTINTEGER;  "LCONST OPERATOR"  # S
140 OP: SHORTINTEGER;  "CURRENT INPUT OPERATOR"  # S
141 CODE_LENGTH: SHORTINTEGER;  # S
142 CODE_READ: SHORTWORD;  # S
143 I: INTEGER;  # S
144
145 "INPUT FILE SUPPORT ROUTINES"
146
147 PROCEDURE READ_INFL (VAR ARG: SHORTINTEGER);  # 2
148 BEGIN
149 IF WORDS_IN = PAGELNGTH THEN BEGIN
150 GET(IN_FILE,PAGES_IN,PAGE_IN);
PAGES_IN:=PAGES_IN+1; WORDS_IN:=0;
END;
151 END;
152
153 PROCEDURE NEXT_COL;  # S
VAR I, J: INTEGER;
BEGIN "PAD TO START OF NEXT OP COLUMN"
I:=FIRST_COL;
I:=(OUT_COL_PTR+MIN_COL_SEP-FIRST_COL-1)*COL_SEP;
BEGIN I=0 THEN I:=0;
I:=I-FIRST_COL;
BEGIN I=OUT_BUF_PTR-1 > LINE_LENGTH THEN BEGIN "WONT FIT"
WRITENL(); I:=FIRST_COL; OUT_COL_PTR:=1;
END;
FOR J:=OUT.COL_PTR TO I-1 DO WRITE(' '*);
END;
OUT.COL_PTR:=I;
END;
PROCEDURE WRITE_BUF (ALIGN: BOOLEAN);  # S
VAR I: INTEGER;  # S
BEGIN
IF ALIGN THEN NEXT_COL;
156 BUFFER OUTPUT ROUTINES"
157
158 "BUFFER OUTPUT ROUTINES"
159
160 "BUFFER OUTPUT ROUTINES"
161
162 PROCEDURE NEXT_COL;  # S
VAR I, J: INTEGER;
BEGIN "PAD TO START OF NEXT OP COLUMN"
I:=FIRST_COL;
I:=(OUT_COL_PTR+MIN_COL_SEP-FIRST_COL-1)*COL_SEP;
BEGIN I=0 THEN I:=0;
I:=I-FIRST_COL;
BEGIN I=OUT_BUF_PTR-1 > LINE_LENGTH THEN BEGIN "WONT FIT"
WRITENL(); I:=FIRST_COL; OUT_COL_PTR:=1;
END;
FOR J:=OUT.COL_PTR TO I-1 DO WRITE(' '*);
END;
OUT.COL_PTR:=I;
END;
PROCEDURE WRITE_BUF (ALIGN: BOOLEAN);  # S
VAR I: INTEGER;  # S
BEGIN
IF ALIGN THEN NEXT_COL;
166 II:=(OUT_COL_PTR+MIN_COL_SEP-FIRST_COL-1)*COL_SEP;
167 IF I=0 THEN I:=0;
168 I:=I-FIRST_COL;
169 IF I=OUT_BUF_PTR-1 > LINE_LENGTH THEN BEGIN "WONT FIT"
170 WRITE(INL); I:=FIRST_COL; OUT_COL_PTR:=1;
171 END;
172 FOR J:=OUT.COL_PTR TO I-1 DO WRITE(' '*);
173 OUT.COL_PTR:=I;
174 END;
175 PROCEDURE WRITE_BUF (ALIGN: BOOLEAN);  # S
VAR I: INTEGER;  # S
BEGIN
IF ALIGN THEN NEXT_COL;
176 BUFFER OUTPUT ROUTINES"
177
178 "BUFFER OUTPUT ROUTINES"
179
180 "BUFFER OUTPUT ROUTINES"
181
182 PROCEDURE NEXT_COL;  # S
VAR I, J: INTEGER;
BEGIN "PAD TO START OF NEXT OP COLUMN"
I:=FIRST_COL;
I:=(OUT_COL_PTR+MIN_COL_SEP-FIRST_COL-1)*COL_SEP;
BEGIN I=0 THEN I:=0;
I:=I-FIRST_COL;
BEGIN I=OUT_BUF_PTR-1 > LINE_LENGTH THEN BEGIN "WONT FIT"
WRITENL(); I:=FIRST_COL; OUT_COL_PTR:=1;
END;
FOR J:=OUT.COL_PTR TO I-1 DO WRITE(' '*);
END;
OUT.COL_PTR:=I;
END;
PROCEDURE WRITE_BUF (ALIGN: BOOLEAN);  # S
VAR I: INTEGER;  # S
BEGIN
IF ALIGN THEN NEXT_COL;
FOR I:=1 TO OUT_BUF_PTR-1 DO WRITE(OUT_BUF[I]);
OUT_BUF_PTR:=OUT_BUF_PTR+INT_BUF_PTR-1;
OUT_BUF_PTR:=1;
END;

PROCEDURE WRITE_CHAR (C: CHAR);
BEGIN
  IF OUT_BUF_PTR+IN_LEN_LENGTH THEN
    WRITE_BUF(TIME);
  OUT_BUF[OUT_BUF_PTR]:=C;
  OUT_BUF_PTR:=OUT_BUF_PTR+1;
END;

"BUFFER FORMATTING ROUTINES"

PROCEDURE WRITE_CHAR (TEXT: CHAR);
VAR I: INTEGER;
BEGIN
  FOR I := 1 TO 8 DO
    IF (TEXT[I] <> '.') THEN
      THEN WRITE_CHAR (TEXT[I])
  END;
PROCEDURE WRITE_HEX_CHAR (VAL: SHORTINTEGER);
VAR I: SHORTINTEGER;
BEGIN
  IF VAL > 9 THEN WRITE_CHAR(CH(VAL-10+ORD('A')))
  ELSE WRITE_CHAR(CH(VAL+ORD('0')));
END;
PROCEDURE WRITE_HEX (VAL: SHORTINTEGER; N: SHORTINTEGER);
VAR I: SHORTINTEGER;
BEGIN
  X := XOR CASK/photos [N];
  Y := (X OR 1); 12
  Z := (X OR 2); 12
  END;
PROCEDURE WRITE_INT (VAL: SHORTINTEGER);
VAR A: ARRAY [1..10] OF CHAR;
I, J, REM: INTEGER;
BEGIN
  REM := VAL; I := 1;
  REPEAT
    A[I]:=CHR(CHAR(REM MOD 10)+ORD('0'));
    I := I+1;
    REM := REM DIV 10;
  UNTIL REM = 0;
  IF VAL<0 THEN WRITE_CHAR('-');
  FOR J := 1 DOWNTO 1 DO WRITE_CHAR(A[J]);
END;

"PRINTS THE VALUE OF 'VAL' IN A FIELD OF 'N' POSITIONS.
C VAL WILL BE RIGHT JUSTIFIED IN THE FIELD."
240 VAR A: ARRAY [1..10] OF CHAR;
241 I, J, REN: INTEGER;
242 BEGIN
243 REN:=VAL; I:=1;
244 REPEAT
245 A[I]:=CHR(ASC(REN MOD 10)+ORD('0'));
246 I:=I+1; REN:=REN DIV 10;
247 UNTIL REN = 0;
248 IF VAL < 0 THEN BEGIN
249 A[I]:="-"; I:=I+1;
250 END;
251 FOR J:=I TO N DO WRITE_CHAR(' ');
252 FOR J:=I-1 DOWNTO 1 DO WRITE_CHAR(A[J]);
253 END;
254
255 PROCEDURE INDEXERROR (TEXT:CHAR; BADINDEX: SHORTINTEGER);
256 "IF THIS PROCEDURE IS INVOKED IT PROBABLY MEANS SOME OPERATOR
C WHICH WAS ENCOUNTERED EARLIER HAS THE WRONG 'NUM_ARGS' (NO. OF
C OPERANDS) VALUE ASSIGNED TO IT IN THE OPERATOR TABLE."
257
258 VAR I: INTEGER;
259 BEGIN
260 WRITE_CHAR ('$'#$#$#$#$');
261 WRITE_CHAR ('INDEX ');
262 WRITE_CHAR ('*');
263 WRITE_CHAR ('ERROR ');
264 WRITE_CHAR ('$'#$#$#$#$');
265 WRITE_CHAR ('TEXT');
266 WRITE_CHAR ('*'');
267 WRITE_CHAR ('*');
268 WRITE_INT (BADINDEX);
269 WRITE_CHAR ('*');
270 WRITE_CHAR ('*');
271 END;
272
273 "GENERAL OPERATOR FORMATTING ROUTINE"
274
275 PROCEDURE WRITE_OP (OP_NAME: CHAR; NUM_ARGS, KIND_ARG, TYPE_ARG, 
276 MODE_ARG, CONTEXT_ARG: SHORTINTEGER);
277 VAR ARG_NO, ARG_VAL, I: SHORTINTEGER;
278 BEGIN
279 WRITE_CHAR (OP_NAME);
280 FOR ARG_NO := 1 TO NUM_ARGS DO
281 BEGIN
282 IF ARG_NO = 1
283 THEN WRITE_CHAR ('(')
284 ELSE WRITE_CHAR ('');
285 READ_FLT (ARG_VAL);
286 IF ARG_NO = KIND_ARG THEN
287 BEGIN
288 IF (ARG_VAL<CHIN_KIND) OR (ARG_VAL>MAX_KIND)
289 THEN INDEXERROR ('BAD_KIND', ARG_VAL)
290 ELSE WRITE_CHAR ('NO_KIND')
291 END ELSE IF
292 ARG_NO = TYPE_ARG THEN
293 BEGIN
294 IF (ARG_VAL<CHIN_TYPE) OR (ARG_VAL>MAX_TYPE)
295 THEN INDEXERROR ('BAD-TYPE', ARG_VAL)
296 256 280
297
298
ELSE WRITE_CHAR ('NO_TYPE_')
ELSE IF
ARG_NO = MODE_ARG THEN
BEGIN
IF (ARG_VAL,MODE) OR (ARG_VAL,MAX_MODE) THEN INDEXERROR ('BAD-MODE', ARG_VAL)
ELSE WRITE_CHAR ('NO_MODE_')
END ELSE IF
ARG_NO = CONTEXT_ARG THEN
BEGIN
IF (ARG_VAL,CONTEXT) OR (ARG_VAL,MAX_CONTEXT) THEN INDEXERROR ('BAD-CTXT', ARG_VAL)
ELSE WRITE_CHAR ('NO_CTXT_')
END ELSE IF (ARG_VAL < 32768) OR (ARG_VAL > 32767) THEN WRITE_HEX (ARG_VAL, 2)
ELSE WRITE_INT (ARG_VAL)
END; "OP_DISPLAY"
IF HIPO_ARG NO THEN WRITE_CHAR(')');
WRITE_BUF(TRUE);
176 188
END;
"SPECIAL CASE ROUTINE FOR NEW LINE OPERATOR"

PROCEDURE NEW_LN;
VAR ARG: SHORTINTEGER;
BEGIN
READ IFL(ARG); "LINE NUMBER"
IF OUT_COL_PTR<>1 THEN WRITE(NL);
WRITE(NL);
OUT_COL_PTR:=1;
WHITE_CHAR('LINE ')'; WRITE_INT(ARG,5);
WHITE_BUF(FALSE); "WRITE WITH NO COL ALIGN"
END;
"SPECIAL CASE ROUTINE FOR LONG CONSTANT"

PROCEDURE WRITE_LCONST;
VAR LEN, I, N, ARG: SHORTINTEGER;
BEGIN
WRITE_CHAR('LCONST'); WRITE_CHAR('(');
READ_IFL(LEN);
WRITE_INT(LEN); WHITE_CHAR(')');
WHITE_CHAR ('HEX-VAL');
NEXT_COL; N:=(LEN-1) DIV 2 + 1;
FOR I:=1 TO N DO BEGIN
IF OUT_COL_PTR<>OUT_BUF_PTR-1+9 THEN BEGIN
WRITE_BUF(FALSE); WRITE(NL); OUT_COL_PTR:=1; NEXT_COL;
END;
END;
WRITE_BUF(FALSE);
END;
"COMMON INITIALIZATION"
PROCEDURE INITIALIZE;
BEGIN

WORDS_IN:=PAGE_LENGTH;

PAGES_IN:=1;

PASS_NO:=PARAM[10].INT;

OUT_COL_PTR:=1;

OUT_ROW_PTR:=1;

FIRST_COL:=12;

MIN_COL_SEP:=3;

COL_SEP:=12;

END;

"WRITE PASS LISTING HEADER"

PROCEDURE WRITE_TEXT (TEXT: LINE);

VAR I: INTEGER;

BEGIN

I:=1;

WHILE TEXT[I] <> '4' DO BEGIN

WRITE(TEXT[I]); I:=I+1;

END;

END;

PROCEDURE WRITE_HEADER;

VAR I, J: INTEGER;

C: CHAR;

BEGIN

C:=CHR(PASS_NO+ORD('0'));

WRITE('P');

WRITE('N');

FOR I:=1 TO LINE_LENGTH DO WRITE('C');

WRITE('L');

WRITE('C');

J:=LINE_LENGTH DIV 2 - 1;

FOR I:=2 TO J-1 DO WRITE(' ');;

WRITE('MPASCA.L INTERMEDIATE CODE PASS *');

WRITE('C');

FOR I:=2 TO J-1 DO WRITE(' ');;

WRITE('C');

FOR I:=1 TO LINE_LENGTH-1 DO WRITE(' ');;

WRITE('C');

WRITE('L');

END;

PROCEDURE WRITE 56 CASE (OP_NAME: CHAR8; PASS: INTEGER);

VAR I, MIN, MAX, MINUS_MIN,

LOCATION,

STM_LABEL, SHORTINTEGER;

BEGIN

WRITE_CHAR(OP_NAME);

WRITE_CHAR(' ');;

READ_TPL(MIN);

WRITE_TPL(MIN);

WRITE_CHAR(' ');;

IF PASS = 5 THEN BEGIN

END;
420 READ_IFL(MAI);
421 WRITE_INT(MAI);
422 MAX_MINUS_MIN := MAX - MIN;
423 END
424 ELSE BEGIN "PASS 6"
425 READ_IFL(MAX_MINUS_MIN);
426 WRITE_INT(MAX_MINUS_MIN);
427 WRITE_CHAR(',');
428 READ_IFL(LOCATION);
429 WRITE_INT(LOCATION);
430 END;
431 FOR I := 0 TO MAX_MINUS_MIN DO
432 BEGIN
433 WRITE_CHAR('*');
434 READ_IFL(SYM_IT_LABEL);
435 WRITE_INT(SYM_IT_LABEL);
436 END;
437 WRITE_CHAR('*/');
438 END;
439
440 PROCEDURE WRITE_PASS7_CASE (OP_NAME: CHAR8);
441 VAR
442 I: INTEGER;
443 5
444 MIN,
445 MAX_MINUS_MIN,
446 NUM_OF_DISTANCES,
447 12
448 BEGIN
449 READ_Char8(OP_NAME);
450 WRITE_Char8('(*' );
451 WRITE_Char8('('* ));
452 READ_IFL(MIN);
453 WRITE_INT(MIN);
454 WRITE_Char8('('* ));
455 READ_IFL(MAX_MINUS_MIN);
456 WRITE_INT(MAX_MINUS_MIN);
457 WRITE_Char8('('* ));
458 NUM_OF_DISTANCES := MAX_MINUS_MIN + 1;
459 FOR I := 1 TO NUM_OF_DISTANCES DO
460 BEGIN
461 WRITE_Char8('('* );
462 READ_IFL(ARG);
463 WRITE_IT(ARG);
464 END;
465 WRITE_Char8('*/');
466 END;
467
468 PROCEDURE PASS5;
469 VAR OP5: OP5;
470 CONST MIN_OP5 = 0; MAX_OP5 = 48;
471 CONST
472 PASS5_TABLE = [
473 'PUSHCONST', 1, 'PUSHVAR', 3, '1'
474 'PUSHIND', 1, 'PUSHADD', 2,
475 'FIELD', 1, 'INDEX', 3,
476 'POINTER', 0, 'VARIANT', 2,
477 'RANGE', 2, 'ASSIGN', 1, '9'
478 'ASSIGNTAG', 0, 'COPY', 1,
479 'NEW', 0, 'NOT', 0,
AND 1, 'OR' 1,
'NEG' 1, 'ADD' 1, "19"
'SUB' 1, 'MUL' 1,
'DIV' 1, 'MOD' 1,
'INVALID' 0, 'INVALID' 0,
'function' 2, 'HILSET' 0,
'COMPARE' 2, 'CHISTRUC' 2,
'FUNCTION' 2, 'OUTPUTLABEL' 1,
'JUMP' 1, 'FALSJUMP' 1,
'CASEJUMP' 0, 'INITVAR' 0,
'CALL' 3, 'ENTER' 5,
'RETURN' 1, 'TOP' 1,
'newline' 1, 'ERROR' 0,
'INCARGS' 0, 'MESSAGE' 2,
'INCARG' 0, 'DOCUMENT' 0,
'PROCEDURE' 1, 'INIT' 4,
'FUSHLABEL' 1, 'CALLPROC' 0,
'EOM' 11,

ARRAY[MAX_OPS .. MAX_OPS] OF

RECORD
  OP_NAME: CHAR8;
  NUM_ARGS: BYTES;
END;

CONST
  MAX_OPS = 40;
  NUM_ARGS = 214;

REF IN

IN_FILE := 2;
CH_SEP := 3;
REPEAT
  HEAD_IFL (OP);
  WITH PASSS_TABLE[OP] DO
    IF (OP < MIN OPS) OR (OP > MAX_OPS) THEN INERROR ('BAD_OP', OP)
    ELSE IF OP = NEWLINES THEN NEWLINE
    ELSE IF OP = CASEJUMPS THEN WRITE_S Case (OP_NAME, PASS_N)
    THEN WRITE_S Case (OP_NAME, PASS_N)
    ELSE IF OP = LONGCONSTS THEN WRITE (LCONST)
    ELSE WRITE_OP (OP_NAME, NUM_ARGS, 0, 0, 0)
    UNTIL OP = EOM;
  WRITE (E.M.);
END;

PROCEDURE PASSS;
  MAX_OPS = 113;

CONST
  PASSS_TABLE :=
    'CONSTANT' 1, 'LOCALADD' 1, "1"
    'GLOBALADD' 1, 'PUSHCONS' 1,
    'PUSHLOC' 1, 'OUTPUTLABEL' 1,
    'FUSHLABEL' 0, 'PUSHBYTE' 0,
ARRAY [MIN_OPCODE..MAX_OPCODE] OF

RECORD
  OP_NAME: CHAR;
  NUM_ARGS: BYTE;
END;

534 534
535 535
536 536
537 537
538 538
539 539
540 540
541 541
542 542
543 543
544 544
545 545
546 546
547 547
548 548
549 549
550 550
551 551
552 552
553 553
554 554
555 555
556 556
557 557
558 558
559 559
560 560
561 561
562 562
563 563
564 564
565 565
566 566
567 567
568 568
569 569
570 570
571 571
572 572
573 573
574 574
575 575
576 576
577 577
578 578
579 579
580 580
581 581
582 582
583 583
584 584
585 585
586 586
587 587
588 588
589 589
590 590
591 591
592 592
593 593
594 594
595 595
596 596
597 597
598 598
599 599
600 600
BEGIN
IN_FILE:=1;
COL_OPT:=3;
REPEAT
READ_FIL(OP);
WITH PASSTABLE[OP] DO
IF (OP < MIN_OP) OR (OP > MAX_OP)
THEN INDEXERROR('BAD-OP', OP)
ELSE IF OP = NEWLINE
THEN NEW_LINE
ELSE IF OP = CASEJMP
THEN WRITE_5C_CASE(OP_NAME, PASS_NO)
UNTIL OP = EOM;
WRITE(IND);
END;

PROCEDURE PASST;
CONST
PASSTABLE = 
'CONSTADD', 1, 'LOCALADD', 1, 2,
'GLOBALADD', 1, 'PUSHVAR', 1,
'PUSHLOC', 1, 'PUSHLOCN', 1,
'PUSHARG', 0, 'PUSHBYTE', 0,
'PUSHREAL', 0, 'PUSHSET', 0, 10,
'FIELD', 1, 'INDEX', 3,
'POINTER', 1, 'VARIANT', 2,
'RENAME', 2, 'COPYBYTE', 0,
'COPYWORD', 0, 'COPYREAL', 0,
'COPYSET', 0, 'COPYTAG', 1, 20,
'COPYSTRU', 1, 'NEW', 2,
'NEWCHR', 2, 'NOT', 0,
'ANDWORD', 0, 'ANDSET', 0,
'ORWORD', 0, 'ORSET', 0,
'ANDWORD', 0, 'ANDREAL', 0,
'ORWORD', 0, 'ORREAL', 0, 30,
'ANDWORD', 0, 'ANDREAL', 0,
'SUBWORD', 0, 'SUBREAL', 0,
'SUBSET', 0, 'SUBWORD', 0,
'MULREAL', 0, 'DIVWORD', 0,
'MULREAL', 0, 'DIVREAL', 0,
'MULSET', 0, 'INSET', 0,
'LSWORD', 0, 'LSWORD', 0,
'LSWORD', 0, 'LSWORD', 0,
'LSREAL', 0, 'LSREAL', 0,
'LSREAL', 0, 'LSREAL', 0,
'LSREAL', 0, 'LSREAL', 0,
'LSTSTRUCT', 1, 'EQUSTRUCT', 1, 60,
'GSTRUCT', 1, 'HSTRUCT', 1,
'NESTSTRUCT', 1, 'NESTSTRUCT', 1,
'FUNCA', 1, 'JUMP', 1,
'FALSEJMP', 1, 'CASEJUMP', 2,
660 'INITVAR', 1, 'CALL', 1, "70"
661 'CALLTS', 1, 'ENTER', 8, "*
662 'EXIT', 0, 'ENTRPROG', 8, "*
663 'ENTRPROG', 0, 'BECUM', 8, "*
664 'BECUM', 0, 'ENTRCLASS', 8, "*
665 'ENTRCLASS', 0, 'TRANSPORT', 8, "*
666 'TRANSPORT', 0, 'ENTRCLASS', 8, "*
667 'ENTRCLASS', 0, 'EXITMACH', 8, "*
668 'EXITMACH', 0, 'ENTRPROG', 8, "*
669 'ENTRPROG', 0, 'POP', 1, "*
670 'POP', 1, 'INSTRUCTION', 0, "99"
671 'INSTRUCTION', 0, 'INIT', 2, "*
672 'INIT', 2, 'INIT', 4, "*
673 'INIT', 4, 'INIT', 0, "*
674 'INIT', 0, 'TRUE', 0, "*
675 'TRUE', 0, 'ABS', 0, "*
676 'ABS', 0, 'SUCWORD', 0, "100"
677 'SUCWORD', 0, 'COND', 0, "*
678 'COND', 0, 'ATTRIBUTE', 0, "*
679 'ATTRIBUTE', 0, 'REALTIME', 0, "*
680 'REALTIME', 0, 'DISK', 0, "*
681 'DISK', 0, 'CONTINUE', 0, "I/O"
682 'CONTINUE', 0, 'STOP', 0, "110"
683 'STOP', 0, 'SETUP', 0, "WAIT"
684 "112"
685 ARRAY [1..12] OF
686 RECORD
687 'NAME': CHAR8;
688 'NUMARGS': SHORTINTEGER;
689 END;
690 NEWLINE = 178;
691 CASEJUMP = 136;
692 VAR
693 HALF_OP: SHORTINTEGER;
694 'PROGLEN', 'STACKLEN', 'VARLEN', 'CONST', 'SHORTINTEGER';
695 "*
696 BEGIN
697 'IN_FILE' := 3;
698 'COL_NUM' := 1;
699 READ_IFL('PROGLEN'); 128
700 READ_IFL('STACKLEN');
701 READ_IFL('VARLEN');
702 READ_IFL('CONST');
703 READ_IFL('SHORTINTEGER');
704 READ_IFL('SHORTINTEGER');
705 READ_IFL('SHORTINTEGER');
706 READ_IFL('SHORTINTEGER');
707 WRITE_CHAR8('PROGLEN'); 'WRITEposure' 'PROGLEN';
708 WRITE_CHAR8('STACKLEN'); 'WRITEposure' 'STACKLEN';
709 WRITE_CHAR8('VARLEN'); 'WRITEposure' 'VARLEN';
710 WRITE_CHAR8('CONST'); 'WRITEposure' 'CONST';
711 WRITE_CHAR8('SHORTINTEGER'); 'WRITEposure' 'SHORTINTEGER';
712 WRITE_CHAR8('SHORTINTEGER'); 'WRITEposure' 'SHORTINTEGER';
713 WRITE_CHAR8('SHORTINTEGER'); 'WRITEposure' 'SHORTINTEGER';
714 WRITE_CHAR8('SHORTINTEGER'); 'WRITEposure' 'SHORTINTEGER';
715 WRITE_CHAR8('SHORTINTEGER'); 'WRITEposure' 'SHORTINTEGER';
716 WRITE_CHAR8('SHORTINTEGER'); 'WRITEposure' 'SHORTINTEGER';
717 WRITE_CHAR8('SHORTINTEGER'); 'WRITEposure' 'SHORTINTEGER';
718 REPEAT
720  READ_IFL(OP);
721  HALF_OP := OP DIV 2;
722  IF (HALF_OP <= 0) OR (HALF_OP > 112)
723      THEN BEGIN
724          WRITE_CHAR(‘BAD OP: ’);
725          WRITE_INT(OP);
726          WRITE_CHAR(‘ (:10:) ’);
727          FOR I := 1 TO PAGELENGTH + 10 DO WRITE_CHAR(‘ ’);
728      END;
729      WITH PASTY_TABLE[HALF_OP] DO
730          IF OP = NEWLINE
731              THEN NEW_LINE
732              ELSE IF OP = CASE gutter
733                  THEN WRITE_PASS7_CASE(OF_NAME)
734                  ELSE WRITE_OP(OF_NAME, NUM_ARGS, 0, 0, 0, 0)
735          UNTIL CODE_READ_SO_FAR >= CODELENGTH;
736          WRITE_CHAR(‘ (:10:) ’);
737          WRITE_CHAR(‘END-‘);
738          WRITE_CHAR(‘NL’);
739          WRITE_CHAR(‘CSTANT’);
740          WRITE_CHAR(‘S-‘);
741          WRITE_CHAR(‘S-IN-HEX’);
742      END;
743  REPEAT
744  IF OUT_COL_PTR + OUT_BUF_PTR - 1 + 9 > LINE_LENGTH
745      THEN BEGIN
746          WRITE_BUF(FALSE);
747          WRITE(‘NL’);
748          OUT_COL_PTR := 1;
749          NEXT_COL;
750      END;
751  WRITE_HEX(OF, 2);
752  WRITE_CHAR(‘ ’);
753  WRITE_CHAR(‘ ’);
754  UNTIL CODE_READ_SO_FAR >= PROGLNGTH - 8;
755      FOR I := 1 TO PAGELENGTH + 10 DO WRITE_CHAR(‘ ’);
756  END;
757
758
759  BEGIN
760  INITIALIZE;
761  WRITE_HEADER;
762  CASE PASS_NO OF
763      1: "PASS1";
764      2: "PASS2";
765      3: "PASS3";
766      4: "PASS4";
767      5: "PASS5";
768      6: "PASS6";
769      7: "PASS7";
770      8, 9: "NOT IMPLEMENTED";
771      END;
772
773  END.
<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDF</td>
<td>41</td>
</tr>
<tr>
<td>EDFMID</td>
<td>41</td>
</tr>
<tr>
<td>EOF</td>
<td>89*</td>
</tr>
<tr>
<td>EOM</td>
<td>99*</td>
</tr>
<tr>
<td>EOMID</td>
<td>506*</td>
</tr>
<tr>
<td>EOP</td>
<td>526*</td>
</tr>
<tr>
<td>EOPMID</td>
<td>601*</td>
</tr>
<tr>
<td>EOSH</td>
<td>618*</td>
</tr>
<tr>
<td>ECOMOP</td>
<td>138*</td>
</tr>
<tr>
<td>F</td>
<td>76*</td>
</tr>
<tr>
<td>FAILURE</td>
<td>40</td>
</tr>
<tr>
<td>FALSE</td>
<td>332</td>
</tr>
<tr>
<td>FFS</td>
<td>348</td>
</tr>
<tr>
<td>FILE</td>
<td>353</td>
</tr>
<tr>
<td>FILEATTR</td>
<td>746*</td>
</tr>
<tr>
<td>FILEID</td>
<td>9*</td>
</tr>
<tr>
<td>FIRSTCOL</td>
<td>165*</td>
</tr>
<tr>
<td>FOUND</td>
<td>166*</td>
</tr>
<tr>
<td>GET</td>
<td>168*</td>
</tr>
<tr>
<td>170*</td>
<td></td>
</tr>
<tr>
<td>366*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>76*</td>
</tr>
<tr>
<td>78*</td>
<td></td>
</tr>
<tr>
<td>152*</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>691*</td>
</tr>
<tr>
<td>HEADER</td>
<td>721*</td>
</tr>
<tr>
<td>HRAILIMIT</td>
<td>722*</td>
</tr>
<tr>
<td>172*</td>
<td></td>
</tr>
<tr>
<td>1729*</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>163*</td>
</tr>
<tr>
<td>165*</td>
<td></td>
</tr>
<tr>
<td>166*</td>
<td></td>
</tr>
<tr>
<td>167*</td>
<td></td>
</tr>
<tr>
<td>169*</td>
<td></td>
</tr>
<tr>
<td>170*</td>
<td></td>
</tr>
<tr>
<td>172*</td>
<td></td>
</tr>
<tr>
<td>173*</td>
<td></td>
</tr>
<tr>
<td>177*</td>
<td></td>
</tr>
<tr>
<td>180*</td>
<td></td>
</tr>
<tr>
<td>180*</td>
<td></td>
</tr>
<tr>
<td>180*</td>
<td></td>
</tr>
<tr>
<td>196*</td>
<td></td>
</tr>
<tr>
<td>199*</td>
<td></td>
</tr>
<tr>
<td>199*</td>
<td></td>
</tr>
<tr>
<td>200*</td>
<td></td>
</tr>
<tr>
<td>211*</td>
<td></td>
</tr>
<tr>
<td>218*</td>
<td></td>
</tr>
<tr>
<td>219*</td>
<td></td>
</tr>
<tr>
<td>220*</td>
<td></td>
</tr>
<tr>
<td>226*</td>
<td></td>
</tr>
<tr>
<td>229*</td>
<td></td>
</tr>
<tr>
<td>230*</td>
<td></td>
</tr>
<tr>
<td>231*</td>
<td></td>
</tr>
<tr>
<td>234*</td>
<td></td>
</tr>
<tr>
<td>241*</td>
<td></td>
</tr>
<tr>
<td>243*</td>
<td></td>
</tr>
<tr>
<td>245*</td>
<td></td>
</tr>
<tr>
<td>246*</td>
<td></td>
</tr>
<tr>
<td>248*</td>
<td></td>
</tr>
<tr>
<td>249*</td>
<td></td>
</tr>
<tr>
<td>250*</td>
<td></td>
</tr>
<tr>
<td>251*</td>
<td></td>
</tr>
<tr>
<td>252*</td>
<td></td>
</tr>
<tr>
<td>260*</td>
<td></td>
</tr>
<tr>
<td>280*</td>
<td></td>
</tr>
<tr>
<td>338*</td>
<td></td>
</tr>
<tr>
<td>385*</td>
<td></td>
</tr>
<tr>
<td>351*</td>
<td></td>
</tr>
<tr>
<td>374*</td>
<td></td>
</tr>
<tr>
<td>376*</td>
<td></td>
</tr>
<tr>
<td>377*</td>
<td></td>
</tr>
<tr>
<td>378*</td>
<td></td>
</tr>
<tr>
<td>379*</td>
<td></td>
</tr>
<tr>
<td>383*</td>
<td></td>
</tr>
<tr>
<td>386*</td>
<td></td>
</tr>
<tr>
<td>39*</td>
<td></td>
</tr>
<tr>
<td>393*</td>
<td></td>
</tr>
<tr>
<td>396*</td>
<td></td>
</tr>
<tr>
<td>144*</td>
<td></td>
</tr>
<tr>
<td>163*</td>
<td></td>
</tr>
<tr>
<td>165*</td>
<td></td>
</tr>
<tr>
<td>166*</td>
<td></td>
</tr>
<tr>
<td>167*</td>
<td></td>
</tr>
<tr>
<td>168*</td>
<td></td>
</tr>
<tr>
<td>169*</td>
<td></td>
</tr>
<tr>
<td>170*</td>
<td></td>
</tr>
<tr>
<td>172*</td>
<td></td>
</tr>
<tr>
<td>173*</td>
<td></td>
</tr>
<tr>
<td>177*</td>
<td></td>
</tr>
<tr>
<td>180*</td>
<td></td>
</tr>
<tr>
<td>180*</td>
<td></td>
</tr>
<tr>
<td>196*</td>
<td></td>
</tr>
<tr>
<td>199*</td>
<td></td>
</tr>
<tr>
<td>200*</td>
<td></td>
</tr>
<tr>
<td>211*</td>
<td></td>
</tr>
<tr>
<td>218*</td>
<td></td>
</tr>
<tr>
<td>219*</td>
<td></td>
</tr>
<tr>
<td>220*</td>
<td></td>
</tr>
<tr>
<td>226*</td>
<td></td>
</tr>
<tr>
<td>228*</td>
<td></td>
</tr>
<tr>
<td>230*</td>
<td></td>
</tr>
<tr>
<td>231*</td>
<td></td>
</tr>
<tr>
<td>234*</td>
<td></td>
</tr>
<tr>
<td>241*</td>
<td></td>
</tr>
<tr>
<td>243*</td>
<td></td>
</tr>
<tr>
<td>245*</td>
<td></td>
</tr>
<tr>
<td>246*</td>
<td></td>
</tr>
<tr>
<td>248*</td>
<td></td>
</tr>
<tr>
<td>249*</td>
<td></td>
</tr>
<tr>
<td>250*</td>
<td></td>
</tr>
<tr>
<td>251*</td>
<td></td>
</tr>
<tr>
<td>252*</td>
<td></td>
</tr>
<tr>
<td>260*</td>
<td></td>
</tr>
<tr>
<td>280*</td>
<td></td>
</tr>
<tr>
<td>338*</td>
<td></td>
</tr>
<tr>
<td>385*</td>
<td></td>
</tr>
<tr>
<td>351*</td>
<td></td>
</tr>
<tr>
<td>374*</td>
<td></td>
</tr>
<tr>
<td>376*</td>
<td></td>
</tr>
<tr>
<td>377*</td>
<td></td>
</tr>
<tr>
<td>378*</td>
<td></td>
</tr>
<tr>
<td>379*</td>
<td></td>
</tr>
<tr>
<td>383*</td>
<td></td>
</tr>
<tr>
<td>386*</td>
<td></td>
</tr>
<tr>
<td>39*</td>
<td></td>
</tr>
<tr>
<td>393*</td>
<td></td>
</tr>
<tr>
<td>396*</td>
<td></td>
</tr>
<tr>
<td>35*</td>
<td></td>
</tr>
<tr>
<td>37*</td>
<td></td>
</tr>
<tr>
<td>32*</td>
<td></td>
</tr>
<tr>
<td>106*</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td></td>
</tr>
<tr>
<td>60*</td>
<td></td>
</tr>
<tr>
<td>92*</td>
<td></td>
</tr>
<tr>
<td>35*</td>
<td></td>
</tr>
<tr>
<td>67*</td>
<td></td>
</tr>
<tr>
<td>89</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>50*</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td></td>
</tr>
<tr>
<td>59*</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td></td>
</tr>
<tr>
<td>100*</td>
<td></td>
</tr>
<tr>
<td>102*</td>
<td></td>
</tr>
<tr>
<td>35*</td>
<td></td>
</tr>
<tr>
<td>39*</td>
<td></td>
</tr>
<tr>
<td>99*</td>
<td></td>
</tr>
<tr>
<td>KIND</td>
<td>26</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
</tr>
<tr>
<td>KIND_ARG</td>
<td>278</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1.</th>
<th>140</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEN</td>
<td>338</td>
</tr>
<tr>
<td>LENGTH</td>
<td>86</td>
</tr>
<tr>
<td>LINE</td>
<td>16</td>
</tr>
<tr>
<td>LINED</td>
<td>15</td>
</tr>
<tr>
<td>LINED_LENGTH</td>
<td>114</td>
</tr>
<tr>
<td>LINED_OP</td>
<td>139</td>
</tr>
<tr>
<td>LOCATION</td>
<td>408</td>
</tr>
<tr>
<td>LONGCONSTANT</td>
<td>599</td>
</tr>
<tr>
<td>LONGOP</td>
<td>96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>11.</th>
<th>82</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX</td>
<td>406</td>
</tr>
<tr>
<td>MAXARG</td>
<td>40</td>
</tr>
<tr>
<td>MAX_CONTEXT</td>
<td>122</td>
</tr>
<tr>
<td>MAX_KIND</td>
<td>119</td>
</tr>
<tr>
<td>MAX_MINUS_MI</td>
<td>407</td>
</tr>
<tr>
<td>MAX_MOD</td>
<td>121</td>
</tr>
<tr>
<td>MAX_OP</td>
<td>470</td>
</tr>
<tr>
<td>MAX_OPS</td>
<td>534</td>
</tr>
<tr>
<td>MAX_TYPE</td>
<td>120</td>
</tr>
<tr>
<td>MIN</td>
<td>405</td>
</tr>
<tr>
<td>MIN_COL_ARGP</td>
<td>136</td>
</tr>
<tr>
<td>MIN_CONTEXT</td>
<td>122</td>
</tr>
<tr>
<td>MIN_KIND</td>
<td>119</td>
</tr>
<tr>
<td>MIN_MOD</td>
<td>121</td>
</tr>
<tr>
<td>MIN_OPS</td>
<td>470</td>
</tr>
<tr>
<td>MIN_TYPE</td>
<td>120</td>
</tr>
<tr>
<td>MODEARG</td>
<td>279</td>
</tr>
<tr>
<td>MODE</td>
<td>35</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1.</th>
<th>209</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>218</td>
</tr>
<tr>
<td>NMLINES</td>
<td>597</td>
</tr>
<tr>
<td>NMLINESD</td>
<td>602</td>
</tr>
<tr>
<td>NMLINESF</td>
<td>609</td>
</tr>
<tr>
<td>NMLINK</td>
<td>328</td>
</tr>
<tr>
<td>NEXT_COL</td>
<td>162</td>
</tr>
<tr>
<td>ML</td>
<td>52</td>
</tr>
<tr>
<td>ML_TYPE</td>
<td>9</td>
</tr>
<tr>
<td>NOTUSED</td>
<td>29</td>
</tr>
<tr>
<td>NOHARGS</td>
<td>728</td>
</tr>
<tr>
<td>NOH_OF_DISTA</td>
<td>406</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1.</th>
<th>141</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>720</td>
</tr>
<tr>
<td>OPEN</td>
<td>76</td>
</tr>
<tr>
<td>OPERATION</td>
<td>44</td>
</tr>
<tr>
<td>OP_NAME</td>
<td>278</td>
</tr>
<tr>
<td>ORD</td>
<td>205</td>
</tr>
<tr>
<td>CRT</td>
<td>61</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>35</td>
</tr>
<tr>
<td>Variable</td>
<td>Offset</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>OUT_PUTTACK</td>
<td>49</td>
</tr>
<tr>
<td>OUT_BUF</td>
<td>133</td>
</tr>
<tr>
<td>OUT_BUF_PTR</td>
<td>133</td>
</tr>
<tr>
<td>OUT_COL_PTR</td>
<td>132</td>
</tr>
<tr>
<td>OVFLOW</td>
<td>70</td>
</tr>
</tbody>
</table>

- F -

<table>
<thead>
<tr>
<th>Variable</th>
<th>76</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAGE</td>
<td>78</td>
</tr>
<tr>
<td>PAGELENGTH</td>
<td>11</td>
</tr>
<tr>
<td>PAGES_IN</td>
<td>129</td>
</tr>
<tr>
<td>PAGES_OUT</td>
<td>130</td>
</tr>
<tr>
<td>PAGE</td>
<td>100</td>
</tr>
<tr>
<td>PASS</td>
<td>469</td>
</tr>
<tr>
<td>PASS_TABLE</td>
<td>472</td>
</tr>
<tr>
<td>PASS</td>
<td>533</td>
</tr>
<tr>
<td>PASS_TABLE</td>
<td>535</td>
</tr>
<tr>
<td>PASS</td>
<td>623</td>
</tr>
<tr>
<td>PASS_TABLE</td>
<td>625</td>
</tr>
<tr>
<td>PASS</td>
<td>131</td>
</tr>
<tr>
<td>POINTER</td>
<td>58</td>
</tr>
<tr>
<td>POINTERERROR</td>
<td>70</td>
</tr>
<tr>
<td>PRINTDEV</td>
<td>33</td>
</tr>
<tr>
<td>PRODLNGTH</td>
<td>69</td>
</tr>
<tr>
<td>PRODRESULT</td>
<td>69</td>
</tr>
<tr>
<td>PROTECTED</td>
<td>28</td>
</tr>
<tr>
<td>PUT</td>
<td>52</td>
</tr>
</tbody>
</table>

- R -

<table>
<thead>
<tr>
<th>Variable</th>
<th>70</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>73</td>
</tr>
<tr>
<td>READAG</td>
<td>93</td>
</tr>
<tr>
<td>READLINE</td>
<td>91</td>
</tr>
<tr>
<td>READPAGE</td>
<td>99</td>
</tr>
<tr>
<td>READ_IFL</td>
<td>149</td>
</tr>
<tr>
<td>RELEASE</td>
<td>83</td>
</tr>
<tr>
<td>REM</td>
<td>226</td>
</tr>
<tr>
<td>RESULT</td>
<td>107</td>
</tr>
<tr>
<td>RIN</td>
<td>37</td>
</tr>
</tbody>
</table>

- S -

<table>
<thead>
<tr>
<th>Variable</th>
<th>93</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCRATCH</td>
<td>23</td>
</tr>
<tr>
<td>SCICODE</td>
<td>23</td>
</tr>
<tr>
<td>SHORTINTEGER</td>
<td>12</td>
</tr>
<tr>
<td>STACKLENGTH</td>
<td>634</td>
</tr>
<tr>
<td>STACKLIMIT</td>
<td>71</td>
</tr>
<tr>
<td>STARTMEDIUM</td>
<td>41</td>
</tr>
<tr>
<td>STATUS</td>
<td>45</td>
</tr>
<tr>
<td>STATE_LABEL</td>
<td>409</td>
</tr>
</tbody>
</table>

- T -

<table>
<thead>
<tr>
<th>Variable</th>
<th>57</th>
</tr>
</thead>
<tbody>
<tr>
<td>TACOL</td>
<td>210</td>
</tr>
<tr>
<td>TAGZ</td>
<td>210* 211*</td>
</tr>
<tr>
<td>TAGS</td>
<td>210* 212*</td>
</tr>
<tr>
<td>Tapedevice</td>
<td>33</td>
</tr>
<tr>
<td>TASK</td>
<td>104*</td>
</tr>
<tr>
<td>TASK.Kind</td>
<td>49* 104</td>
</tr>
<tr>
<td>Terminated</td>
<td>70</td>
</tr>
<tr>
<td>TEXT</td>
<td>91* 92* 195 199 200 256* 267 373* 377 378</td>
</tr>
<tr>
<td>TIME Limit</td>
<td>71</td>
</tr>
<tr>
<td>TOP</td>
<td>82 104</td>
</tr>
<tr>
<td>Transmission</td>
<td>33</td>
</tr>
<tr>
<td>True</td>
<td>188 310</td>
</tr>
<tr>
<td>Typedevice</td>
<td>33</td>
</tr>
<tr>
<td>Type-lang</td>
<td>278* 294</td>
</tr>
</tbody>
</table>

-UPSPACE 37

-VAL 203* 205 206 209* 210 224* 226 232 233 237* 243 248

-Varianterror | 70 |

-Varlength | 69* 70 71 |

-WORD.S_IN | 127 151 153 155 156 156 361* |

-WRITE 74* 170 172 173 178 179 188 189 187 188 189 190 191 193 193 395 396 397 397

-WRITEARG 93* |

-WRITE.EOF 37 |

-WRITELINE 92* |

-WRITETAG 90* |

-WRITE.SA.CAS 402 522 616 |

-WRITE.BUF 175 188 319 332 340 353 353 746 |

-WRITE.CHAR 185 200 205 206 233 251 252 264 268 269 371 272 286 287 316 340 342 |

-WRITE.CHAR.8 195 252 263 265 266 267 282 294 300 306 312 331 340 330 312 450 708 709 |

-WRITE_HEADER 382 762 |

-WRITE_HRX 209* 315 350 751 |

-WRITE.HEADER.CH 202 219 220 |

-WRITE_INTR 229 270 316 332 415 421 426 429 435 453 457 463 708 710 712 714 716 725 |

-WRITE_INTR.3 371 331 |

-WRITE_LCONST 337 528 |

-WRITE_EXP 278* 325 617 734 |

-WRITE_PASS 481 733 |

-WRITE_TEXT 373* 394 |

-X 212 217 219 220 |

END IREF 230 IDENTIFIERS 948 TOTAL REFERENCES 129 COLLISIONS.
"MIPASCAL MEBNEM - MNEMONIC TEST OUTPUT FORMATTING"
"FOR WESTERN DIGITAL MASCAL MICROENGINE"

1 \$ CONCURRENT := TRUE

2 ''###''

3 1 1 PREFIX $'

4 1 ###''

5 10 TYPE FULLWORD = INTEGER;
6 11 INTEGER = SHORTINTEGER;
7 12 CONST NSL = '((10:1);'  FF = '((12:1);'  CN = '((13:1);'  EM = '((25:1);'
8 13 NSTAGE = 512 DIV 7;
9 14 TYPE PAGE = ARRAY (.1..NSTAGE) OF INTEGER;
10 15 CONST BYTES_PER_INTEGER = 2;
11 16 CONST LINELENGTH = 70;
12 17 TYPE LINE = ARRAY (.1..LINELENGTH) OF CHAR;
13 18 CONST IDLELENGTH = 12;
14 19 TYPE IDENTIFIER = ARRAY (.1..IDLENGTH) OF CHAR;
20 20 TYPE FILE = 1..5;
21 21 TYPE FILEATTR = RECORD
22 22  KIND: FILEATTR;
23 23  ADDON: INTEGER;
24 24  NOTUSED: ARRAY (.1..5.) OF INTEGER
25 25  END;
26 26 TYPE IODEVICE =
27 27  (TYPEDEVICE, DISKDEVICE, TAPEDevice, PRNTDEVICE, CARDDevice);
28 28  TYPE IOOPERATION = (INPUT, OUTPUT, MOVE, CONTROL);
29 29  TYPE IOARM = (WRITEEOF, REMIND, UPSPACE, BACKSPACE);
30 30  TYPE IORESULT =
31 31  (COMPLETE, INTERVENTION, TRANSMISSION, FAILURE,
32 32  ENDFILE, ENDMEASURE, STARTMEDIAN);
33 33  TYPE IOPARAM = RECORD
34 34  OPERATION: IOOPERATION;
35 35  STATUS: IORESULT;
36 36  END: IOARM
37 37  END;
38 38 TYPE TASKKIND = (INPUTTASK, JOBTASK, OUTPUTTASK);
39 40 100;
40 45 45 100;
46 46 100;
47 47 100;
48 48 100;
49 49 100;
50 50 100;
51 51 100;
52 52 100;
53 53 100;
54 54 100;
55 55 100;
56 56 100;
57 57 100;
58 58 100;
59 59 100;
60 60 100;
TABLEPTR = @TABLE;
TABLE = RECORD
   NEXTPOSITION: TABLEPTR;
   CONTENTS: ARRAY[1..MAXWORD] OF INTEGER
END;

TABLEPART = RECORD
   ORIGINANCE, STACKLENGTH: INTEGER;
   JUMPABLE, CONSTANTABLE, JUMPABLE,
   EXITABLE, DATANIZEABLE: TABLEPTR
END;

TABLEPTR = @TABLEPART;
OPTION = 0..8;

CONST MAXINTFACE = 10;

TYPE
  IFPTR = &IFINFO;
  IFINFO = RECORD
    INTERFACES: INTEGER;
    INTERFACESIZE: ARRAY[1..MAXINTFACE] OF INTEGER
  END;

PASSTPR = @PASSTLINK;
PASSTLINK = RECORD
  OPTIONS: SET OF OPTION;
  LABELS, BLOCKS, CONSTANTS, XEP_OFFSETS: INTEGER;
  RESETPOINT: FULLWORD;
  TABLES: TABLELPR;
  INTERFACE: IFPTR
END;

TYPE ARTAG =
  (NILTYPE, BOOLTYPE, INTTYPE, IDTYPE, PTYTYPE);

TYPE ARTYPE = RECORD
   CASE TAG: ARTAG OF
   NILTYPE, BOOLTYPE: (BOOL: BOOLEAN);
   INTTYPE: (INT: INTEGER);
   IDTYPE: (ID: IDENTIFIER);
   PTYTYPE: (PTR: PASSTPR)
END;

CONST MAXARG = 10;

TYPE ARGLIST = ARRAY (.1..MAXARG) OF ARTYPE;

TYPE ARSEQ = (INF, OMT);

TYPE PRORESULT =
   (T_TERMINATED, OVERFLOW, POINTERERROR, RANGEERROR, VARIANTERROR,
    HEAPLIMIT, STACKLIMIT, CODELIMIT, TIMELIMIT, CALLERROR);

PROCEDURE READ(VAR C: CHAR);
PROCEDURE WRITE(C: CHAR);

PROCEDURE OPEN(F: FILE; ID: IDENTIFIER; VAR FOUND: BOOLEAN);
PROCEDURE CLOSE(F: FILE);
PROCEDURE GET(F: FILE; F: INTEGER; VAR BLOCK: UNIV PAGE);
PROCEDURE PUT(F: FILE; F: INTEGER; VAR BLOCK: UNIV PAGE);
FUNCTION LENGTH(F: FILE): INTEGER;

PROCEDURE MARK(VAR TOT: INTEGER);

PROCEDURE RELEASE(TOP: INTEGER);

PROCEDURE IDENTIFY(HEADER: LINE);

PROCEDURE ACCEPT(VAR C: CHAR);

PROCEDURE DISPLAY(C: CHAR);

PROCEDURE READPAGE(VAR BLOCK: UNIV PAGE; VAR EOF: BOOLEAN);

PROCEDURE WRITEPAGE(BLOCK: UNIV PAGE; EOF: BOOLEAN);

PROCEDURE HEADLINE(VAR TEXT: UNIV LINE);

PROCEDURE WRITEFILE(TEXT: UNIV LINE);

PROCEDURE READARG(S: ARGSEQ; VAR ARG: ARGTYPE);

PROCEDURE WRITEARG(S: ARGSEQ; ARG: ARGTYPE);

PROCEDURE LOOKUP(ID: IDENTIFIER; VAR ATTR: FILEATTR; VAR FOUNDED: BOOLEAN);

PROCEDURE IوترNSFER

PROCEDURE IOMUXE DEVICE; VAR PARAM: TPARAM; VAR BLOCK: UNIV PAGE);

PROCEDURE IOMUXE DEVICE: IODEVICE; VAR PARAM: TPARAM);

FUNCTION TASK: TASKIND;

PROCEDURE RUN(ID: IDENTIFIER; VAR PARAM: ARGLIST;

VAR LIST: INTEGER; VAR RESULT: PROGRESS);

PROGRAM HELPERS(VAR PARAM: ARGLIST);

"SIMPLE TYPES AND CONSTANTS"

CONST LINE_LENGTH = 70; "OUTPUT DEV LINE LENGTH"

TYPE CHAR = ARRAY [1..8] OF CHAR;

"CONSTANTS"

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 FILE BUFFER"
IN_FILE: INTEGER; "INPUT FILE"

PAGES_IN: INTEGER; "CURRENT PAGE NUMBER"

PAGE_IN: PAGE; "INPUT BUFFER"
PASS_NO: INTEGER; "COMPILER PASS NUMBER"
OUT_COL_PTR: INTEGER; "POSS ON OUTPUT LINE FOR BUF"
OUT_BUF_PTR: INTEGER; "POSS IN BUFFER"
OUT_BUF: LINE; "OPERATOR OUTPUT BUFFER"

FIRST_COL: INTEGER; "FIRST COL TO USE FOR NON-NEWLINE"
MIN_COL_SEP: INTEGER; "MIN BLANKS BETWEEN OPS"
COL_SEP: INTEGER; "SEP BETWEEN COL. BOUNDARIES"
OF: INTEGER; "CURRENT INPUT OPERATOR"

LINK: PASSPTR;
180  "INPUT FILE SUPPORT ROUTINES"
181
182  PROCEDURE READ_FIL (VAR ARG: INTEGER);
183       *  *  *  * 5
184  BEGIN
185  IF WORDS_IN = PAGELENGTH THEN BEGIN
186       167  15
187       GETIN_FILE; PAGES_IN:=PAGES_IN+1;
188       PAGES_IN:=PAGEN IN+1; WORDS_IN:=0;
189       169  169  167
190       END;
191  WORDS_IN:=WORDS_IN+1;
192  ARG:=PAGE_IN[WORDS_IN];
193  END;
194  "DUFFER OUTPUT ROUTINES"
195
196  PROCEDURE NEXT_COL;
197       *  *  *  * 5
198  VAR I, J: INTEGER;
199  BEGIN "PAD TO START OF NEXT OF COLUMN"
200     I := OUT_COL_PTR + MIN_COL_SEP;
201     IF I>OUT_BUF_PTR >> LINE_LENGTH "WON'T FIT"
202     THEN BEGIN
203     WRITE(ML);  114  13
204     I := FIRST_COL;
205     OUT_COL_PTR := 1;
206     END;
207     FOR J:=OUT_COL_PTR TO I-1 DO WRITE(' ');
208     OUT_COL_PTR:=I;
209     END;
210  "DUFFER BUFFER ROUTINES"
211  PROCEDURE WRITE_BU (ALIGN: BOOLEAN);
212  VAR I: INTEGER;
213  BEGIN
214     IF ALIGN THEN NEXT_COL;
215     FOR I:=1 TO OUT_BUF_PTR-1 DO WRITE(OUT_BUF[I]);
216     OUT_BUF_PTR:=OUT_COL_PTR+OUT_BUF_PTR-1;
217     END;
218
219  PROCEDURE WRITE_CHAR (C: CHAR);
220  BEGIN
221     IF OUT_BUF_PTR >> LINE_LENGTH THEN
222     WRITE_BU(TRUE);
223     OUT_BUF[OUT_BUF_PTR]:=C;
224     OUT_BUF_PTR:=OUT_BUF_PTR+1;
225     END;
226
227  "DUFFER FORMATTING ROUTINES"
228  PROCEDURE WRITE_CHAR (TEXT: CHAR);
229  VAR I: INTEGER;
230  BEGIN
231     FOR I := 1 TO 6 DO
232     IF (TEXT[I] <> ' ')
233     THEN WRITE_CHAR(TEXT[I]);
234     END;
235
236  PROCEDURE WRITE_HEX_CHAR (VAL: INTEGER);
237  BEGIN
238     IF VAL > 9 THEN WRITE_CHAR(CH(VAL-10+ORD('A')))
239     IF VAL < 10 THEN WRITE_CHAR(CH(VAL));
240   ELSE WRITE_CHAR(CHR(VAL+ORD('O')));
241 END;
242
243 PROCEDURE WRITE_HEX (VAL: INTERGER; N: INTERGER);
244   TYPE TAGS = (TAG1, TAG2);
245   VAR I: INTERGER;
246   X: RECORD CASE TAGS OF
247      TAG1: (INT: INTERGER);
248      TAG2: (BYTES: ARRAY [1..2] OF BYTE);
249   END;
250 BEGIN
251   X.INT:=VAL;
252   FOR I:=3-N TO 2 DO BEGIN
253      WRITE_HEX_CHAR(X.BYTES[I] DIV 16);
254      WRITE_HEX_CHAR(X.BYTES[I] MOD 16);
255   END;
256 END;
257
258 PROCEDURE WRITE_INT (VAL: INTERGER);
259   VAR A: ARRAY [1..10] OF CHAR;
260      I, J, REM: INTERGER;
261 BEGIN
262   REM:=VAL; I:=1;
263   REPEAT
264      A[I]:=CHR(ABS(REM MOD 10)+ORD('0'));
265      I:=I+1; REM:=REM DIV 10;
266      UNTIL REM = 0;
267      IF VAL<0 THEN WRITE_CHAR('-' );
268      FOR J:=I-1 DOWNTO 1 DO WRITE_CHAR(A[J]);
269 END;
270
271 PROCEDURE WRITE_INTR (VAL: INTERGER; N: INTERGER);
272 "WRITE INTERGER, RIGHT-JUST IN FIELD OF WIDTH N"
273 PROCEDURE WRITE_INTR (TEXT:CHAR8; INDX: INTERGER);
274   END;
275 BEGIN
276   REM:=VAL; I:=1;
277   REPEAT
278      A[I]:=CHR(ABS(REM MOD 10)+ORD('0'));
279      I:=I+1; REM:=REM DIV 10;
280   UNTIL REM = 0;
281 IF VAL<0 THEN BEGIN
282      A[I]:='-'; I:=1;
283 END;
284 FOR J:=I TO N DO WRITE_CHAR(' ');
285 FOR J:=I-1 DOWNTO 1 DO WRITE_CHAR(A[J]);
286 END;
287
288 PROCEDURE INDEXERROR (TEXT:CHAR8; BADINDEX: INTERGER);
289   END;
290 BEGIN
291 WRITE_CHAR ('$$$$$$$$$$');
292 WRITE_CHAR ('INDEX ');
293 WRITE_CHAR ('ERROR ');
294 WRITE_CHAR (''');
295 WRITE_CHAR ('''');
296 WRITE_CHAR (''$'$'$'$');
297 WRITE_CHAR (TEXT);
298 WRITE_CHAR ('=');
299 WRITE_CHAR ('''');
300 }
PROCEDURE WRITE_OP (OP_NAME : CHARS; NUM_ARGS : INTEGER, KIND_ARG : CHARS, TYPE_ARG : CHARS;
  MODX_ARG : MODX; CONTEXT_ARG : CONTEXT); BEGIN
  VAR ARG_NO , ARG_VAL : INTEGER;
  BEGIN
    WRITE_CHAR (OP_NAME);
    FOR ARG_NO := 1 TO NUM_ARGS DO
      IF ARG_NO = 1 THEN WRITE_CHAR ('( ')
      ELSE WRITE_CHAR (' ');
      READ_IFL (ARG_VAL);
      IF ARG_NO = KIND_ARG THEN
        BEGIN
          IF (ARG_VAL CHIN KIND) OR (ARG_VAL=MAX_KIND) THEN
            THEN INDEXERROR ('BAD_KIND', ARG_VAL)
          ELSE WRITE_CHAR ('NO_KIND ',)
          END ELSE IF
        ARG_NO = TYPE_ARG THEN
        BEGIN
          IF (ARG_VAL CHIN TYPE) OR (ARG_VAL=MAX_TYPE) THEN
            THEN INDEXERROR ('BAD_TYPE', ARG_VAL)
          ELSE WRITE_CHAR ('NO_TYPE ',)
          END ELSE IF
        ARG_NO = MODE_ARG THEN
        BEGIN
          IF (ARG_VAL CHIN MODE) OR (ARG_VAL=MAX_MODE) THEN
            THEN INDEXERROR ('BAD_MODE', ARG_VAL)
          ELSE WRITE_CHAR ('NO_MODE ',)
          END ELSE IF
        ARG_NO = CONTEXT_ARG THEN
        BEGIN
          IF (ARG_VAL CHIN CONTEXT) OR (ARG_VAL=MAX_CONTEXT) THEN
            THEN INDEXERROR ('BAD_CTX', ARG_VAL)
          ELSE WRITE_CHAR ('NO_CTX ',)
          END ELSE IF
        (ARG_VAL < -32768) OR (ARG_VAL > 32767) THEN INDEXERROR ('BAD_INT', ARG_VAL)
        ELSE WRITE_CHAR ('NO_INT ',)
        END ELSE IF
      END;
      IF NUM_ARGS>0 THEN WRITE_CHAR (')');
      WRITE_IFL (TRUE);
    END;
END;

"SPECIAL CASE ROUTINE FOR NEW LINE OPERATOR"
PROCEDURE NEW_LINE; BEGIN
  VAR ARG : INTEGER;
  BEGIN
    IF ARG < 6 THEN READ_IFL(ARG); "LINE NUMBER"
    IF OUT_COL_PTR<>1 THEN WRITE_CHAR(' ');
  END;

"GENERAL OPERATOR FORMATTING ROUTINE"
PROCEDURE WRITE_OP (OP_NAME : CHARS; NUM_ARGS : INTEGER, KIND_ARG : CHARS, TYPE_ARG : CHARS;
  MODX_ARG : MODX; CONTEXT_ARG : CONTEXT); BEGIN
  VAR ARG_NO , ARG_VAL : INTEGER;
  BEGIN
    WRITE_CHAR (OP_NAME);
    FOR ARG_NO := 1 TO NUM_ARGS DO
      IF ARG_NO = 1 THEN WRITE_CHAR ('( ')
      ELSE WRITE_CHAR (' ');
      READ_IFL (ARG_VAL);
      IF ARG_NO = KIND_ARG THEN
        BEGIN
          IF (ARG_VAL CHIN KIND) OR (ARG_VAL=MAX_KIND) THEN
            THEN INDEXERROR ('BAD_KIND', ARG_VAL)
          ELSE WRITE_CHAR ('NO_KIND ',)
          END ELSE IF
        ARG_NO = TYPE_ARG THEN
        BEGIN
          IF (ARG_VAL CHIN TYPE) OR (ARG_VAL=MAX_TYPE) THEN
            THEN INDEXERROR ('BAD_TYPE', ARG_VAL)
          ELSE WRITE_CHAR ('NO_TYPE ',)
          END ELSE IF
        ARG_NO = MODE_ARG THEN
        BEGIN
          IF (ARG_VAL CHIN MODE) OR (ARG_VAL=MAX_MODE) THEN
            THEN INDEXERROR ('BAD_MODE', ARG_VAL)
          ELSE WRITE_CHAR ('NO_MODE ',)
          END ELSE IF
        ARG_NO = CONTEXT_ARG THEN
        BEGIN
          IF (ARG_VAL CHIN CONTEXT) OR (ARG_VAL=MAX_CONTEXT) THEN
            THEN INDEXERROR ('BAD_CTX', ARG_VAL)
          ELSE WRITE_CHAR ('NO_CTX ',)
          END ELSE IF
        (ARG_VAL < -32768) OR (ARG_VAL > 32767) THEN INDEXERROR ('BAD_INT', ARG_VAL)
        ELSE WRITE_CHAR ('NO_INT ',)
        END ELSE IF
      END;
      IF NUM_ARGS>0 THEN WRITE_CHAR (')');
      WRITE_IFL (TRUE);
    END;
END;
360  WHITE_CHAR (NL); "DOUBLE SPACE BEFORE NEW LINE"
361  OUT_COL_PTR := 1;  
362  IF PASS_NO < 6  
363  THEN BEGIN  
364  WRITE_CHAR ('LINE  
365  WRITE_INTR (ARG, 5);  
366  WRITE_BUF (FALSE)  
367  END
368  ELSE WRITE_BUF (FALSE);  
369  END;  
370  
371  PROCEDURE SEIF (LINES: INTEGER);  
372  VAR I: INTEGER;  
373  BEGIN  
374  IF OUT.COL_PTR <> 1  
375  THEN WRITE_CHAR (NL);  
376  FOR I := 1 TO LINES DO WRITE_CHAR (NL);  
377  OUT.COL_PTR := 1;  
378  WRITE_BUF (FALSE)  
379  END;  
380  
381  "SPECIAL CASE ROUTINE FOR LONG CONSTANT"
382  
383  PROCEDURE WRITE_LCONST;  
384  VAR LEN, T, N, AND: INTEGER;  
385  BEGIN  
386  WRITE_CHAR ('LNCONST'); WRITE_CHAR ('');  
387  READ_IPL (LEN);  
388  WRITE_INTR (LEN); WRITE_CHAR ('');  
389  WRITE_CHAR ('HEX-VAL:');  
390  WRITE_COL; N := (LEN - 1) DIV 2 + 1;  
391  FOR I := 1 TO N DO BEGIN  
392  READ_IPL (ARG);  
393  IF OUT_COL_PTR + OUT_BUF_PTR - 1 > LINE_LENGTH THEN BEGIN  
394  WRITE_BUF (FALSE); WRITE (NL); OUT_COL_PTR := 1; NEXT_COL;  
395  END;  
396  WRITE_BUF (FALSE);  
397  IF I < N THEN WRITE_CHAR ('', ) ELSE WRITE_CHAR ('');  
398  END;  
399  
400  "COMMON INITIALIZATION"
401  
402  PROCEDURE INITIZE;  
403  BEGIN  
404  WORDS_IN := PAGELENGTH;  
405  PAGES_IN := 1;  
406  PASS_NO := PARAM [10].INT;  
407  OUT_COL_PTR := 1;  
408  OUT_BUF_PTR := 1;  
409  FIRST_COL := 12;  
410  MIN_COL_SEP := 3;  
411  COL_SEP := 3;  
412  LINK := PARAM [2].PTR  
413  END;
PROCEDURE WRITE_TEXT (TEXT: LINE); VAR I: INTEGER; S: STRING
BEGIN I := 1;
WHILE TEXT[I] <> ^DO BEGIN
WRITE(TEXT[I]); I := I + 1;
END;
END;

PROCEDURE WRITE_HEADER(); VAR I, J: INTEGER;
C: CHAR;
BEGIN C := CHR(PASS_NO+ORD('^'));
WRITE (C); WRITE(C); WRITE(C); WRITE(C); WRITE(C);
FOR I:=1 TO LINE_LENGTH DO WRITE(C);
WRITE(C); WRITE(C); WRITE(C);
FOR I:=2 TO J-1 DO WRITE(' ');
WRITE_TEXT ('HEPASCAL INTERMEDIATE CODE PASS ^');
WRITE(C); WRITE(C);
FOR I:=I+33 TO LINE_LENGTH-1 DO WRITE(' ');
WRITE(C); WRITE(C); WRITE(C);
FOR I:=I+1 TO LINE_LENGTH DO WRITE(C);
WRITE(C); WRITE(C); WRITE(C);
END;

PROCEDURE WRITE_S_CASE (OP_NAME: CHAR); VAR I: INTEGER;
MIN, MAX, MAX_MINUS_MIN, LOCATION, STMT_LABEL: INTEGER;
BEGIN WRITE_CHAR(OP_NAME);
WRITE_CHAR(' '); READ_IFL(MIN);
WRITE_INT(MIN);
WRITE_CHAR(' '); READ_IFL(MAX);
WRITE_INT(MAX);
MAX_MINUS_MIN := MAX - MIN;
FOR I := 0 TO MAX_MINUS_MIN DO BEGIN
WRITE_CHAR(' '); READ_IFL(STMT_LABEL);
WRITE_INT(STMT_LABEL);
END;
WRITE_CHAR(' '); WRITE_RUP(TRUE);
END;
480 PROCEDURE DUMP_TABLE (TABLE: TABLEPTR; ENTRIES: INTEGER);
482 CONST FLDSIZE = 6;
483 VAR I, J, K: INTEGER; PORTION: TABLEPTR;
484 CONST: BOOLEAN;
485 BEGIN
486 CONST : = TABLE = LINK@TABLE@.CONSTANT;
487 PORTION := TABLE;
488 I := ENTRIES - MAXWORD;
489 WITH PORTION do
490 BEGIN
491 WHILE I > 0 DO
492 BEGIN
493 IF CONST
494 THEN FOR J := 1 TO MAXWORD DO
495 BEGIN
496 K := CONTENTS(J);
497 WRITE_IN(K DIV 256, FLDSIZE);
498 WRITE_BUF(TRUE);
499 WRITE_IN(K MOD 256, FLDSIZE);
500 WRITE_BUF(TRUE);
501 END;
502 ELSE FOR J := 1 TO MAXWORD DO
503 BEGIN
504 WRITE_IN(CONTENTS(J), FLDSIZE);
505 WRITE_BUF(TRUE);
506 END;
507 I := I - MAXWORD;
508 PORTION := NEXT_PORTION;
509 BEGIN
510 I := MAXWORD + 1; "I IS NEGATIVE AT THIS POINT"
511 IF CONST
512 THEN FOR J := 1 TO I DO
513 BEGIN
514 K := CONTENTS(J);
515 WRITE_IN(K DIV 256, FLDSIZE);
516 WRITE_BUF(TRUE);
517 WRITE_IN(K MOD 256, FLDSIZE);
518 WRITE_BUF(TRUE);
519 END;
520 ELSE FOR J := 1 TO I DO
521 BEGIN
522 WRITE_BUF(TRUE);
523 WRITE_BUF(TRUE);
524 END;
525 END;
526 END;
527 END;
528 END;
529 PROCEDURE PASS5;
530 CONST MIN_OPS = 0; MAX_OPS = 49;
531 CONST
532 PASS5_TABLE = {
533 "PUSHCHAR", 1, "PUSHVAR", 3, * 1"
534 "PUSHIND", 1, "PUSHADDR", 2,
535 "FIELD", 1, "INDEX", 4,
536 "POINTER", 0, "VARIANT", 7,
537 "VALUE", 2, "ASSIGN", 1, * 9"
538 "ASSIGN", 0, "COPIES", 1,
539 "NEW", 0, "HOT", 0
540 }
570 'AND', 1, 'OR', 1,
571 'NEG', 1, 'AND', 1,
572 'DUB', 1, 'MIN', 1, "19"
573 'DIV', 1, 'MOD', 1,
574 'INVALID', 0, 'INVALID', 0,
575 'FUNCTION', 2, 'NULLSET', 0,
576 'COMPARE', 2, 'CMPTRUC', 2,
577 'FUNCTION1', 2, 'DEFLABEL', 1, "29"
578 'JUMP', 1, 'FAILJUMP', 1,
579 'CASEJUMP', 0, 'INITVAR', 0,
580 'CALL', 3, 'ENTER', 5,
581 'RETURN', 1, 'POP', 1,
582 'NEWLINE', 1, 'ERROR', 0, "39"
583 'LNOCONST', 0, 'MESSAGE', 2,
584 'INCREMENT', 0, 'DECREMENT', 0,
585 'PROCEDURE', 1, 'INIT', 4,
586 'PUSHLABEL', 1, 'CALLFROM', 0, "47"
587 'EDN', 1, 'EOPITOS', 0, "99"
588
589 ARRTAY[MlNOPS .. MAXOPS] OF
590 RECORD
591 OF_NAME: CHAR8;
592 NUM_ARGS: BYTE
593 END;
594
595 CONST
596 EDMS = 88;
597 NEWLINES = 36;
598 CASEJUMP = 32;
599 LONGCONSTANTS = 40;
600
601 BEGIN
602 IN_FILE := 2;
603 REPEAT
604 WITH_PASS_TABLE[OP] DO
605 IF (OP < MINOPS) OR (OP > MAXOPS)
606 THEN INDEXERROR ('BAD_OP', OP)
607 ELSE IF OP = NEWLINES
608 THEN NEWLINE
609 ELSE IF OP = CASEJUMP
610 THEN WRITE5_CASE (OF_NAME)
611 ELSE IF OP = LNOCONST
612 THEN WRITE1CONST
613 ELSE WRITE_OP (OF_NAME, NUM_ARGS, 0, 0, 0, 0)
614 UNTIL OP = EDMS;
615 WRITE (NL);  
616 END;
617
618 PROCEDURE PASS6;
619 CONST MINOPS = 0; MAXOPS = 255;
620 CONST PASS6_TABLE : (  
621 'SLDC00', 0, 'SLDC01', 0, "001"
622 'SLDC02', 0, 'SLDC03', 0,
623 'SLDC04', 0, 'SLDC05', 0, "005"
624 'SLDC06', 0, 'SLDC07', 0,
ARRAY [MIN_OP6..MAX_OP6] OF T91
RECORD
OF_NAME: CHAR8;
NOH_ARGS: BYTE;
END;

CONST EOM6 = 233;
NJFL = 139;
FJFL = 213;

NEWLINES = 234;
INTFL = 6;

VAR SEG6, SEGNO, SEGNOS, ARG1: INTEGER;

BEGIN

"SCAN CONCURRENT SEGMENT"

IN_FILE:=1;
SKIP(1); WRITE_TEXT('** CONCURRENT SEGMENT **'); SKIP(1);
REPEAT
READ_IFL(OP);
WITH PASS6_TABLE(OP) DO
IF (OP < MIN_OP6) OR (OP > MAX_OP6)
THEN THINCORRECT ('BAD-OP ', OP)
ELSE IF OP = NEWLINES
THEN NEW_LINE
ELSE BEGIN
WRITE_OP (OF_NAME, NOH_ARGS, 0, 0, 0, 0);
IF (OP=UJFL) OR (OP=FJFL)
THEN BEGIN
READ_IFL(ARG1);
WRITE_CHAR('LOC=');
WRITE_INT(ARG1); WRITE_CHAR('**');
WRITE_BUF(TRUE);
END
UNTIL OP = EOM6;

"SCAN INTERFACE SEGMENT"

IN_FILE := 4; WORDS_IN := PAGELENGTH; PAGES_IN := 1;
SKIP(2);
WRITE_TEXT('** INTERFACE SEGMENT**');
SEG6 := LINC6, INTERFACE6, INTERFACES6;
SKIP(0);
IF SEG6 = 0
THEN WRITE_CHAR('**...NONE**')
ELSE BEGIN
WRITE_CHAR('**SEGMENTS**'); WRITE_INT(SEGS) END;
SKIP(1);
SEDNO := 1;
WHILE SEGDNO < SEG6 DO
BEGIN
SEGNOS := LINC6, INTERFACE6, INTERFACE6[SEGDNO];
WRITE_TEXT('** SEGMENT NO. **'); WRITE_INT(SEGNOS);
WRITE_BUF(TRUE); NEXT_COL;
WRITE_TEXT('**NO. OF ROUTINES: **'); WRITE_INT(SEGNOS); SKIP(1);
READ_IFL(ARG1); WRITE_CHAR('**SEGDNO**');
END

A-36
780 WRITE_INT(ARG); WRITE_NUP(TRUE);
781 READ_IFL(ARG); WRITE_CHAR('MININDEX=');
782 WRITE_INT(ARG); WRITE_NUP(TRUE);
783 READ_IFL(ARG); WRITE_CHAR('MAXINDEX=');
784 WRITE_INT(ARG); WRITE_NUP(TRUE);
785 SKIP(0);
786 WRITE_TEXT('CASEUMP OFFSETS:'); WRITE_NUP(TRUE);
787 FOR I := 1 TO SEGMENTS DO
788   BEGIN
789 END;
790 READ_IFL(ARG);
791   WRITE_INT(ARG); WRITE_NUP(TRUE);
792   SKIP(0);
793 READ_IFL(ARG); WRITE_CHAR('EXIT-IC=');
794   WRITE_INT(ARG); WRITE_NUP(TRUE);
795 END;
796 WRITE_INT(ARG); WRITE_NUP(TRUE); NEXT_COL;
797 READ_IFL(ARG); WRITE_CHAR('BAYANIZE=');
798   WRITE_INT(ARG);
799   SKIP(0);
800 FOR I := 1 TO (4 + *SEGMENTS + 2) DO
801   BEGIN
802   READ_IFL(OP);
803   WITH PASSN_TABLE[OP] DO
804     BEGIN
805       IF (OP < NUM_OPS) OR (OP > MAX_OPS) THEN ERROR('BAIL-OF *', OP)
806     ELSE WRITE_OF (OP_NAME, NUM_ARGS, 0, 0, 0, 0)
807     END;
808   SKIP(0);
809 END;
810 READ_IFL(ARG); WRITE_CHAR('RTN-PTR=');
811   WRITE_INT(ARG); WRITE_NUP(TRUE);
812 READ_IFL(ARG); WRITE_CHAR('LO-ORDER=');
813   WRITE_INT(ARG); WRITE_NUP(TRUE);
814 WRITE_INT(ARG MOD 256); WRITE_NUP(TRUE); NEXT_COL;
815 WRITE_CHAR('HI-ORDER='); WRITE_INT(ARG DIV 256);
816   SKIP(2);
817 SEGNO := SWCC(SEGNO)
818 END;
819 SKIP(2); WITH LINK# DO
820 BEGIN
821 PRINT SOME OF THE PASSLINK FIELDS:
822   SKIP(1); WRITE_TEXT('*** PASSLINK FIELDS ***'); SKIP(0);
823 WRITE_CHAR('LABELS'); WRITE_INTR(LABELS, SFIELD);
824     END;
825 WRITE_CHAR('BLOCKS'); WRITE_INTR(BLOCKS, SFIELD);
826 SKIP(0);
827 WRITE_TEXT('CONSTANTS**'); WRITE_INTR(CONSTANTS, SFIELD);
828   END;
829 WRITE_CHAR('WORDS'); WRITE_INTR(WORDS, SKIP(0);
830   END;
831 WRITE_TEXT('UPL_OFFSETS**'); WRITE_INTR(UPL_OFFSETS, SFIELD);
832 WRITE_CHAR('SPL_OFFSETS'); WRITE_INTR(SPL_OFFSETS, SFIELD);
833   END;
834 WRITE_CHAR(''); WRITE_CHAR(''); WRITE_CHAR(''); WRITE_CHAR(''); SKIP(0);
835 WRITE_TEXT('XEDISTANCE**'); WRITE_INTR(XEDISTANCE, SFIELD);
836 WRITE_CHAR(''); WRITE_CHAR(''); WRITE_CHAR(''); WRITE_CHAR('');
837 PRINT CONTENTS OF THE TABLES PASSED THRU HEAP:
838   SKIP(2); WRITE_TEXT('*** TABLES ***');
840  SKIP(1); WRITE_CHAR('CONSTWTs'); SKIP(0);
841  IF CONSTANTS > 0 372 229 372
842  THEN DUMP_TABLE(TABLES8,CONSTTABLE, CONSTANTS) 881 89 68 87
843  ELSE WRITE_CHAR(......NONE......) 229
844  ELSE WRITE_CHAR(......NONE......) 229
845  SKIP(2); WRITE_CHAR('XJPTABLE'); SKIP(0);
846  IF XJP_OFFSETS > 0 372 229 372
847  THEN DUMP_TABLE (TABLES8,XJPTABLE, XJP_OFFSETS) 881 89 68 87
848  ELSE WRITE_CHAR(......NONE......) 229
849  ELSE WRITE_CHAR(......NONE......) 229
850  SKIP(2); WRITE_CHAR('XJMTABLE'); SKIP(0);
851  IF LABELS > 0 372 229 372
852  THEN DUMP_TABLE (TABLES8,XJMTABLE, LABELS) 881 89 68 87
853  ELSE WRITE_CHAR(......NONE......) 229
854  ELSE WRITE_CHAR(......NONE......) 229
855  SKIP(2); WRITE_CHAR('EXIT-IC'); SKIP(0);
856  IF BLOCKS > 0 372 229 372
857  THEN DUMP_TABLE (TABLES8,EXITTABLE, BLOCKS) 881 89 69 87
858  ELSE WRITE_CHAR(......NONE......) 229
859  ELSE WRITE_CHAR(......NONE......) 229
860  SKIP(2); WRITE_CHAR('DATASIZE'); SKIP(0);
861  IF BLOCKS > 0 372 229 372
862  THEN DUMP_TABLE (TABLES8,DATASIZE TABLE, BLOCKS) 881 89 69 87
863  ELSE WRITE_CHAR(......NONE......) 229
864  ELSE WRITE_CHAR(......NONE......) 229
865  END;
866  FOR I := 1 TO PAGELENGTH + 10 DO WRITE_CHAR(' ')
867  END;
868  END;
869
870
871
872
873
874  PROCEDURE PASTT;
875  VAR
876  CONST
877  MIN_OPT = 1; MAX_OPT = 1;
878  PASTT_TABLE = ('NULLOP ' , 0);
879  ARRAY [MIN_OPT...MAX_OPT] OF
880  RECORD
881  GP_NAME: CHAR;
882  NUM_ARGS: RST;
883  END;
884
885
886  VAR
887  I: INTEGER;
888
889  BEGIN
890  IN_FILE := 3;
891  COL_SEP := 1;
892  FOR I := 1 TO PAGELENGTH + 10 DO WRITE_CHAR(' ');
893  END;
894
895
896
897  BEGIN
898  INITIALIZE;
899  WRITE_HEADER;
<table>
<thead>
<tr>
<th>CROSS REFERENCE</th>
<th>15 DEF</th>
<th>15 A50</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>255*</td>
<td>264*</td>
<td>268</td>
</tr>
<tr>
<td><strong>ABS</strong></td>
<td>26*</td>
<td>278</td>
</tr>
<tr>
<td><strong>ACCEPT</strong></td>
<td>126*</td>
<td></td>
</tr>
<tr>
<td><strong>ADDR</strong></td>
<td>31*</td>
<td></td>
</tr>
<tr>
<td><strong>ALIGN</strong></td>
<td>210*</td>
<td>213</td>
</tr>
<tr>
<td><strong>ARG</strong></td>
<td>50*</td>
<td>133*</td>
</tr>
<tr>
<td><strong>ARGLIST</strong></td>
<td>105*</td>
<td>145*</td>
</tr>
<tr>
<td><strong>ARGSEQ</strong></td>
<td>107*</td>
<td>133*</td>
</tr>
<tr>
<td><strong>ARGTAG</strong></td>
<td>93*</td>
<td>97</td>
</tr>
<tr>
<td><strong>ARGTYPE</strong></td>
<td>96*</td>
<td>105</td>
</tr>
<tr>
<td><strong>ARG_NO</strong></td>
<td>310*</td>
<td>313*</td>
</tr>
<tr>
<td><strong>ARG_VALUE</strong></td>
<td>310*</td>
<td>318*</td>
</tr>
<tr>
<td><strong>ASCII</strong></td>
<td>27</td>
<td></td>
</tr>
<tr>
<td><strong>ATTR</strong></td>
<td>136*</td>
<td></td>
</tr>
<tr>
<td><strong>BACKSPACE</strong></td>
<td>1*</td>
<td></td>
</tr>
<tr>
<td><strong>BADINDEX</strong></td>
<td>289*</td>
<td>300</td>
</tr>
<tr>
<td><strong>BLOCK</strong></td>
<td>118*</td>
<td>119*</td>
</tr>
<tr>
<td><strong>BLOCKS</strong></td>
<td>87*</td>
<td>82A</td>
</tr>
<tr>
<td><strong>BOOLO</strong></td>
<td>98</td>
<td></td>
</tr>
<tr>
<td><strong>BOOLEAN</strong></td>
<td>32</td>
<td>98</td>
</tr>
<tr>
<td><strong>BOOLETYPE</strong></td>
<td>98</td>
<td>98</td>
</tr>
<tr>
<td><strong>BYTE</strong></td>
<td>248</td>
<td>562</td>
</tr>
<tr>
<td><strong>BITES</strong></td>
<td>248</td>
<td>293</td>
</tr>
<tr>
<td><strong>BITES_PER_IN</strong></td>
<td>17*</td>
<td></td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>113*</td>
<td>114*</td>
</tr>
<tr>
<td><strong>CALLER</strong></td>
<td>111</td>
<td></td>
</tr>
<tr>
<td><strong>CALLERCFG</strong></td>
<td>3T</td>
<td></td>
</tr>
<tr>
<td><strong>CASEMINC</strong></td>
<td>568*</td>
<td>580</td>
</tr>
<tr>
<td><strong>CHAR</strong></td>
<td>20</td>
<td>23</td>
</tr>
<tr>
<td><strong>CHARB</strong></td>
<td>155*</td>
<td>229</td>
</tr>
<tr>
<td><strong>CHARX</strong></td>
<td>239</td>
<td>240</td>
</tr>
<tr>
<td><strong>CLOSE</strong></td>
<td>117*</td>
<td></td>
</tr>
<tr>
<td><strong>CODEEJECT</strong></td>
<td>111</td>
<td></td>
</tr>
<tr>
<td><strong>COL_SEP</strong></td>
<td>171*</td>
<td>817*</td>
</tr>
<tr>
<td><strong>COMPLETE</strong></td>
<td>44*</td>
<td></td>
</tr>
<tr>
<td><strong>COND</strong></td>
<td>27</td>
<td></td>
</tr>
<tr>
<td><strong>CONST</strong></td>
<td>48*</td>
<td>486*</td>
</tr>
<tr>
<td><strong>CONSTANTS</strong></td>
<td>87*</td>
<td>921</td>
</tr>
<tr>
<td><strong>CONTENTS</strong></td>
<td>63*</td>
<td>896*</td>
</tr>
<tr>
<td><strong>CONTEXT</strong></td>
<td>309*</td>
<td>338</td>
</tr>
<tr>
<td><strong>CONTROL</strong></td>
<td>39</td>
<td></td>
</tr>
<tr>
<td><strong>CRC</strong></td>
<td>13*</td>
<td></td>
</tr>
<tr>
<td><strong>DATASIZED</strong></td>
<td>69*</td>
<td>863</td>
</tr>
<tr>
<td><strong>DEVICE</strong></td>
<td>139*</td>
<td>141*</td>
</tr>
<tr>
<td><strong>DISKDEVICE</strong></td>
<td>37</td>
<td></td>
</tr>
<tr>
<td><strong>DISPLAY</strong></td>
<td>122*</td>
<td></td>
</tr>
<tr>
<td><strong>DUMP_TABLE</strong></td>
<td>68*</td>
<td>843*</td>
</tr>
<tr>
<td>Variable</td>
<td>Length</td>
<td>Offset</td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>IN_FILE</td>
<td>512</td>
<td></td>
</tr>
<tr>
<td>HOARD</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>J0VICE</td>
<td>141</td>
<td></td>
</tr>
<tr>
<td>J0MOVE</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>J0PARM</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>J0RESULT</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>J0TRANSF</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>J0-</td>
<td>512</td>
<td>514</td>
</tr>
<tr>
<td>J0TASK</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>J0UNTABLE</td>
<td>853</td>
<td></td>
</tr>
<tr>
<td>K-</td>
<td>497</td>
<td>499</td>
</tr>
<tr>
<td>KIND</td>
<td>514</td>
<td>515</td>
</tr>
<tr>
<td>KIND_ARG</td>
<td>320</td>
<td></td>
</tr>
<tr>
<td>L-</td>
<td>852</td>
<td>853</td>
</tr>
<tr>
<td>LEN</td>
<td>307</td>
<td>390</td>
</tr>
<tr>
<td>LENGTH</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>LFIELD</td>
<td>162</td>
<td></td>
</tr>
<tr>
<td>LINE</td>
<td>125</td>
<td>131</td>
</tr>
<tr>
<td>LINELEN</td>
<td>377</td>
<td></td>
</tr>
<tr>
<td>LINES</td>
<td>418</td>
<td>486</td>
</tr>
<tr>
<td>LINK_LEN</td>
<td>775</td>
<td>825</td>
</tr>
<tr>
<td>LOCATION</td>
<td>569</td>
<td></td>
</tr>
<tr>
<td>LONGCONST</td>
<td>562</td>
<td></td>
</tr>
<tr>
<td>LOOKUP</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td>H-</td>
<td>467</td>
<td>468</td>
</tr>
<tr>
<td>HAZARG</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>HAZMAP</td>
<td>81</td>
<td></td>
</tr>
<tr>
<td>HAZWORD</td>
<td>50</td>
<td>448</td>
</tr>
<tr>
<td>HAZ_CONTEXT</td>
<td>139</td>
<td></td>
</tr>
<tr>
<td>HAZ_KIND</td>
<td>320</td>
<td></td>
</tr>
<tr>
<td>HAL_MINUS</td>
<td>471</td>
<td></td>
</tr>
<tr>
<td>MAXMODE</td>
<td>427</td>
<td></td>
</tr>
<tr>
<td>MAX_OPS</td>
<td>576</td>
<td></td>
</tr>
<tr>
<td>MAX_OP7</td>
<td>880</td>
<td></td>
</tr>
<tr>
<td>MAX_TYPE</td>
<td>320</td>
<td></td>
</tr>
<tr>
<td>HBTRIM</td>
<td>332</td>
<td></td>
</tr>
<tr>
<td>HBTRIM</td>
<td>332</td>
<td></td>
</tr>
<tr>
<td>HIM</td>
<td>465</td>
<td>467</td>
</tr>
<tr>
<td>HIM_COLSEP</td>
<td>416</td>
<td></td>
</tr>
<tr>
<td>HIM_CONTEXT</td>
<td>139</td>
<td></td>
</tr>
<tr>
<td>HIM_KIND</td>
<td>322</td>
<td></td>
</tr>
<tr>
<td>HIM_MODE</td>
<td>334</td>
<td></td>
</tr>
<tr>
<td>HIM_OPS</td>
<td>576</td>
<td></td>
</tr>
<tr>
<td>HIM_OP7</td>
<td>880</td>
<td></td>
</tr>
<tr>
<td>HIM_TYPE</td>
<td>320</td>
<td></td>
</tr>
<tr>
<td>MNLARG</td>
<td>332</td>
<td></td>
</tr>
<tr>
<td>MOVE</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>-M-</td>
<td>167</td>
<td>186</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>WRITE</td>
<td>114</td>
<td>207</td>
</tr>
<tr>
<td>WRITEARG</td>
<td>131</td>
<td></td>
</tr>
<tr>
<td>WRITEJOIN</td>
<td>81</td>
<td></td>
</tr>
<tr>
<td>WRITEPAGE</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>WRITE_S_CASE</td>
<td>452</td>
<td>581</td>
</tr>
<tr>
<td>WRITE_HNP</td>
<td>210</td>
<td>222</td>
</tr>
<tr>
<td>WRITE_CVAR</td>
<td>319</td>
<td>331</td>
</tr>
<tr>
<td>WRITE_CHAR</td>
<td>316</td>
<td>317</td>
</tr>
<tr>
<td>WRITE_CHARB</td>
<td>223</td>
<td>232</td>
</tr>
<tr>
<td>WRITE_HEADER</td>
<td>432</td>
<td>437</td>
</tr>
<tr>
<td>WRITE_HEX</td>
<td>281</td>
<td>293</td>
</tr>
<tr>
<td>WRITE_HEX_C</td>
<td>237</td>
<td>253</td>
</tr>
<tr>
<td>WRITE_INT</td>
<td>258</td>
<td>300</td>
</tr>
<tr>
<td>WRITE_INTR</td>
<td>271</td>
<td>365</td>
</tr>
<tr>
<td>WRITE_LCONST</td>
<td>388</td>
<td>589</td>
</tr>
<tr>
<td>WRITE_OP</td>
<td>388</td>
<td>584</td>
</tr>
<tr>
<td>WRITE_TRN</td>
<td>423</td>
<td>444</td>
</tr>
</tbody>
</table>

**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 BLOCKDUMP:

(* PROGRAM TO DISPLAY (ON SCREEN OR PRINTER) THE CONTENTS 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 DISPLAYED AS ASCII VALUES UNDER THE CHARACTER REPRESENTATIONS. 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 = 12;
LF = 10;
BLLENGTH = 512; (* BYTES IN DISK BLOCK *)
LINELEN = 80; (* CHAR PER LINE *)
MAXLINES = 22; (* LINES PER SCREEN PAGE *)
MAXBLOCK = 987; (* MAX BLOCK # ON 8" DUAL DENIS 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;
OUTFLNAME: 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] := CHR(TEMP + ORD0)
      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;
FRTOPT := 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 ('PRINT OR SCREEN OUTPUT, OR QUIT?');
REPEAT
  WRITELN ('ENTER P, S, OR Q');
  READLN(OPT).
CASE OPT OF
  'P': BEGIN
    OK := TRUE; FRTOPT := 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+2.2+2.3+4.4+5.5
    END;
  'Q': BEGIN
    OK := TRUE; QUIT := TRUE
    END
END
UNTIL OK;
IF NOT QUIT
THEN BEGIN
RESET (OUTFL, OUTFLNAM);
FOR POSIT := 1 TO LINELEN DO CHARLINE[POSIT] := ' ';
FOR BLOCK := STARTBL TO (STARTBL+LENGTH-1) DO
BEGIN
UNITWAIT (DISK);
UNITREAD (DISK, BLOCKIN, BLENGTH, BLOCK);
WRITEN (OUTFL);
WRITEN (OUTFL, 'UNIT ', DISK, ' ', BLOCK, ' ', BLOCK);
WRITEN (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 BLENGTH 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; (* P. 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 PRIOPR THEN WRITE (OUTFL, CHR(CR));
WRITE (OUTFL, NUMLINE);
IF PRIOPR 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

METHASCAL 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 after the last packet of good data, so its data portion contains garbage and can be ignored. Byte 516 is a checksum on the data bytes only. The formula is the sum of the data bytes, modulus 256:

\[ \text{SUM} := (\text{SUM} + \text{BYTE}) \mod 256. \]

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 entire 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.
"FILEXFER - ASCII LINE FILE TRANSFER PROGRAM"
"WRITTEN BY ROBERT YOUNG"
"INCLUDE SVC_TYPES"
"INCLUDE FDL1, PREFIX"

PROGRAM FILEXFER;

CONST
RST=-1; ACK=6; NAK=21;

TYPE FILE_MODE = (BYTE_MODE, PAGE_MODE, ASC_MODE);

TYPE PAGE = ARRAY [1..512] OF BYTE;

VAR
BUF: RECORD
  DATA: PAGE;
  EOF: SHORTINTEGER;
  CKSUM: SHORTINTEGER;
END;

IN_MODE, OUT_MODE: FILE_MODE;
FM_FOUND: BOOLEAN;
BLOCKS: INTEGER;

PROCEDURE SVC1 (VAR PARN: SVC1 BLOCK); EXTERN;

PROCEDURE SVC2 (VAR PARN: SVC2 BLOCK); EXTERN;

PROCEDURE SVC2PA; EXTERN;

"DISPLAY ROUTINES"

PROCEDURE DISPLAY_TEXT (TEXT: LINK);

VAR J: INTEGER;

BEGIN
  30 J:=1;
  31 WHILE TEXT[J]<> '"' DO BEGIN
    32 DISPLAY(TEXT[J]); J:=J+1;
  33 END;

PROCEDURE DISPLAY_HEX_CHAR (VAL: INTEGER);

BEGIN
  38 IF VAL>9 THEN DISPLAY(CHR((VAL-10)+ORD('A')))
  39 ELSE DISPLAY(CHR((VAL-48)+ORD('0')));
END;

PROCEDURE DISPLAY_HEX_CHARS (VAL: INTEGER);

BEGIN
  42 DISPLAY_HEX_CHAR(VAL DIV 16);
  45 DISPLAY_HEX_CHAR(VAL MOD 16);
END;

PROCEDURE DISPLAY_HEX (VAL: INTEGER; N: INTEGER);

TYPE TAGS = (TAG1, TAG2);

VAR I,J: INTEGER;

1: RECORD CASE TAGS OF
  52 TAG1: (INT: INTEGER);
    53 TAG2: (BYTES: ARRAY [1..N] OF BYTE);
END;

BEGIN
  55 X.INT:=VAL;
  57 FOR I:=5 TO N DO
    DISPLAY_HEX_CHARS(X.BYTES[I]);
END;
"INITIALIZATION"

PROCEDURE INIT_FILE (LU: INTEGER; VAR NMOD: FILE_MODES);
  VAR SVCT_PARM: SVCT_BLOCK;
  BEGIN
   WITH SVCT_PARM DO BEGIN
   SVCT_LU := LU; SVCT_MOD := SVCT_FETCH_ATTR;
   SVCT(SVCT_PARM) := 22 64
   IF SVCT_STAT <> 0 THEN BEGIN
     DISPLAY_TEXT(SVCT ERROR $1);
     DISPLAY_HEX(SVCT_STAT,1);
     DISPLAY_TEXT(' ON LU $41');
     DISPLAY_HEX(SVCT_LU,1);
     DISPLAY(NL);
     SVCT_MOD := 0; SVCT RECLEM := 0;
   END;
   IF SVCT MOD = 200 THEN MODE := ASC_MODE
   ELSE IF (SVCT RECLEM = 0) OR (SVCT RECLEM > 512) THEN MODE := PAGE_MODE
   ELSE MODE := BITMAP_MODE;
  END;
END;

"ERROR LOGGING"

PROCEDURE SVC1 ERROR LOG (SVCT_PARM: SVC1 BLOCK);
  BEGIN
   WITH SVCT_PARM DO BEGIN
   DISPLAY_TEXT(SVCT ERROR $1);
   DISPLAY_HEX(SVCT1_STAT,1);
   DISPLAY_HEX(SVCT1_DEV_STAT,1);
   DISPLAY_TEXT(' ON LU $41');
   DISPLAY_HEX(SVCT1_LU,1);
   DISPLAY_TEXT(' FUNC $41');
   DISPLAY_HEX(SVCT1_FUNC,1);
   DISPLAY_TEXT(' XLEN $41');
   DISPLAY_HEX(SVCT1_XLEN,2);
   DISPLAY_TEXT(' BLOCK $41');
   DISPLAY_HEX(BLOCKS,2);
   DISPLAY(NL);
  END;
END;

"ASC INPUT PROCESSING"

PROCEDURE READ ASC (VAR EOF: BOOLEAN);
  VAR SVC1 PARM: SVC1 BLOCK;
  I, J: SHORT INTEGER;
  REPL1: BYTE;
  ERRORS: INTEGER;
  BEGIN
   ERRORS := 0;
   WITH SVCT_PARM DO REPEAT
   SVCT_LU := 1; SVCT FUNC := SVC1 READ; SVC1 IMAGE := SVC1 WAIT;
   SVC1 DUPSTART := ADDRESS(DUP);
   SVC1 DUPEND := SVC1 DUPSTART + 15;
   SVC1(PARM);
   IF SVCT STAT <> 0 THEN BEGIN
   REPL1 := RAK;
   SVC1_ERROR LOG(SVCT_PARM);
   END
END;

A-88
120   END
121   ELSE   REPLY:=ACK;
122   J:=o;
123   FOR I:=1 TO 512 DO
124      J:=J+BUF.DATA[I];
125      J:=J MOD 256;
126      IF ((J MOD 256)<BUF.CKSUM) AND (REPLY=ACK) THEN BEGIN
127         DISPLAY_TEXT('CHECKSUM ERROR!');
128         DISPLAY_TEXT(16#0f#);  DISPLAY_HEX(J,2);
129         DISPLAY_TEXT(16#0c#);  DISPLAY_HEX(BUF.CKSUM,2);
130         DISPLAY(16#10#);
131         REPLY:=ACK;
132      END;
133      SVC1_FUNC:=SVC1_WRITE+SVC1_IMAGE+SVC1_WAIT;
134      SVC1_BUFEND:=ADDRESS(REPLY);
135      SVC1_BUFSTART:=BUF.BUFSTART;
136      IF SVC1_STATUS=BUF then SVC1_ERROR_LOG(SVC1_PARM);
137      IF REPLY=NO THEN ERRORS:=ERRORS+1;
138      IF ERRORS>16 OR (REPLY=ACK);
139      IF BUF.EOF = EOF THEN EOF:=TRUE ELSE EOF:=FALSE;
140      IF REPLY=ACK THEN EOF:=TRUE;  "FORCE ABORT"
141      END;
142      END;
143
144 **ASC OUTPUT**
145
146   PROCEDURE WRITE_ASC (EOF: BOOLEAN);
147   VAR   REPLY: BYTE;
148      SVC1_PARM; SVC1_BLOCK;
149      I: INTEGER;
150      ERRORS: INTEGER;
151      OK: BOOLEAN;
152      BEGIN
153      ERRORS:=0;
154      BUF.CKSUM:=0;
155      FOR I:=1 TO 512 DO
156         BUF.CKSUM:=BUF.CKSUM+(BUF.DATA[I]);
157      BUF.CKSUM:=BUF.CKSUM MOD 256;
158      IF EOF THEN BUF.EOF:=EOF ELSE BUF.EOF:=0;
159      ERRORS:=0;
160      WITH SVC1_PARM DO REPEAT
161         SVC1_BUF:=SVC1_WRITE+SVC1_IMAGE+SVC1_WAIT;
162         SVC1_BUFEND:=ADDRESS(BUF);
163         SVC1_BUFSTART:=BUF.BUFSTART+515;
164         SVC1_IMAGE:=21184;
165         IF SVC1_STATUS=BUF then BEGIN
166            SVC1_ERROR_LOG(SVC1_PARM);
167            OK:=FALSE;
168         END
169      END;
160      ELSE OK:=TRUE;
170      SVC1_FUNC:=SVC1_READ+SVC1_IMAGE+SVC1_WAIT;
171      SVC1_BUFEND:=ADDRESS(REPLY);
172      SVC1_BUFSTART:=BUF.BUFSTART;
173      SVC1_IMAGE:=21184;
174      IF SVC1_STATUS=BUF then BEGIN
175         SVC1_ERROR_LOG(SVC1_PARM);  OK:=FALSE;
176      END;
177      IF OK AND (REPLY <> ACK) THEN BEGIN
178         DISPLAY_TEXT('NEGATIVE ACKNOWLEDGMENT $#');
179         DISPLAY_HEX(REPLY,1);  DISPLAY(16#10#);
PROCEDURE READ_FILE (VAR EOF: BOOLEAN);  

VAR I: INTEGER; C: CHAR;

BEGIN
  IF IN_MODE = ASC_MODE THEN READ_ASC(EOF)
  ELSE IF IN_MODE = PAGE_MODE THEN READPAGE(MUF_DATA, EOF)
  ELSE BEGIN
    I:=1; EOF:=FALSE;
    IF EN_FOUND THEN EOF:=TRUE
    ELSE WHILE I<512 DO BEGIN
      READ(C); MUF_DATA[I]:=ORD(C);
      IF C = EN THEN BEGIN
        l:=512; EN_FOUND:=TRUE;
        END;
      END;
      I:=I+1;
    END;
  END;
END;

PROCEDURE WRITE_FILE (EOF: BOOLEAN);

VAR I: INTEGER; C: CHAR;

BEGIN
  IF OUT_MODE = ASC_MODE THEN WRITE_ASC(EOF)
  ELSE IF OUT_MODE = PAGE_MODE THEN WRITEPAGE(MUF_DATA, EOF)
  ELSE IF NOT EOF THEN BEGIN
    I:=1;
    WHILE I<512 DO BEGIN
      C:=CHR(MUF_DATA[I]);
      WRITE(C);
      IF C=EN THEN I:=I+1
      END;
    END;
  END;
END;

PROCEDURE RUN;

VAR EOF: BOOLEAN;

BEGIN
  EN_FOUND:=FALSE;
  BLOCKS:=1;
  INIT_FILE(1, IN_MODE);
  INIT_FILE(2, OUT_MODE);
  REPEAT
    READ_FILE(EOF);
    WRITE_FILE(EOF);
    BLOCKS:=BLOCKS+1;
    UNTIL EOF;
  END;
END.
<table>
<thead>
<tr>
<th>CROSRS REFERENCE</th>
<th>* IS DEF</th>
<th>= IS ASCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK</td>
<td>8*</td>
<td>121</td>
</tr>
<tr>
<td>ADDRESS</td>
<td>114*</td>
<td>130</td>
</tr>
<tr>
<td>ASC_MODE</td>
<td>9*</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>19*</td>
<td>98</td>
</tr>
<tr>
<td>BKF</td>
<td>18*</td>
<td>105</td>
</tr>
<tr>
<td>BF</td>
<td>12*</td>
<td>118</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BYTR</td>
<td>10*</td>
<td>53</td>
</tr>
<tr>
<td>BYTT</td>
<td>53*</td>
<td>58</td>
</tr>
<tr>
<td>BYTT_MODE</td>
<td>9*</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>187*</td>
<td>195</td>
</tr>
<tr>
<td>CHAR</td>
<td>187*</td>
<td>205</td>
</tr>
<tr>
<td>CHR</td>
<td>38*</td>
<td>39</td>
</tr>
<tr>
<td>CKSNUM</td>
<td>15*</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATA</td>
<td>13*</td>
<td>126</td>
</tr>
<tr>
<td>DISPLAY</td>
<td>32*</td>
<td>30*</td>
</tr>
<tr>
<td>DISPLAY_HEX</td>
<td>48*</td>
<td>71</td>
</tr>
<tr>
<td>DISPLAY_HEX</td>
<td>36*</td>
<td>42*</td>
</tr>
<tr>
<td>DISPLAY_TEXT</td>
<td>27*</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>196*</td>
<td>214*</td>
</tr>
<tr>
<td>EN FOUND</td>
<td>18*</td>
<td>193*</td>
</tr>
<tr>
<td>ENF</td>
<td>18*</td>
<td>105*</td>
</tr>
<tr>
<td>EOT</td>
<td>8*</td>
<td>140*</td>
</tr>
<tr>
<td>ERRORS</td>
<td>109*</td>
<td>111*</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>170*</td>
<td>167*</td>
</tr>
<tr>
<td>FILEFER</td>
<td>7*</td>
<td></td>
</tr>
<tr>
<td>FILE_MODES</td>
<td>9*</td>
<td>17*</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>28*</td>
<td>30*</td>
</tr>
<tr>
<td>INIT_FILE</td>
<td>63*</td>
<td>225*</td>
</tr>
<tr>
<td>INT</td>
<td>92*</td>
<td>56*</td>
</tr>
<tr>
<td>INTENDER</td>
<td>15*</td>
<td>20*</td>
</tr>
<tr>
<td>INT_MODE</td>
<td>17*</td>
<td>189*</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>50*</td>
<td>107*</td>
</tr>
<tr>
<td>LINE</td>
<td>27*</td>
<td></td>
</tr>
<tr>
<td>LJ</td>
<td>63*</td>
<td>67*</td>
</tr>
<tr>
<td>MODE</td>
<td>63*</td>
<td>77*</td>
</tr>
</tbody>
</table>
PROGRAM X822.HD

CONSTR
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 *)
LONGTIME = (* TEST 1000 *) 200; (* LONG TIMEOUT, ABOUT 5.5 MIN *)
MEDTIME = 126; (* MEDIUM TIMEOUT, ABOUT 30 SEC *)
SHRTIME = 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 25-41) 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 REGISTER 1 *)
RINGREADY = 0;
REGTOSND = 1;
RECENABL = 2;
PARENB = 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 = RINGREADY..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, ENG: CHAR; (* ASCII CONTROL CHAR *)
OK, (* RESPONSE MEANING PACKET WAS REC'D OK *)
NOTOK, (* RESPONSE MEANING PACKET NOT REC'D OK *)
FLAGSBYTE: 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 *)
STARTBYTE: INTEGER; (* BYTE WITHIN BUFFER WHERE THE CURRENT
PACKET (AND BLOCK) STARTS *)
DISKFILE: FILE; (* NAME BY WHICH THIS PDM KNOWS THE OUTPUT FILE *)
OUTFLNAME: 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 (CHAR: CHAR; VAR ABORT: BOOLEAN);
<* WHEN 822 IS READY TO RECEIVE, IT SENDS AN ENQ CHAR. *
<* PROCEDURE WAITS FOR ENQ THEN SENDS RESPONSE MESSAGE OF A *
<* SINGLE CHAR *>)
VAR
  TIME: INTEGER;
  EXPECT: CHAR;
  ABYTE: BYTETYPE;
  ERROR: BOOLEAN;
  TICK: INTEGER;
  HAVECHAR: BOOLEAN;
BEGIN
  ABORT := FALSE;
  TIME := MDTIME;
  ERROR := FALSE;
REPEAT
  TICK := 1000;
  REPEAT (* POLL LINE UNTIL CHAR ARRIVES OR TIME TO DECRED.
  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: BYTE TYPE;
TICK: INTEGER;
HAVECHAR: BOOLEAN;
I, R: INTEGER; (* TEST *)
BEGIN
PKTERROR := FALSE;
ABORT := FALSE;
BYT := STARTBYT;
LASTBYT := STARTBYT + PKTLEN;
LINE_PORTPTR^DATA := ENG;
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];
EXPCTENQ := ABYTE[1];
IF TIME <= 0 THEN BEGIN (* TIMED OUT *)
  WRITELN;
  WRITELN ('PROCEDURE RESPOND:');
  WRITELN ('TIMED OUT WAITING FOR ENQ');
  ABORT := TRUE
END;
ELSE 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;
  CALCCRC: INTEGER; (* THE CALCULATED CHECKSUM *)
  CHARA: CHAR;
BEGIN
  CALCCRC := 0;
  FOR BYTE := STARTBYT TO (STARTBYT+ELLEN-1) DO (*RECALCULATION*)
    BEGIN
      CHARA := BUFF [BYTE];
      CALCCRC := (CALCCRC + ORD(CHARA)) MOD 256
    END;
  CHARA := BUFF [STARTBYT + CRCDIST]; (* GET CRC FROM PACKET *)
  CRC_OK := CALCCRC = 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 := LONGETIME;
   ERROR := FALSE;
REPEAT
   TICK := 1000;
   REPEAT (* FOLL LINE UNTIL CHAR ARRIVES OR TIME TO DEC. 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;
   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 PKERROR := TRUE;
IF TIME <= 0
THEN BEGIN
    WRITELN('PROC READPKT:');
    WRITELN('TIMEOUT BETWEEN PACKETS:');
    ABORT := TRUE
END;
BYT := SUCC(BYT);
R := 0;
REPEAT
    TIME := SHPTIME;
    ERROR := FALSE;
    REPEAT
        TICK := 1000;
        REPEAT ( = POLL LINE UNTIL CHAR ARRIVES OR TIME TO DEC. 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 PKERROR := TRUE;
    BYT := SUCC(BYT);
    R := SUCC(R)
UNTIL (BYT >= LASTBYT) OR (TIME <= 0);

IF TIME <= 0
THEN PKERROR := 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.[LOOPNORM] := TRUE; (** NORMAL (NOT TEST) OPERATION **) CR2.[CLOCK1] := TRUE; (** CLOCK RATE 1 (32X **) CR2.[CLOCK2] := FALSE;
    END;
  END;

*PROCEDURE INIT;
BEGIN
STARTBVT := 1;
BLOCKS := 0;
BLOKSIN := 0;
BLOKSOUT := 0;

ACK := CHR(6); OK := ACK;
NAK := CHR(21); NOTOK := NAK;
END := CHR(5);

WRITELN;
WRITELN (**NAME OF OUTPUT FILE?**);
READLN (OUTFNAM);
REWRT (DISHFILE, OUTFNAM);
IF IORESULT = 0 THEN BEGIN
  WRITELN;
  WRITELN (**PROCEDURE INIT:**);
  WRITELN (**I/O ERROR # / IORESULT**);
  HALT;
END;
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 BUFI[I] := '/';
INIT;
REPEAT
READPKT (STARTBYT, ERROR, FATALERR),
IF CRC_OK (STARTBYT) AND NOT (ERROR OR FATALERR)
THEN BEGIN
IF NOT FATALERR
THEN BEGIN
BLOCKS := SUCQ(BLOCKS);
FLAGBYTE := BUFF [STARTBYT+FLAGDIST];
LASTPKT := FLAGBYTE = EOT;
BEGIN (* TEST IF BLOCKS >= 5 THEN LASTPKT := TRUE; *)
IF (BLOCKS >= BLINBUFF) OR LASTPKT
THEN BEGIN (* BUFFER FULL OR RECEIVED LAST PACKET *)
BLOCKS := 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
"# MODIFIED TO PRODUCE CODE FOR MICROENGINE. #"
"##########################################################################"
"CODE TO BE FINALIZED LATER IS MARKED $$$"
"SRCS 32 := FALSE"
"SRCS 16 := NOT SRC 32"
"*TER DRUNCH HANSEN
11 INFORMATION SCIENCE
12 CALIFORNIA INSTITUTE OF TECHNOLOGY
13 PASADENA, CALIFORNIA 91125
14 FPD 11/5 CONCURRENT/SEQUENTIAL PASCAL
15 COMPILER PASS 6: CODE SELECTION
16 FOR PASCAL MICROENGINE."
17
18 TYPE FULLWORD = INTEGER;
19 TYPE INTEGER = SHORTINTEGER;

21 "### # PREFIX #"
22 "###
23
25 CONST EOL = '\010'; FF = '\012'; EOM = '\025';
26 6 = 6;
27 PREMLIMIT = 18;
28 WORDLENGTH = 2 "BYTES";
29 REALLength = 8 "BYTES";
30 SEQLength = 16 "BYTES";
31 "ON STACK, WORD LENGTH (8) OF SET IS ALWAYS PUSHeD"
32 "AFTER THE SET ITSELF, TO CONFORM TO MICROENGINE CONVENTION."
33 MICRO SUPPORTS VARIABLE-LENGTH SETS, BUT THEY ARE
34 NOT USED BY THIS VERSION OF CONCURRENT PASCAL."
35
36 SOURCE_WORD_LENGTH = "\151IF SRC 16" 2; "\153END"
37 "\151IF SRC 32" 4; "\153END"
38 LISTOPTION = 0; SUMMARYOPTION = 1; TESTOPTION = 2; CHECKOPTION = 3;
39 CODEOPTION = 4; NUMBEROPTION = 5; VARNключение OPTION = 8; "\152VARN"
40 MAXWORD = 100;
41
42 TYPE FILE = ..";
43
44 CONST INLENGTH = 12;
45 TYPE IDENTIFIER = ARRAY (.1..INLENGTH.) OF CHAR;
46
47 TYPE POINTER = \@ INTEGER;
48 TABLEFD = @TABLEFD;
49 TABLE = RECORD
50   NEXTOPTION: TABLEFD;
51   CONTENTS: ARRAY (.1..MAXWORD.) OF INTEGER
52   END;
53 TABLEPART = RECORD
54   SEGDISTANCE, STACKLENGTH: INTEGER;
55   JUMPABLE, CONSTTABLE, XTABLE,
56   EXTABLE, DATAEXTABLE: TABLEFD
57   END;
58 TABLESRC = @TABLEPART;
59 OPTION = LISTOPTION..VARNключне OPTION; "\152VARN"
""
60 const maxintfac = 14;
61 type
62 ifptr = @finfo;
63 info = record
64 interfaces: integer;
65 interfacesizes: array[1..maxintfac] of integer
66 end;
67
68 passptr = i_passlink;
69 passtplink = record
70 options: set of option;
71 labels, blocks, constants, xip_offsets: integer;
72 nextpoint: fullword;
73 table: tablestptr;
74 interface: ifptr
75 end;
76
77 type argtag =
78 (hilitype, booltype, inttype, idtype, ptrtype);
79
80 type argtype = record
81 case tag: argtag of
82   /hilitype := booltype := (bool: boolean);
83   /inttype := inttype := (int: integer);
84   /idtype := idtype := (id: identifier);
85   /ptrtype := ptrtype := (ptr: passtptr)
86 end;
87
88 const maxarg = 10;
89 text_length = 10;
90 type arlist = array[1..maxarg] of argtype;
91 text_type = array[1..text_length] of char;
92 const pagelen = "$if src_16$ 0$if src_32$ 0$end$";
93 $if src_32$ 256; "$end$
94 type page = array[1..page_length] of integer;
95
96 procedure read(var c: char);
97 procedure write(c: char);
98 procedure motused;
99 procedure motused2;
100 function file_length(p: file): integer;
101 function put(f: file; p: integer; var block: unit page);
102 function get(f: file; p: integer; var block: unit page);
103 function put(f: file; p: integer; var block: unit page);
104 function put(f: file; p: integer; var block: unit page);
105 function file_length(p: file): integer;
106 procedure hark(var top: fullword);
107 procedure release(top: fullword);
108 program main(var param: arlist);
109
110 "*****************************************************************************
111 # Pascal code generated by PascalLink#*****************************************************************************
112 #*****************************************************************************
113
114 "input operators"
115 "(Assumption is Never Produced by Pass 5.)"
116 "(InitTag is Never Produced by Pass 5.)"
NO MICROEGINE OPERATION COMES FROM 64 TO 119, * 

SIMD12 = 12; SIMD12 = 121; SIMD12 = 123; 

SIMD22 = 122; SIMD22 = 125; SIMD22 = 126; SIMD22 = 127; 

LDC22 = 128; LDC22 = 129; LDC22 = 130; LDC22 = 131; 

LDC22 = 132; LDC22 = 133; LDC22 = 135; LDC22 = 137; 

LDC22 = 136; LDC22 = 137; LDC22 = 138; LDC22 = 139; 

LDC22 = 140; LDC22 = 141; LDC22 = 142; LDC22 = 143; 

LDC22 = 144; LDC22 = 145; LDC22 = 146; LDC22 = 147; 

LDC22 = 148; LDC22 = 149; LDC22 = 150; LDC22 = 151; 

LDC22 = 152; LDC22 = 153; LDC22 = 154; LDC22 = 155; 

LDC22 = 156; LDC22 = 157; LDC22 = 158; LDC22 = 159; 

LDC22 = 160; LDC22 = 161; LDC22 = 162; LDC22 = 163; 

LDC22 = 164; LDC22 = 165; LDC22 = 166; LDC22 = 167; 

EQUL2 = 176; NEQUL2 = 177; LEQL2 = 178; LEQUL2 = 179; 

LEQL2 = 180; GEQUL2 = 181; EQQUL2 = 182; LEQUL2 = 183; 

LEQUL2 = 184; LEQUL2 = 185; LEQUL2 = 186; LEQUL2 = 187; 

LEQUL2 = 188; LEQUL2 = 189; LEQUL2 = 190; LEQUL2 = 191; 

LEQUL2 = 192; LEQUL2 = 193; LEQUL2 = 194; LEQUL2 = 195; 

LEQUL2 = 196; LEQUL2 = 197; LEQUL2 = 198; LEQUL2 = 199; 

LEQUL2 = 200; LEQUL2 = 201; LEQUL2 = 202; LEQUL2 = 203; 

LEQUL2 = 204; LEQUL2 = 205; LEQUL2 = 206; LEQUL2 = 207; 

LEQUL2 = 208; LEQUL2 = 209; LEQUL2 = 210; LEQUL2 = 211; 

LEQUL2 = 212; LEQUL2 = 213; LEQUL2 = 214; LEQUL2 = 215; 

LEQUL2 = 216; LEQUL2 = 217; LEQUL2 = 218; LEQUL2 = 219; 

LEQUL2 = 220; LEQUL2 = 221; LEQUL2 = 222; LEQUL2 = 223; 

LEQUL2 = 224; LEQUL2 = 225; LEQUL2 = 226; LEQUL2 = 227; 

LEQUL2 = 228; LEQUL2 = 229; LEQUL2 = 230; LEQUL2 = 231; 

LEQUL2 = 232; LEQUL2 = 233; LEQUL2 = 234; 

NO MICROEGINE OPERATION COMES FROM 168 TO 175. * 

NO MICROEGINE OPERATION COMES FROM 231. * 

NO MICROEGINE OPERATION COMES > 231. * 

MESSAGE = 232; EOM = 233; NEWLN = 234; 

OPERATORS 232, 233, 234 ARE VIRTUAL OPS TO BE REMOVED IN FUTURE. * 

OTHER CONSTANTS * 

MACHINE REGISTERS (NOTE: POSITIVE REGS ARE FIELDS IN ACTIVE TB) * 

READYQ_REG = -3; "POINTS TO READY QUEUE" 

SUBC_DICT_REG = -2; "POINTS TO SUBDIRECT DICTIONARY" 

TASK_REG = -1; "POINTS TO RUNNING TASK'S TB" 

GLINK_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" 

BASE_REG = 5; "LOCAL BASE" 

GRASE_REG = 6; "GLOBAL BASE" 

PC_REG = 7; "PROGRAM COUNTER" 

SEQ_REG = 8; "CURRENT SEGMENT ADDRESS" 

FIXCL_LINK = 1; "LOCAL WORD 1 IS ALWAYS LINK" 

FIXCL_OLDG = 2; "LOCAL WORD 2 IS ALWAYS OLD GLOBAL BASE"
FIXGNL_OLDG = 2; "GLOBAL WORD 2 IS OLD GLOBAL BASE"

HECNISIZE = 8; "MARK STACK TAKES 8 BYTES"

HIPILE = TIME;

ICONCURREN = TRUE;

ILENGTH = 8 "WORDS PER REAL"

IHALFORD = 1;

IKWORD = 2;

IWORDS = 6; FIELDS = 8; FIVWORDS = 10;

ISTACK_LIMIT = 32667 "GREATEST INTEGER = 100";

ICEP_LIM = 32667;

I THIS_PASS = 61;

IOFILE = 1; INTERFACEFILE = 4;

I"FILE NUMBER 3 IS USED BY PASS5 FOR FINAL CODE OUTPUT"

I"FILE FIELD 11 11TH IN TIB"

ISQO_SCG = 129; "AT RUN TIME, SECOND NO. OF SEQ L FON IS 129"

IINTERINE = 129; "AT RUN TIME, SECOND NO. OF INTERFACE SEGMENT"

ISCALL_PROC = 1; "START EXECUTION OF COMMAND WITH PROC NO. 1"

ISCALL_ERROR = 2;

ICALLERRORDISPL = 0; "***" "BYTES OF OBJECT CODE REQUIRED TO

CALL A NON-TIME ERROR ROUTINE."

"IF SRC_32"

I TYPE SRC_INT = ARRAY[1..2] OF SIGNED INTEGER;

I"END SRC_32"

I TYPES OF OUTPUT CODE PRODUCED BY THIS PASS;

I OPERATORS, UNIONS, BYTE OPERANDS, SIGNER BYTE OPERANDS;

I "DON'T CARE" BYTE OPERANDS, "BIC" OPERANDS, WORD OPERANDS,

I OR SOME VIRTUAL CODE (NOT MICROCODE)."

ITYPES SEC = (OPTM, OB, OR, OR, B, W, NOTM);

I VAR

I LINK, PASSPTR;

ITABLETENV = FULLWORD;

I SUMMARY, TEST, CHECK;

I GENERATE, NUMBER, AFTERMTEM;

I AFTERERROR, DONE, VARMCHECK: BOOLEAN; "VARM"

I GENINTFAC: BOOLEAN; "TRUE IF GENERATING AN INTERFACE SEQ"

I JUMPABLE, STACKABLE, CONSTABLE, LIFTABLE;

I EXITABLE, DATASIZABLE: TABLEPTR;

I "NUMBER OF ROUTINES IN EACH INTERFACE"

I IFSUBRTS: ARRAY[1..MAX_IFINTFAC] OF INTEGER;

I CONSTANTS, "COUNTS NO. OF WORDS OF CONSTANTS IN CONST TABLE"

I "COUNTS NO. OF WORDS OF I-XF OFFSETS IN CONST TABLE"

I "TOTAL WORDS IN I-XF POOL, INCLUDING I-XF OFFSETS"

I STACKLENGTH, "AS IT COMES FROM PASS5, STACKLENGTH IS THE AMOUNT OF EXTRA STACK SPACE REQUIRED BY A ROUTINE."

I IT IS INCREASED BY THE AMOUNT OF STACK SPACE

I THE ROUTINE NEEDS FOR VARIABLES & CALCULATIONS (IN-

I "BITS OF LOCAL VARS NEEDED FOR CURRENT ROUTINE"

I "BITE OF PARMS NEEDED FOR CURRENT ROUTINE"

I "BYTES TO POP AFTER CURRENT ROUTINE"
300 TEMP,
301 HA TPUP, "MAX BYTES OF CALCULATION STACK FOR CURRENT RUN"
302 BLOC K, "ROUTINE (PROCEDURE) LABEL FOR CURRENT ROUTINE"
303 LOCATION, "BYTE LOCATION COUNTER, RELATIVE TO START OF SEGMENT,"
304 & POINTS TO NEXT RTTP TO BE GENERATED."
305 IFSENDUN, "SEQUENTIAL NUMBER OF THE LAST INTERFACE SEG END"
306 LINE, "SOURCE LINE NUMBER"
307 OP, "VIRTUAL MACHINE INSTRUCTION OPERATOR, FROM PAGES"
308 ARD1, ARD2, ARD3, ARD4, ARDS "VIRTUAL MACHINE INSTRUC. OPERANDS"
309 : INTEGER;
310
311
312 "###################################"
313 "COMMON TEST OUTPUT MECHANISM"
314 "###################################"
315
316 PRINTED: INTEGER;
317
318 OK: BOOLEAN;
319 "PASS 1 TO 6: OK = NOT DISK OVERFLOW"
320 PASS 7: OK = NOT DISK OVERFLOW & PROGRAM CORRECT
321
322 PAGE_IN: PAGE; PAGES_IN: INTEGER;
323 PAGE_OUT: PAGE; PAGES_OUT, WORDS_OUT: INTEGER;
324 IPPAGE_OUT: PAGE; IPPAGES_OUT, IPPWORDS_OUT: INTEGER;
325
326 PROCEDURE PRINT_TEXT (TEXT: TEXT_TYPE);
327 VAR I: INTEGER;
328 BEGIN
329 WRITE(EOL);
330 FOR I := 1 TO TEXT_LENGTH DO WRITE(TEXT(.I.));
331 WRITE(EOL)
332 END;
333
334 PROCEDURE FILE_LIMIT;
335 BEGIN
336 PRINT_TEXT('PASS 6: FILE LIMIT');
337 OK := FALSE
338 END;
339
340 PROCEDURE INIT_PASS (VAR LINK: PASSPTR);
341 BEGIN
342 LINK := PARAM(.2.).PTR;
343 OK := TRUE;
344 PAGE_IN := 11; WORDS_IN := PAGE LENGTH;
345 PAGE_OUT := 11; WORDS_OUT := 0;
346 IPPAGE_OUT := 11; IPPWORDS_OUT := 0;
347 END;
348
349 PROCEDURE NEXT_PASS (LINK: PASSPTR);
350 BEGIN
351 IF WORDS_OUT > 0
352 THEN IF PAGES_OUT > FILE_LENGTH(OUTFILE)
353 THEN FILE_LIMIT
354 ELSE PUT(OUTFILE, PAGES_OUT, PAGE_OUT);
355 IF IPPWORDS_OUT > 0
356 THEN IF IPPAGE_OUT > FILE_LENGTH(INTERFACEFILE)
357 THEN FILE_LIMIT
358 ELSE PUT(INTERFACEFILE, IPPAGES_OUT, IPPAGE_OUT);
359 WITH PARAM(.1.) DO BEGIN
360
361
360  Tag := BOOL; Tag := OK END;
361  WITH Param.1 DO BEGIN
362  Tag := PTYPE; Param := LINK END;
363  WITH Param.2 DO BEGIN
364  Tag := INTTYPE; Int := PAGES.Out END;
365  END;
366  PROCEDURE READ_IFL (VAR I: INTEGER);
367  BEGIN
368  IF WORS."IN =" PAGE LENGTH THEN BEGIN
369  IF PAGES."IN >" FILE LENGTH THEN File LIMIT
370  END ELSE BEGIN
371  GET(INFILE, PAGES."IN, PAGE."IN);
372  PAGES."IN :=" SUCC(PAGES."IN);
373  END;
374  WORD."IN :=" 0;
375  WORD."IN :=" SUCC(WORDS."IN);
376  I := PAG"E. IN.(WORDS."IN); END;
377  PROCEDURE WRITE_IFL (I: INTEGER);
378  BEGIN
379  IF GEN"ERING.INT FAC
380  THEN BEGIN
381  IFWORDS."OUT :=" SUCC(IFWORDS."OUT);
382  IFPAGE."OUT := IFWORDS."OUT :=" I;
383  IFWORDS."OUT :=" PAGE LENGTH
384  THEN BEGIN
385  IF PAGES."OUT >" FILE LENGTH THEN File LIMIT
386  END;
387  PAGES."OUT :=" SUCC(PAGES."OUT);
388  END;
389  IFWORDS."OUT :=" 0;
390  END;
391  PROCEDURE PRINTARC(ARG: INTEGER);
392  VAR X: ARRAY [.1, MAXDIGIT] OF CHAR; Rem, Digit, I: INTEGER;
393  BEGIN
394  Rem := ARG; Digit := 0;
395  REPEATE
396  Digit := Digit + 1;
397  END;
420    Y.DIGIT. := CHR(ASC(RED MOD 10) + ORD('0'));
421    RED := RED DIV 10;
422    UNTIL (RED=0) OR (DIGIT=MAXDIGIT);
423    FOR I := DIGIT DOWNTO 1 DO WRITE('.,I,);'
424    FOR I := DIGIT + 1 TO MAXDIGIT DO WRITE(' ',
425      END;
426
427    PROCEDE PRINTN;
428    PRINTN WRITE(EL); PRINTN := 0 END;
429
430    PROCEDURE PRINTFF;
431    VAR I : INTEGER;
432    BEGIN
433    PRINTN; FOR I := 1 TO 15 DO WRITE('6'); PRINTN
434    END;
435
436    PROCEDURE PRINTTOPOP: INTEGER;
437    BEGIN
438    IF PRINTN = PRINTLIMIT THEN PRINTN;
439    WRITE('C'); PRINTN;
440    PRINTN := PRINTN + 1;
441    END;
442
443    PROCEDURE PRINTARG(ARG: INTEGER);
444    BEGIN
445    IF PRINTN = PRINTLIMIT THEN PRINTN;
446    IF ARG < 0 THEN WRITE(' '); ELSE WRITE(' ');
447    PRINTN(ARG);
448    PRINTN := PRINTN + 1;
449    END;
450
451 "################################################
452 "INPUT PROCEDURES"  
453 "################################################
454
455    PROCEDURE READARG1;
456    BEGIN READ_IFL(ARG1) END;
457
458    PROCEDURE READARG2;
459    BEGIN READ_IFL(ARG2) END;
460
461    PROCEDURE READARG3;
462    BEGIN READ_IFL(ARG3) END;
463
464    PROCEDURE READARG;
465    BEGIN
466    READ_IFL(ARG1); READ_IFL(ARG2);
467    READ_IFL(ARG3);
468    END;
469
470    PROCEDURE READARG;
471    BEGIN
472    READ_IFL(ARG1); READ_IFL(ARG2); READ_IFL(ARG3);
473    READ_IFL(ARG4); READ_IFL(ARG5);
474    END;
475
476 "################################################
477 "OUTPUT PROCEDURES"  
478 "################################################
PROCEDURE ERROR (PASS, NUMBER: INTEGER); FORWARD;

PROCEDURE UPDLOC (CODE: TYPECODE; VALUE: INTEGER);
BEGIN "UPDATE THE OUTPUT CODE LOCATION COUNTER."
IF LOCATION < COMP_LIMIT
THEN CASE CODE OF
06: LOCATION := LOCATION + HALFWORD;
87: B: IF VALUE <= 177
THEN LOCATION := LOCATION + HALFWORD
ELSE LOCATION := LOCATION + ONWORD;
M: LOCATION := LOCATION + ONWORD;
OTHERWISE: 269
END;
ELSE BEGIN
ERROR (THIS_PASS, CODE_ERROR);
LOCATION := 0;
END;
END;

LOCATION COUNTER IS KEPT FOR THE CONCURRENT SEGMENT ONLY.

IF CODE OUTPUT ROUTINES ARE CALLED WHILE GENERATING AN INTERFACE
THEN DON'T FOOL WITH CONCURRENT CODE

PROCEDURE WRITE1(OP: INTEGER);
BEGIN IF TEST THEN PRINTOP(OP);
WRITE_IFL(OP);
IF NOT GENRINGINTPAC THEN UPDLOC (OPTR, OP)
END;
PROCEDURE WRITE2(OP, ARG: INTEGER; ARG_TYP: TYPECODE);
BEGIN IF TEST THEN BEGIN
PRINTOP(OP); PRINTARG(ARG)
END;
WRITE_IFL(OP); WRITE_IFL(ARG);
IF NOT GENRINGINTPAC THEN BEGIN
UPDLOC(OPTR, OP); UPDLOC(ARG_TYP, ARG)
END;
PROCEDURE WRITE3(OP, ARG1, ARG2: INTEGER;
ARG1_TYP, ARG2_TYP: TYPECODE);
BEGIN IF TEST THEN BEGIN
PRINTOP(OP); PRINTARG(ARG1); PRINTARG(ARG2)
END;
WRITE_IFL(ARG1); WRITE_IFL(ARG2);
IF NOT GENRINGINTPAC THEN BEGIN
UPDLOC(OPTR, OP); UPDLOC(ARG1_TYP, ARG1);
UPDLOC(ARG2_TYP, ARG2)
END;
PROCEDURE WRITER(O1, O2, O3: INTEGER; INTEGER; O1TYPE, O2TYPE, O3TYPE: TYPEOFCODE);

BEGIN
    IF TEST THEN 276

    PRINTOP; PRINTARG(O1); 436 541 443 541
    PRINTARG(O2); PRINTARG(O3); 443 541 443 541

    WRITE_IPL(O1); WRITE_IPL(O2); 381 541 381 541
    WRITE_IPL(O3); WRITE_IPL(O4); 381 541 381 541

    IF NOT GENUINSINTFACE 280

    THEN BEGIN
    UPOLOC(OPTN, OP); UPOLOC(O1TYPE, O1); 482 541 541 482 541
    UPOLOC(O2TYPE, O2); UPOLOC(O3TYPE, O3); 482 541 541 482 541

    END;

    END;

    IF TEST THEN 276

    THEN BEGIN
    PRINTOP; 436 558
    PRINTARG(O1); PRINTARG(O2); 483 558 483 558
    PRINTARG(O3); PRINTARG(O4); 483 558 483 558

    WRITE_IPL(O1); WRITE_IPL(O2); 381 558
    WRITE_IPL(O3); WRITE_IPL(O4); 381 558 381 558

    IF NOT GENUINSINTFACE 280

    THEN BEGIN
    UPOLOC(OPTN, OP); 482 558
    UPOLOC(O1TYPE, O1); UPOLOC(O2TYPE, O2); 482 559 558 482 559 558
    UPOLOC(O3TYPE, O3); UPOLOC(O4TYPE, O4); 482 559 558 482 559 558

    END;

    END;

PROCEDURE WRITARG(ARG: INTEGER; ARGTYPE: TYPEOFCODE);

BEGIN
    IF TEST THEN PRINTARG(ARG); 276 443 518

    WRITE_IPL(ARG); 381 578

    IF NOT GENUINSINTFACE THEN UPOLOC(ARGTYPE, ARG); 280 482 578 578

    END;

PROCEDURE WRITELOCATION;

BEGIN
    IF TEST THEN PRINTARG(LOCATION); 276 443 303

    WRITE_IPL(LOCATION); 381 303

    END;

PROCEDURE ERROR;

BEGIN
    IF NOT AFTERERROR 278

    THEN BEGIN
    AFTERERROR; TRUE; 278 249
    WRITEARG(MESSAGE, 2, NOTHE); 578 217 269
    WRITEARG(PASS, NOTHE); 578 480 269
    WRITEARG(MESS, NOTHE); 578 240 269
    WRITEARG(LINK, NOTHE); 578 306 269
600  GENERATE:= FALSE
601  END
602  END;
603
604  "******************************************************************************
605  "PSUEDO MIN-THE STACK PROCEDURES"
606  "******************************************************************************
607
608  PROCEDURE FUSHWORD;
609  BEGIN
610  IF TEMP < STACK_LIMIT THEN TEMP:= TEMP + WORLENGTH
611  ELSE ERROR(TTHIS_PASS, STACK_ERROR);
612  IF TEMP > HATTEMP THEN HATTEMP:= TEMP;
613  END;
614
615  PROCEDURE FPOWORD;
616  BEGIN TEMP:= TEMP - WORLENGTH;
617  END;
618
619  PROCEDURE FUSHREAL;
620  BEGIN
621  IF TEMP < STACK_LIMIT THEN TEMP:= TEMP + REALLENGTH
622  ELSE ERROR(TTHIS_PASS, STACK_ERROR);
623  IF TEMP > HATTEMP THEN HATTEMP:= TEMP;
624  END;
625
626  PROCEDURE FPOREAL;
627  BEGIN TEMP:= TEMP - REALLENGTH END;
628
629  PROCEDURE FUSSET;
630  BEGIN
631  IF TEMP < STACK_LIMIT THEN TEMP:= TEMP + SETLENGTH
632  ELSE ERROR(TTHIS_PASS, STACK_ERROR);
633  IF TEMP > HATTEMP THEN HATTEMP:= TEMP;
634  END;
635
636  PROCEDURE FPOSET;
637  BEGIN TEMP:= TEMP - SETLENGTH END;
638
639  PROCEDURE FUSH(LLENGTH; INTEGER);
640  BEGIN
641  IF TEMP < STACK_LIMIT - LENGTH THEN TEMP:= TEMP + LENGTH
642  ELSE ERROR(TTHIS_PASS, STACK_ERROR);
643  IF TEMP > HATTEMP THEN HATTEMP:= TEMP;
644  END;
645
646  PROCEDURE FPO(LLENGTH; INTEGER);
647  BEGIN TEMP:= TEMP - LENGTH END;
648
649  "******************************************************************************
650  " SHORT CONSTANT PUSH "
651  "******************************************************************************
652
653  PROCEDURE FICK_PUSHCONST (VALUE; INTEGER);
654  BEGIN  "FICK 1 OF 3 OPERATORS WHICH PUSH IMMEDIATE CONSTANTS"
655  IF (VALUE >= 0) AND (VALUE <= 31) THEN WRITE1(VALUE)
656  ELSE IF (VALUE >= 32) AND (VALUE <= 255) THEN WRITE2(LDCK_2, VALUE, UB)
657  ELSE WRITE2(LDC1_2, VALUE, W)
658  END;
659
660  "******************************************************************************
660 END;
661  "---------"
662 "VARIABLE PROCEDURES"
663  "---------"
664
665 FUNCTION DISP[MODE, ARG; INTVAR]; INTEGER;
666 "HARTMANN'S VARIABLES HAVE NEGATIVE DISPLACEMENT, PARAMETERS HAVE
C 667 POSITIVE DISPLACEMENTS, BOTH IN BYTES & 2-BASED.
C 668 FOR MICROMODE, BOTH VARIABLES AND PARAMETERS HAVE
C 669 POSITIVE WORD DISPLACEMENTS (1-BASED). VARS HAVE SMALLER DISPLA
C 670 SINCE THEY ARE CLOSER TO THE NCM (PUSHED LATER) THAN PARM."  5
671 BEGIN
672 CASE MODE OF
673   012 CASE MODE OF
674      5 "PROCEDURE, CLASS ENTRY, MONITOR ENTRY, NAM ENTRY" 666
675                             MODEL1, NODE2, NODE3, NODES, NODE23:
676      IF ARG < 0 180 181 182 183 184 185
677      THEN DISPL := (-ARG + TWOWORDS) DIV WORDLENGTH 666 666 247 28
678      ELSE DISPL := (ARG + VARLENGTH + TWOWORDS) DIV WORDLENGTH; 666 666 247 28
679      END;
680      "PROCESS ENTRY"
681      "INTERFACE PROCEDURE HAS NO LOCAL VARS (NOT EVEN LINE & OLD
C 682 GLOBAL BASE). HOWEVR, IT DOES HAVE AN EXTRA FARM (THE NO.
C 683 OF THE PREFIX PROCEDURE BEING CALLED) WHICH HAS NOT BEEN
C 684 TAKEN INTO ACCOUNT IN DISPLACEMENT CALCULATION TO THIS POINT.
C 685 IF ARG < 0 THEN REFERENCE IS TO A LOCAL VARIABLE IN THE ENTRY
C 686 PROCEDURE. OTHERWISE, REFERENCE IS TO FARM IN THE INTERFACE
C 687 PROCEDURE."
688      IF ARG < 0 666
689      THEN DISPL := (-ARG + TWOWORDS) DIV WORDLENGTH 666 666 247 28
690      ELSE DISPL := (ARG + ONEROWD) DIV WORDLENGTH; 666 666 246 28
691      "PROCESS, CLASS, MONITOR, MANAGER MODES"
692      NODES, NODE3, NODE, NODE3, NODE24:
693      "IN THESE MODES, REFERENCE IS TO SHARED VARIABLES OR FAMNS
C 694 TO THE INITIAL STATEMENT. THE RECORD CONTAINING THESE FAMNS
C 695 & FAMNS IS STORED IN THE SHARED VARIABLE AREA OF THE INITIAL
C 696 PROCESS. SPACE FOR THE RECORD WAS ALLOCATED IN PRIOR PASSES.
C 697 THE GLOBAL Records DO NOT CONTAIN TWO WORDS FOR LINE NO. &
C 698 OLD GLOBAL BASE."
699      IF ARG < 0 666
700      THEN DISPL := (-ARG) DIV WORDLENGTH 666 666 28
701      ELSE DISPL := (ARG + VARLENGTH) DIV WORDLENGTH; 666 666 297 28
702      "UNDEFINED MODE" 189
703 END;
704 END;

710 BEGIN
711 PROCEDURE PUSHV[MODE, ARG; INTVAR];
712 BEGIN
713 "PROCEDURE, CLASS ENTRY, MONITOR ENTRY, NAM ENTRY MODES"
714 MODEL1, NODE2, NODE3, NODE23:
715      "HELLO"
716      "WRITE(4,2, DISP[MODE, ARG],B);"
717      "PROGRAM MODE" 181
718
719 MODE2
720 WRITE(LDO_2, DISPL(MODE, ARG), B);
721
722 MODE3: "PROCESS ENTRY MODE"
723 IF ARG < 0
724 THEN WRITE(LDO_2, DISPL(MODE, ARG), B)
725 ELSE WRITE(LDO_2, 1, DISPL(MODE, ARG), DB, B);
726 "PROCESS, CLASS, MONITOR, MANAGER NODES"
727 MODE6, MODE7, MODEB, MODE11: "MHR"
728 WRITE(LDO_2, DISPL(MODE, ARG), B);
729
730 MODE10: "UNDEFINED MODE"
731
732 END;
733 PUSHWORD;
734 END;
735
736 PROCEDURE PUSHADDRESS(MODE, ARG: INTEGER);
737 BEGIN
738 CASE MODE OF
739 MODE0: "CONSTANT MODE"
740 WRITE(LCA_2, DISPL(MODE, ARG), B);
741 "PROCEDURE, CLASS ENTRY, NEW ENTRY, MM. ENTRY"
742 NODE1, NODE2, NODE5, NODE12: "NE" "NE"
743 WRITE(LLA_2, DISPL(MODE, ARG), B);
744 "PROGRAM NODE"
745 WRITE(LAO_2, DISPL(MODE, ARG), B);
746 "PROCESS ENTRY MODE"
747 WRITE(LDO_2, DISPL(MODE, ARG), B);
748 MODE3: "PROCESS ENTRY MODE"
749 IF ARG < 0
750 THEN WRITE(LDO_2, DISPL(MODE, ARG), B)
751 ELSE WRITE(LDO_2, 1, DISPL(MODE, ARG), DB, B);
752 "PROCESS, CLASS, MONITOR, MANAGER NODES"
753 MODE6, MODE7, MODEB, MODE11: "MHR"
754 WRITE(LAO_2, DISPL(MODE, ARG), B);
755 MODE10: "UNDEFINED MODE"
756
757 END;
758 PUSHWORD;
759 END;
760
761 PROCEDURE PUSHINDIRECT(VARTYPE: INTEGER);
762 BEGIN
763 CASE VARTYPE OF
764 BYTETYPE: 765 BEGIN WRITE1(LDB_2); PUSHWORD END;
766 WORDTYPE: 135
767 WRITE1(SINDO_2);
768 REALTYPE: 135
769 BEGIN WRITE1(LLM_2, LENGTHWORD DIV WORDLENGTH, UN);
770 PUSHWORD; PUSHREAL;
771 END;
772 SETTYPE: 135
773 BEGIN WRITE1(LLM_2, SETLENGTH DIV WORDLENGTH, UN);
774 PUSHWORD; PUSHSET;
775 LENGTHWORD DIV WORDLENGTH);
776 PICK_PUSHCONST(SetLENGTH DIV WORDLENGTH);
777 PUSHWORD; "PUSH WORD LENGTH OF SET" 608
778 END
END;

PROCEDURE ALLOCATE(VAR T: TABLEPTR; ENTRIES: INTEGER);
VAR PORTION: TABLEPTR; I: INTEGER;
BEGIN
  NEW(T); PORTION := T;
  I := ENTRIES - MAXWORD;
  WHILE I > 0 DO
    WITH PORTION OF
      NEW(NEXTPORTION); PORTION := NEXTPORTION;
    I := I - MAXWORD;
  END;
END;

PROCEDURE ENTERM(T: TABLEPTR; I, J: INTEGER);
VAR PORTION: TABLEPTR; K: INTEGER;
BEGIN
  PORTION := T; K := I;
  WHILE K > MAXWORD DO
    BEGIN
      PORTION := PORTION OF NEXTPORTION;
      K := K - MAXWORD;
    END;
  BEGIN
    PORTION OF CONTENTS(K) := J;
  END;
END;

FUNCTION ENTR(T: TABLEPTR; I: INTEGER): INTEGER;
VAR PORTION: TABLEPTR; J: INTEGER;
BEGIN
  PORTION := T; J := I;
  WHILE J > MAXWORD DO
    BEGIN
      PORTION := PORTION OF NEXTPORTION;
      J := J - MAXWORD;
    END;
  BEGIN
    ENTR := PORTION OF CONTENTS(J);
  END;
END;

"INITIALIZATION AND TERMINATION PROCEDURES"

PROCEDURE BEGINPASS;
VAR I: INTEGER;
BEGIN
  WITH LINK OF
    BEGIN
      SUMMARY := SUMMARYOPTION IN OPTIONS;
      TEXT := TEXTOPTION IN OPTIONS;
      CHECK := CHECKOPTION IN OPTIONS;
      VARCHCK := VARCHCKOPTION IN OPTIONS;
      NUMBER := NUMBEROPTION IN OPTIONS;
    END;
END;

BEGIN
  END;
END.
GENERATE := TRUE;
FOR I := 1 TO MAXINTFACTOR DO
    IFDSIZE[I] := LENGTHOFINTERFACESIZE[I];
ALLOCATE(JUMP TABLE, LABELS);
ALLOCATE(CONSTANT, CONSTANTS DIP WORDLENGTH);
ALLOCATE(LIQUID, LIQUID_OFFSETS DIP WORDLENGTH);
ALLOCATE(EXECUTABLE, BLOCKS);
ALLOCATE(DATASIZABLE, BLOCKS);
MARK (TABLESET);
ALLOCATE(MARKTABLE, BLOCKS);
END;
CONSTANTS := 0;
XJPOFFSETPTR := 0;
POOLSIZE := LINK_OFFSETS + LINK_OFFSETS;
LINE := 0;
LOCATION := ONWORD + POOLSIZE;
AFTERBEGIN := FALSE;
IFSIGM := 0;
GENMINGINTFACTOR := FALSE;
"LABEL 1 (TARGET OF 1ST DISCARDRED JUMP) IS NEVER RESOLVED"
ENTER(JUMP TABLE, STARTPROC, -1);
IF TEST THEN PRINTF;
PROCEDURE ENDPASS;
BEGIN
LINK := CONSTANTS;
LINK := LINK_OFFSETS;
WITH LINK DO
    IF GENERATE THEN OPTIONS := OPTIONS OR (.CODEOPTION,);
    RELEASE (TABLESET);
    REMOVEDICTIONARY;
    BEGINTABLE,
    WITH (LINK)
    "COECCOMMENT IS LOC. CTR. + SPACE FOR COECCOMMENT DICTIONARY.
    LOCATION WILL THEN POINT TO THE END OF THE
    (LAST) WORD WHICH WILL CONTAIN NO. OF PROCESSES IN SEG. &
    SEGMENT NUMBER."
    TABLES := LOCATION + 2*LINK_OFFSETS;
    TABLES := JUMP TABLE;
    TABLES := CONSTANT TABLE;
    TABLES := LIQUID TABLE;
    TABLES := EXECUTABLE TABLE;
    TABLES := DATASIZABLE TABLE;
END;
BEGIN
    "IF SRC_32"
    PROCEDURE SPLIT_CONSTANT(WORD := UNIWORD, SPLIT_INTEGER);
    VAR WORD, LWORD, INTGER;
    BEGIN "SPLIT A 32-BIT INTEGER INTO TWO 16-BIT INTEGERS"
    LWORD := WORD[1]; LOWORD := WORD[2];
    END;
    "END"
    IF NUMBER
    THEN BEGIN
    THEN

900 WRITEAO (NEWLIN_2, NOTHE); "CRUTCH FOR MACHINICS FON."
901 Polec:numcst (LINE); PUSHWORD;
902 WRITE2 (STL_2, FIXCL_LINE, B); POPWORD;
903 END;
904 Polec:fushcst (GNSA_REQ);
905 WRITE1 (LFR_2);
906 WRITE2 (STL_2, FIXCL_GLSA, B); POP(TWOWORDS)
907 END;
908
909 PROCEDURE GEN_EXIT;
910 "EPILOGUE FOR ROUTINES"
911 BEGIN
912 Polec:fuishcst (GNSA_REQ);
913 WRITE1 (ALNSO_2);
914 WRITE1 (SFR_2);
915 WRITE2 (KLPL_2, TPLLENGTH DIV WORDLENGTH, B);
916 POP(POPLNGTH + NROWSIZE)
917 END;
918
919 "########################################################
C 921 # GENERATE AN INTERFACE SEGMENT #
C 922########################################################
C 923
C 924 PROCEDURE GEN_INTERFACE;
C 925 TYPE
C 926 TOPFTR = 6KENTRTY;
C 927 KENTRTY = RECORD
C 928 DATA: INTEGER;
C 929 KNTRTY = TOPFTR *POINTS TO THING PUSHED JUST BEFORE"
C 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 SYMT. I.E., PREFIX RTNS ARE
C 935 NUMBERED STARTING AT 1"
936 DATASIZE = 0; "ROUTINE BEING GEN'D HAS NO LOCAL VAR SPACE"
937 PARAMETERS = 1; "ROUTINE BEING GEN'D NEEDS 1 PARAMETER"
938 XJNTABLE = 1; "TABLE FOR CASEJUMP INSTRCS IN WORD 1 OF SEG."
939 PROCSEND = 1; "INTERFACE SEG CONSIST OF ONLY 1 PROCEDURE"
940 VAR
941 OILAQHTOPC: FULLWORD;
942 RTNS, "NO. OF ROUTINE LABELS IN THIS INTERFACE"
943 KASE, "LOOP VAR FOR RTNS & CASES IN CASE SYMT"
944 ILABEL: INTEGER; "INTERFACE ROUTINE LABELS"
945 TOS; TOPFTR; "POINTS TO CURRENT TOP OF STACK"
946 NEWTOP; TOPFTR; "POINTS TO THING TO BE PUSHED NEXT"
947 BEGIN
948 GEMININTFAc := TRUE; "FROM NOW TO END OF THIS PROC, SEND
C 949 OUTPUT TO THE FILE OF INTERFACE SEGNS"
950 IFSEGNUM := ISZINC(ISZNUM);
951 RTNS := IFSEGNUM[IFSEGNUM]; "GET NO. OF ROUTINES IN THIS INTFAC.
C 952 WRITEAO (RTNS+B+27) DIV 2, W); "WORDS IN SEG, LESS 1"
953 "GEN CONSTANT BLOCK. IT CONSISTS ONLY OF THE TABLE FOR CASE SYMT."
954 WRITEAO (HIINDEX, W); "MIN INDEX OF CASE SYMT"
955 WRITEAO (RTNS, W); "MAX INDEX OF CASE SYMT"
956 FOR KASE := 0 TO RTNS-1 DO WRITEAO (KASE*4+B, W); "OFFSETS"
"GEN CODE TO USE THE NO. OF A PREFIX ROUTINE (THE CALLING
SEQUENTIAL PROGRAM MUST SUPPLY THIS AS A PARAMETER) AS THE INDEX
INTO CASE STMNT AND CALL CORRESPONDING CONCURRENT PROCESS
ENTRY ROUTINE."  

"PUSH THE PARAMETER"

"GEN'D IN ILLUSION OF WESTERN DIGITAL"

"FOR THE CASE WHERE FARP IS OUT OF RANGE"

"THE INTERFACE ROUTINE LABELS COME INTO THIS PASS IN REVERSE ORDER.
THEY NEED TO BE IN FORWARD ORDER. STACK THEM AND POP THEM TO
MAKE THE CONNECTION."

BEGIN

BEGIN

BEGIN

BEGIN

BEGIN

BEGIN

BEGIN

BEGIN

BEGIN

BEGIN

BEGIN

BEGIN

BEGIN

BEGIN
C 1020 OF PROCEDURES IN THE SEGMENT JUST GEN'D, & THE LOW ORDER BYTE
C 1021 CONTAINS THE SEQ NUMBER OF SEG JUST GEN'D, ASSUMING 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 REMEMBERED BY THE KERNEL WHEN LOADED, BUT THESE NUMBERS
C 1025 SHOULD HELP IN TESTING SINCE WEST. BID. SOFTWARE DOESN'T SUPPORT
C 1026 MORE THAN 16 SEGMENTS.*
1027 whitearg (procseg * horder + (ifsgenun + congso), h); 578 939 933 305 932 269
1028
1029 genmingintfac := FALSE "SEND OUTPUT TO CONCURRENT FILE" 280 337
1030 END;
1031
1032
1033
1034 ************
1035 "OPERATOR*"
1036 ************
1037
1038 PROCEDURE SCAN;
1039 VAR ARG1LO, ARG1HI; INT8ER;
1040 BEGIN
1041
1042 read_ifl (op); 367 558 455
1043 1044 done:=false; "KEEP READING & TRANSLATING CODE UNTIL END-OF-MEDIUM.*"
1045 repeat
1046 read_ifl(op);
1047 case op of
1048
1049 pushconst1"(value)";
1050 begin
1051 read1arg; 455 653 558 608
1052 end;
1053
1054 pushvar1"(type, mode, displ)";
1055 begin read3arg; 120
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)";
1065 begin read1arg; 120
1066 pushdirect(arg1) 455 763 558
1067 end;
1068 pushaddr1"(mode, displ)";
1069 begin read2arg; 120
1070 pushaddress(arg1, arg2) 458 736 558 558
1071 end;
1072 if arg1 <> 0 558
1073 then write(inc_2, (arg1 div wordlength), h) 511 215 558 28 269
1074 end;
1075
1076 index1"(min, max, length, type)";
1077 begin readarg; 121
1078 push(const1 arg1); 464
1079 push(const1 arg2); 653 558 608
1080 push(const1 arg2); 653 558 608
1080 WRITE(CMK_2); POP(TWOMODS); 504 208 646 247
1081 PICK_PUSHSRT(ARGO); PUSHWORD; 653 598 608
1082 WRITE(SMK_2); POPWORD; 504 199 615
1083 IF ARG0 <> BYTETYPE 558 135
1084 THEN BEGIN
1085 WRITE((MA, 2), ARG3 DIV WORDLNGTH, 0); POPWORD 511 211 558 28 269 615
1086 END;
1087 END;
1088
dofoil: 121
1089 BEGIN 216
1090 IF THEN BEGIN
1091 WRITE(DMP_2); PUSHWORD; 504 211 608
1092 WRITE(SLDCO_2); PUSHWORD; 504 172 608
1093 WRITE((FLJ_2, CALLERBORDISL, SN)); POP(TWOMODS); 511 240 257 269 646 247
1094 "generate call to pointer error routine. ***" 121
1095 END
1096 END;
1097
dvarint("tagset, displ"); 121
1098 "change prior passes so that tos tag is <value> of
1099 BEGIN READARG; 458
1100 IF (WARNCHECK) 216
1101 THEN BEGIN
1102 WRITE(DMP_2); PUSHWORD; 504 211 608
1103 WRITE((LAND_2, ARG2 DIV WORDLNGTH, 0); "POPMOD; PUSHWORD;" 511 240 558 28 269
1104 WRITE((LDCL_2, ARG1, 0); PUSHWORD; 511 191 558 269 608
1105 WRITE((LAND_2,); "POPMOD;" 504 199
1106 WRITE((SLDCO_2,); "PUSHWORD;* 504 172
1107 WRITE((FLJ_2, CALLERBORDISL, SN)); POP(TWOMODS); 511 210 257 269 646 247
1108 "generate call to variant error routine. ***" 121
1109 END
1110 END;
1111
drange="(min, max)"; 122
1112 BEGIN READARG; 458
1113 IF CHECK 216
1114 THEN BEGIN
1115 PICK_PUSHSRT(ARG0); PUSHWORD; 653 598 608
1116 PICK_PUSHSRT(ARG02); PUSHWORD; 653 598 608
1117 WRITE(CMK_2); POP(TWOMODS); 504 208 646 247
1118 END
1119 END;
1120
dassign("type"); 122
1121 BEGIN READARG; 458
1122 CASE ARG0 OF
1123 BYTETYPE 135
1124 BEGIN WRITE(STL_2); POPWORD END;
1125 WORDTYPE 135
1126 BEGIN WRITE(STL_2); POPWORD END;
1127 REALTYPE 135
1128 BEGIN READ; 135
1129 WRITE(STL_2, REALLENGTH DIV WORDLNGTH, 0); POPREAL 511 194 29 28 269 626
1130 END;
1131 SETTYPE 135
1132 BEGIN 135
1133 WRITE((FPJ_2, 0, 0)); POPWORD; "POP SET LENGTH VALUE" 511 211 269 615
1134
"ASSIGNTAG1 (LENGTH) : ASSIGNTAG1 NEVER PRODUCED BY PASS 5.

C BEGIN READARG;
C IF ARG1 = 0 THEN WRITE(COPTWORD2);
C ELSE WRITE(COPTTAG2, ARG1 DIV WORDLENGTH);
C POPWORD; POPWORD;
C END;
N
"ASSIGNTAG1 (LENGTH, INITIALIZE): 

C BEGIN READARG;
C IF (ARG2 = 1) & CHECK
C THEN WRITE(MEMINIT2, BLOCK, ARG1)
C ELSE WRITE(MEM, BLOCK, ARG1);
C POPWORD;
C END;
N
NOT1:
C WRITE(LNH2);
C AND1(TYPE);
C BEGIN READARG;
C IF ARG1 = WORDTYPE
C THEN BEGIN WRITE(LAND2); POPWORD POPWORD END
C ELSE BEGIN WRITE(IN12); POPWORD; POPSET END;
C END;
N
OR1(TYPE);
C BEGIN READARG;
C IF ARG1 = WORDTYPE
C THEN BEGIN WRITE(LOR2); POPWORD POPWORD END
C ELSE BEGIN WRITE(OUN12); POPWORD; POPSET END;
C END;
N
NOT3(TYPE);
C BEGIN READARG;
C IF ARG1 = WORDTYPE
C THEN BEGIN WRITE(AND2); POPWORD POPWORD END
C ELSE BEGIN WRITE(AOR2); POPREAL POPREAL END;
C END;
N
SUB1(TYPE);
C BEGIN READARG;
C CASE ARG1 OF
C
1200  WORDTYPE: BEGIN WRITE((SNL_2)); POPWORD END;
1201  REALTYPE: BEGIN WRITE((SNL_2)); POPREAL END;
1202  SETTYPE: BEGIN WRITE((DIE_2)); POPWORD; POPREAL END
1203  END;
1204  END;
1205  END;
1206  MULTI(TYPE)*;
1207  BEGIN READARG;
1208  IF ARG1 = WORDTYPE
1209  THEN BEGIN WRITE((HIP_2)); POPWORD END
1210  ELSE BEGIN WRITE((HIE_2)); POPREAL END;
1211  END;
1212  END;
1213  DIVI(TYPE)*;
1214  BEGIN READARG;
1215  IF ARG1 = WORDTYPE
1216  THEN BEGIN WRITE((DVL_2)); POPWORD END
1217  ELSE BEGIN WRITE((DVI_2)); POPREAL END;
1218  END;
1219  END;
1220  MODI(TYPE)*;
1221  BEGIN READARG; WRITE((MODI_2)); POPWORD END;
1222  *(NOT USED)*
1223  *(NOT USED)*
1224  *(NOT USED)*
1225  *(NOT USED)*
1226  *(NOT USED)*
1227  FUNCTION*(STANDARDFUNC, TYPE)*:
1228  BEGIN READARG;
1229  IF (ARG1 >= MIN_FUNC) AND (ARG1 <= MAX_FUNC) THEN
1230  CASE ARG1 OF
1231  TRUNC1: 160
1232  BEGIN WRITE((MNC_2)); POPREAL; PUSHWORD END;
1233  ARG2: 160
1234  IF ARG2 = WORDTYPE
1235  THEN WRITE((AR2_2)); 160
1236  ELSE WRITE((ARB_2)); 160
1237  SUC1;
1238  BEGIN
1239  WRITE((SLUCO1_2)); PUSHWORD;
1240  WRITE((ADL_2)); POPWORD
1241  END;
1242  PREC1;
1243  BEGIN
1244  WRITE((SLUCO1_2)); PUSHWORD;
1245  WRITE((SBL_2)); POPWORD;
1246  END;
1247  CONV1:
1248  BEGIN WRITE((FNL_2)); POPWORD; PUSHREAL END;
1249  PMT1:
1250  BEGIN
1251  WRITE((SLUCO2_2)); PUSHWORD;
1252  WRITE((EQO_2)); POPWORD;
1253  END;
1254  ATTRIBUTE1; "MOD"
1255  "WRITE(ATTRIBUTE2);"
1256  REALTIME1; "MOD"
1257  "BEGIN WRITE(REALTIME2); PUSHWORD END"
1258  END;
1259  END;
CASE ARG1 OF
LENS: BEGIN WRITE2(QUOTE_2, ARG2, B); WRITE1(LNOUT_2); END;
EQUAL: BEGIN WRITE2(QUOTE_2, ARG2, B); END;
GREATER: BEGIN WRITE2(QUOTE_2, ARG2, B); WRITE1(LNOUT_2); END;
NOTLESS: BEGIN WRITE2(QUOTE_2, ARG2, B); WRITE1(LNOUT_2); END;
NOTSIGNED: BEGIN WRITE2(QUOTE_2, ARG2, B); WRITE1(LNOUT_2); END;
NOTGREATER: BEGIN WRITE2(QUOTE_2, ARG2, B); END;
END;
FUNCTION1("MODE, TYPE"): BEGIN READARG;
CASE ARG1 OF
MODE1, MODE3: "PROCEDURE MODE, PROCESS ENTRY MODE"
IF ARG2 = WONTYPE THEN BEGIN WRITE2(SLDCO_2); PUSHWORD END;
ELSE BEGIN WRITE2(SLDCO_2); WRITE1(SLDCO_2); PUSHREAL END;
END;
MODE4, MODE5, MODE7: "ENTRY MODES: CLASS, MONITOR, NOW"
IF ARG2 = WONTYPE THEN BEGIN
WRITE2(SLDCO_2); WRITE1(SNAP_2); PUSHWORD
END;
ELSE BEGIN
WRITE2(SLDCO_2); WRITE1(SNAP_2); PUSHREAL
END;
END;
DEFLAB1("LABEL"): BEGIN READARG;
ENTER(INTERFACE, ARG1, LOCATION);
IF NUMBER THEN BEGIN
PICK_PUSCONST(LINE); PUSHWORD;
WRITE2(SLDCO_2, FIXLC(LINE, B)); POPWORD;
END;
END;
JUMP1("LABEL"): BEGIN READARG;
WRITE2(SLDCO_2, ARG1, W); WRITELOCATION;
END;
FALSEJUMP1("LABEL"): BEGIN READARG;
WRITE2(SLDCO_2, ARG1, W); WRITELOCATION;
END;
END;
CASEJUMP1("MIN, MAX, LABELS"): "GENERATE RANGE CHECK ON CASE SELECTOR VALUE (TOS)
AND PUT CASEJUMP OPERATOR INTO CODE STREAM.
OPERAND TO XREF IS OFFSET (IN WORDS, FROM START
OF SEGMENT) INTO CONSTANT POOL WHERE CASE TABLE
WILL BE FOUND AT RUN TIME. CASE TABLE CONSISTS
OF MIN, CASE INDEX (1 WORD), MAX. CASE INDEX
C 1380 (1 WORD), AND JUMP DISPLACEMENTS (1 WORD EACH).*
C 1390 BEGIN READARG;
C 1380 IF ARG2-ARG1 > 0 458
C 1381 THEN BEGIN 558 558
C 1382 PICK_PUSHCONST(ARG1); PUSHWORD; "PUSH MIN" 653 558 608
C 1383 PICK_PUSHCONST(ARG2); PUSHWORD; "PUSH MAX" 653 558 608
C 1384 WRITE1(CHR,2); POP(TMWORD); "READCHECK" 504 208 646 247
C 1385 "LINE@.CONSTANTS" IS AMOUNT OF SPACE NEEDED IN CONSTANT
C 1386 TABLE FOR LONG CONSTANTS ONLY (DOES NOT INCLUDE SPACE
C 1387 NEEDED FOR FOR CASE TABLES).*
C 1388 "XJIFFOFFSET" POINTS TO WORD LAST FILLED.*
C 1389 BEGIN WRITE2(XJIP, (ORIGIN+LINE@.CONSTANTS) DIV WORDLENGTH) + 511 211 246 349 288 28
C 1390 XJIFFOFFSET, D); POPWORD; 289 269 615
C 1391 "BUILD CASE TABLE"
C 1392 XJIFFOFFSET += SUCCESS(XJIFFOFFSET);
C 1393 ENTER(XJITABLE, XJIFFOFFSET, ARG1); "MIN" 802 938 208 558
C 1394 XJIFFOFFSET += SUCCESS(XJIFFOFFSET);
C 1395 ENTER(XJITABLE, XJIFFOFFSET, ARG2); "MAX" 802 938 208 558
C 1396 "REMOVE CASE LABELS FROM CODE STREAM AND PUT
C 1397 CORRESPONDING OFFSETS IN CASE TABLE."
C 1398 ARG3 := ARG1; 558 558
C 1399 REPEAT
C 1400 READARG; "READ A CASE LABEL" 455
C 1401 ARG3 := ENTR(JUMP_TABLE, ARG1) - LOCATION; 558 814 282 558 303
C 1402 XJIFFOFFSET += SUCCESS(XJIFFOFFSET);
C 1403 ENTER(XJITABLE, XJIFFOFFSET, ARG3); 289 373 289 558
C 1404 ARG3 := ARG3 + 1;
C 1405 UNTIL ARG3 > ARG2; 558 558
C 1406 END;
C 1407 END;
C 1411 "HITVAR(L) : NEVER PRODUCED BY PASS 5.
C 1412 BEGIN READARG;
C 1413 IF Check THEN WRITE2(INITVAR2, ARG1 DIV WORDLENGTH);
C 1414 END; 
C 1415 "CALHI"(MODE, LABEL, PARAM, LENGTH): "****
C 1416 BEGIN READARG;
C 1417 IF ARG1 = MODE) "IF PROCESS ENTRY"
C 1418 THEN BEGIN
C 1419 PICK_PUSHCONST(ARG2); PUSHWORD; "PREFI ROUTINE NO."
C 1420 WRITE3(CHR,2, INITSEQ, STARTPROC, IN, UD); 529 195 254 255 269 249
C 1421 ARG1 := WORDLENGTH 558 28
C 1422 END;
C 1423 ELSE BEGIN
C 1424 WRITE2(CHL,2, ARG2, UD); PUSH(MSCVSIZE);
C 1425 IF Concurrent 243
C 1426 THEN ARG1 := ENTR(STACKTABLE, ARG2) 558 814 282 558
C 1427 ELSE ARG1 := WORDLENGTH 558 28
C 1428 END;
C 1429 PUSH(ARG1); POP(ARG1 + ARG3); 619 558 646 558 558
C 1430 END;
C 1431 "ENTER1"(MODE, LABEL, PARAM, LENGTH, VARNLENGTH, TEMPLENGTH): 128
C 1432 BEGIN READARG;
C 1433 BLOCK := ARG2; 302 558
C 1434 PARAMLENGTH := ARG3; 298 558
C 1435 VARNLENGTH := ARG4; 297 558
C 1436 STACKLENGTH := ARG5; 291 308
CASE ARG1 OF  
558  
559  "PROCEDURE"  
560  
561  BEGIN  
562  ENTER (DATASIZABLE, BLOCK, (VARLENGTH + TWOWORDS));  
563  802 283 302 297 247  
564  DIV WORDLENGTH;  
565  28  
566  POLENGTH := PARAMLENGTH + VARLENGTH + TWOWORDS;  
567  299 298 297 247  
568  PICK_PUSHCONSTR(LINE);  
569  PUSHWORD;  
570  653 306 608  
571  WRITE (STL_.?, FIXDL_.LINE, B);  
572  PUSHWORD;  
573  511 209 258 269 615  
574  END;  
575  
576  "PROGRAM"  
577  
578  PRINT "JOB" " 
579  
580  GEN_SAVELINE_SAVEBASE;  
581  665  
582  "MAKE GLOBAL BASE SAME AS PRESENT LOCAL BASE"  
583  
584  PICK_PUSHCONSTR(BASE_RED);  
585  PUSHRD;  
586  653 333 608  
587  PICK_PUSHCONSTR(BASE_RED);  
588  PUSHRD;  
589  653 333 608  
590  WRITE (LNR_2);  
591  564 193  
592  WRITE (SNR_2);  
593  504 210 646 247  
594  END;  
595  
596  "PROCESS ENTRY"  
597  
598  BEGIN  
599  ENTER (DATASIZABLE, BLOCK, (VARLENGTH + TWOWORDS));  
600  802 283 302 297 247  
601  DIV WORDLENGTH;  
602  28  
603  POLENGTH := VARLENGTH + TWOWORDS;  
604  299 297 247  
605  GEN_SAVELINE_SAVEBASE;  
606  665  
607  "RE-ESTABLISH OLD GLOBAL BASE"  
608  
609  PICK_PUSHCONSTR(BASE_RED);  
610  PUSHRD;  
611  653 333 608  
612  WRITE (OLDLNR_2);  
613  504 108 608  
614  WRITE (SNR_2);  
615  504 210 646 247  
616  "SET 'DON' := 0 " "###"  
617  
618  END;  
619  
620  "CLASS ENTRY"  
621  
622  "CLASS"  
623  
624  "POLENGTH' MUST INCLUDE THE LOCATION CONTAINING  
625  
626  THE ADDRESS OF THE CLASS VARIABLE.  
627  
628  THE NEW GLOBAL BASE RED MUST POINT A FEW WORDS  
629  
630  BELOW FIRST GLOBAL VARIABLE. THE 'FEN WORDS'  
631  
632  ARE THE PCW HARDWARE TAKES THE SIZE OF THE  
633  
634  PCW INTO ACCOUNT WHEN ACCESSING VARIABLES."  
635  
636  BEGIN  
637  IF ARG1 = MODEL  
638  THEN BEGIN "CLASS ENTRY"  
639  ENTER (DATASIZABLE, BLOCK, (VARLENGTH + TWOWORDS));  
640  802 283 302 297 247  
641  DIV WORDLENGTH;  
642  28  
643  POLENGTH := PARAMLENGTH + ONWORD +  
644  
645  VARLENGTH + TWOWORDS;  
646  299 298 297 247  
647  END;  
648  ELSE BEGIN "CLASS INITIAL STATEMENT"  
649  ENTER (DATASIZABLE, BLOCK, (TWOWORDS DIV WORDLENGTH));  
650  802 283 302 247 28  
651  POLENGTH := ONWORD + TWOWORDS;  
652  299 246 247  
653  END;  
654  
655  GEN_SAVELINE_SAVEBASE;  
656  865  
657  "SET NEW GLOBAL BASE"  
658  
659  PICK_PUSHCONSTR (BASE_RED);  
660  PUSHRD;  
661  653 333 608  
662  WRITE (LNR_2, POLENGTH DIV WORDLENGTH, B);  
663  PUSHRD;  
664  511 192 299 28 269 608  
665  PICK_PUSHCONSTR (HICWSIZE);  
666  PUSHRD;  
667  653 241 608
WRITE1 (SRL_2);  FOP;  508  199  615  247
WRITE1 (SFR_2);  POP (TWOWORDS);  507  210  646  247
END;

MODES:  "MONITOR ENTRY"  114
MODE2:  "MANAGER ENTRY"  151

BEGIN
ENTER (DATASIZETABLE, BLCK, VARLENGTH + TWOWORDS);  802  283  302  247  247
DIV WORDLENGTH;  28
POPLENGTH := VARLENGTH + ONEWORD + VARLENGTH + TWOWORDS;  299  298  246  247  247
GEN_SAVEOLD_SAVEBASE;  895
"NEW GLOBAL BASE"  151

PICK_PICKCONST (BASE_REG);  PUSHWORD;
WRITE2 (IN_.L_2, POPLENGTH DIV WORDLENGTH, D);  PUSHWORD;
WRITE2 (IN_.L_2, PICC, WORDSIZE, D);  PUSHWORD;
WRITE1 (SRL_2);  POP;  508  199  615
WRITE1 (SFR_2);  POP (TWOWORDS);  507  210  646  247
"KERNEL CALL TO ENTER GATE"  "###"
END;

MODES:  "PROCESS"  115

BEGIN
ENTER (DATASIZETABLE, BLCK, TWOWORDS DIV WORDLENGTH);
POPLENGTH := TWOWORDS;
PICK_PICKCONST (LINE);
WRITE2 (STL_2, IN_.L_2, D);  POPWORD;
WRITE1 (SRL_2);  POP;  511  200  218  269  615
END;

MODES:  "MONITOR"  117
MODE2: "MANAGER"  "###"

BEGIN
ENTER (DATASIZETABLE, BLCK, TWOWORDS DIV WORDLENGTH);
POPLENGTH := ONEWORD + TWOWORDS;
GEN_SAVEOLD_SAVEBASE;
"SET NEW GLOBAL BASE"  151

PICK_PICKCONST (BASE_REG);  PUSHWORD;
WRITE2 (IN_.L_2, POPLENGTH DIV WORDLENGTH, D);  PUSHWORD;
WRITE2 (IN_.L_2, PICC, WORDSIZE);  PUSHWORD;
WRITE1 (SRL_2);  POPWORD;
WRITE1 (SFR_2);  POP (TWOWORDS);
"KERNEL CALL TO INITIALIZE GATE"  "###"
END;

MODE10:  "UNDEFINED"  149

END;

RETURN"(MODE)":  129
BEGIN READIN;
"CURRENT LOCATION IN THIS ROUTINE'S 'EXIT IC'
ENTER (EXITTABLE, BLCK, LOCATION);
CASE ARG1 OF

MODE1: BEGIN  "PROCEDURE WRITE"
WRITE2(NPUL_2, POPLENGTH DIV WORDLENGTH, D);
POPOP (POPLENGTH + PICC);
END;

MODE2: BEGIN  "PROGRAM WRITE"  "###"
"PUSH CONTINUE"  141
WRITE1(SLDCO_2);
WRITE2(NPUL_2, "DISTANCE TO CONO[", SR);
WRITE1(SLDCO_2);
"STORE IN 'RESULT'"
"CONO:
"PUSH POINTER TO 'LINE'"
C 1560 WRITE(SLDO1_2);
C 1561 WRITE(STO_2);
C 1562 *PUSH 'JOB'*
C 1563 WRITE(SLDCOL_2);
C 1564 WRITE(FIJ_2, "DISPLACEMENT TO SEQ1: ", SN);
C 1565 "KERNEL CALL--SYSTEM ERROR"
C 1566 WRITE(FJUP_2, "DISPLACEMENT TO DONE: ", SN);
C 1567 "SEQ: 
C 1568 WRITE(SLDOCE_2);
C 1569 WRITE(SLDOGU_2);
C 1570 WRITE(SNR);
C 1571 "CLEAR 'JOB'"
C 1572 *DONE:**
C 1573 WRITE(FJU_2, POPLENGTH DIV WORDLENGTH, B) 511 196 299 28 269
C 1574 END;
C 1575 MODE2: BEGIN "PROCESS ENTRY MODE"
C 1576 Gen_EXIT; "JOB" = SUC('JOB') **
C 1577 END;
C 1578 MODE3: Gen_EXIT; "CLASS ENTRY MODE"
C 1579 MODE4: "MONITOR ENTRY"
C 1580 MODE5: BEGIN "MANAGER ENTRY MODE"
C 1581 "KERNEL CALL TO LEAVE GATE" **
C 1582 Gen_EXIT 909
C 1583 END;
C 1584 MODE6: "PROCESS MODE"
C 1585 WRITE(FJU_2, POPLENGTH DIV WORDLENGTH, B); 511 196 299 28 269
C 1586 MODE7: Gen_EXIT; "CLASS MODE"
C 1587 MODE8: "MONITOR MODE"
C 1588 MODE9: BEGIN "MANAGER MODE"
C 1589 "KERNEL CALL TO LEAVE GATE" **
C 1590 Gen_EXIT 909
C 1591 END;
C 1592 MODE10: "UNDEFINED MODE"
C 1593 END;
C 1594 IF (STACKLENGTH + MAXTEMP + VARLENGTH) < STACK_LIMIT 291 301 297 248
C 1595 THEN STACKLENGTH := STACKLENGTH + MAXTEMP + VARLENGTH 291 291 301 297
C 1596 ELSE ERROR (THIS PASS, STACK_ERROR)
C 1597 ENTER (STACKTABLE, BLOCK, STACKLENGTH); 802 282 302 291
C 1598 AFTERBEGIN := FALSE; 277 337
C 1599 IF (LOCATION MOD 2) = 1 "IF NOT ON WORD BOUNDARY"
C 1600 THEN WRITE(HOF_2) "THEN PAD TO A WORD BOUNDARY"
C 1601 END;
C 1602 POP('*LENGTH*'); 129
C 1603 BEGIN READARG;
C 1604 WHILE ARG1 >= TWOWORDS DO 455
C 1605 BEGIN
C 1606 WRITE(FJU_2, 0, SB); POP(TWOWORDS);
C 1607 ARG1 := ARG1 - TWOWORDS;
C 1608 END;
C 1609 END;
C 1610 IF ARG1 = ONEWORD
C 1611 THEN BEGIN WRITE(FJC_2, 0, SN); POPWORD; END;
C 1612 END;
C 1613 NEWLINE1 (*NUMBER*);
C 1614 "LOCAL WORD 1 ALWAYS RESERVED FOR LINE #.*
C 1615 BEGIN READARG;
C 1616 WHILE ARG1 := ARG1;
C 1617 LINE := ARG1;
C 1618 AFTERERROR := FALSE;
C 1619 IF (NUMBER AND AFTERBEGIN) 480 277
THEN BEGIN
  WRITEASS (HEPLIN_2, NOTHE): "CATCH FOR MNEMONICS PROGRAM"
  PICK_PUSHPUSH1(ARG1); pushword;
  WRITE(ARG2, PUSHPUSH1, ARG1); forward;
  END;
END;
ERROR1:
  GENERATE: = FALSE;
CONSTANTS "(LENGTH, VALUE)"
  REMOVE LONG CONSTANT FROM CODE STREAM & PUT IN CONSTANT POOL.
BEGIN READYARG;
  ARG2 := ARG1;
  FOR ARG3 := 1 TO (ARG1+SOURCE_WORD_LENGTH-1) DIV SOURCE_WORD_LENGTH DO
BEGIN
  CONSTANTS := CONSTANTS + 1 "WORD"
END;
END;
"IF SRC_16"
BEGIN
  ENTER(CONSTTABLE, CONSTANTS, ARG1);
ENDIF
"ENDIF"
BEGIN
  SPLIT_CONST(ARG1, ARG2, ARG3, ARG4);
  ENTER(CONSTTABLE, CONSTANTS, ARG1);
  IF (ARG3-1)*SOURCE_WORD_LENGTH + WORDLENGTH < ARG2 THEN BEGIN
    CONSTANTS := 1
    ENTER(CONSTTABLE, CONSTANTS, ARG1)
END
"ENDIF"
BEGIN
  ENTER(CONSTTABLE, CONSTANTS, ARG1)
END;
MESSAGE1("PASS, ERROR"):
ERROR(ARG1, ARG2)
END;
BEGIN
  INCREMENT1:
BEGIN
  WHITE1(DUPL_2); pushword;
  WHITE1(LINK_2, 1, UN); pushword;
  WHITE1(SLDCO_2); pushword;
  WHITE1(ADJ_2); forward;
  WHITE1(STO_2); pop(twowords);
END;
INCREMENT1:
BEGIN
  WHITE1(DUPL_2); pushword;
  WHITE1(LINK_2, 1, UN); pushword;
  WHITE1(SLDCO_2); pushword;
  WHITE1(ADJ_2); forward;
  WHITE1(STO_2); pop(twowords);
END;
PROCEDURE *(STANDARDPROCEDURE)*: ***

IF (ARG1 > MIN_PROC) AND (ARG1 <= MAX_PROC) THEN
CASE ARG1 OF
1: DELAY1:
BEGIN WRITEI(DELAY1); POPW END;
CONTINUE:
BEGIN WRITEI(CONTINUE); POPW END;
I01:
BEGIN WRITEI(I01); POP(TWOARGS) END;
START:
BEGIN WRITEI(START); POPW END;
STOP1:
BEGIN WRITEI(STOP1); POP(TWOARGS) END;
SETHEAP:
BEGIN WRITEI(SETHEAP); POPW END;
WAIT1:
BEGIN WRITEI(WAIT1); POPW END;
END;

INITI*(MODE, LABEL, PARAMETERS, VARENGTH)*:
BEGIN READARG;
IF ARG1 = MODE "IF PROCESS MODE"
THEN BEGIN
"KERNEL CALL TO INIT PROCESS" ***
BEGIN WRITEI(FP2, 0, 0); POPW
END
ELSE BEGIN "CLASS OR MONITOR"
BEGIN WRITEI(STM2, ARG3 DIV WORDLENGTH, UB);
PUSH (ARG3 DIV WORDLENGTH + 1);
BEGIN WRITEI(CPL2, ARG2, UB);
PUSH (HSMSTIP);
END
END

PUSHLABEL1*(LABEL)*: GEN_INTERFACE;

CALLPROC1:
BEGIN
"ADDRESS OF SEQ'L PROGRAM VARIABLE ON TOP"
"PUT IT IN SEQ'L FOR'S SIB"
BEGIN WRITEI(SLCODE1, 2); POPW; "GET @ OF CURRENTLY RUNNING TIB"
BEGIN WRITEI(MCI2); POPW
BEGIN WRITEI(LFH2); POPW
BEGIN WRITEI(MN2, SIB_OFFSET, B); "GET ADDRESS OF SIB VECTOR"
BEGIN WRITEI(INC, 5, B): "SORAGE FIELD IN SIB FOR SEGMENT" 
129 (SEQ'L PCH) IS 5 WORDS FROM
START OF SIB VECTOR."

"AFTER EXECUTION OF THE 'INC (5)', SEQ'L PROGRAM VARIABLE
15 IN THE TOP-1 WORD, AND @ OF THE SORAGE FIELD IN SIB
NO. 129 IS IN TOP WORD."
BEGIN WRITEI(SNAP2); POPW
BEGIN WRITEI(STO2); POP(TWOARGS); "PUT @ PCH TOP IN SIB"
BEGIN WRITEI(CLI2, SRL, SGL, STM, UB, UB); "CALL SEQ'L PCH"
BEGIN WRITEI(SNAP1); GETARGS "DISTANCE OF 'POP' VIRTUAL OPERATOR"
BEGIN READDARG; "MICROENGINE."
END;
EOM"(VARLENGTH)";
BEGIN
DONE:=TRUE;
READARG; VARLENGTH:=ARG1;
ENTR (DATASIZEABLE, STARTPROC,
(VARLENGTH + TWOWORDS) DIV WORDLENGTH);
WRITEARG(EOM, 2, NOARG);
END;

, ,
DUPPOST: BEGIN WRITE(DUPL_2); PUSHWORD END
END
UNTIL DONE
END;

BEGIN
INIT_PASS(LINK);
BEGIN_PASS;
SCAR;
END_PASS;
INIT_PASS(LINK);
END.
<table>
<thead>
<tr>
<th>ARL_2</th>
<th>214</th>
<th>1235</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARL_2</td>
<td>214</td>
<td>1236</td>
</tr>
<tr>
<td>ARS</td>
<td>420</td>
<td></td>
</tr>
<tr>
<td>ARS1</td>
<td>160</td>
<td>1233</td>
</tr>
<tr>
<td>ARD</td>
<td>179</td>
<td>193</td>
</tr>
<tr>
<td>ARD_2</td>
<td>207</td>
<td></td>
</tr>
<tr>
<td>ARD_2</td>
<td>205</td>
<td>1199</td>
</tr>
<tr>
<td>ATERNATING</td>
<td>279</td>
<td>595</td>
</tr>
<tr>
<td>ATERNATING</td>
<td>789</td>
<td>848</td>
</tr>
<tr>
<td>ARD_1</td>
<td>123</td>
<td>1169</td>
</tr>
<tr>
<td>ARG</td>
<td>414</td>
<td>471</td>
</tr>
<tr>
<td>ARG</td>
<td>601</td>
<td>693</td>
</tr>
<tr>
<td>ARG</td>
<td>751</td>
<td>752</td>
</tr>
<tr>
<td>ARG1</td>
<td>308</td>
<td>456</td>
</tr>
<tr>
<td>ARG1</td>
<td>1051</td>
<td>1056</td>
</tr>
<tr>
<td>ARG1</td>
<td>1208</td>
<td>1215</td>
</tr>
<tr>
<td>ARG1</td>
<td>1400</td>
<td>1419</td>
</tr>
<tr>
<td>ARG1</td>
<td>1681</td>
<td>1695</td>
</tr>
<tr>
<td>ARG11</td>
<td>1039</td>
<td>1695</td>
</tr>
<tr>
<td>ARG12</td>
<td>1039</td>
<td>1685</td>
</tr>
<tr>
<td>ARG13</td>
<td>525</td>
<td>531</td>
</tr>
<tr>
<td>ARG2</td>
<td>308</td>
<td>459</td>
</tr>
<tr>
<td>ARG2</td>
<td>1059</td>
<td>1068</td>
</tr>
<tr>
<td>ARG2</td>
<td>1408</td>
<td>1423</td>
</tr>
<tr>
<td>ARG2TTP</td>
<td>525</td>
<td>531</td>
</tr>
<tr>
<td>ARG3</td>
<td>308</td>
<td>462</td>
</tr>
<tr>
<td>ARG3TTP</td>
<td>542</td>
<td>554</td>
</tr>
<tr>
<td>ARG4</td>
<td>308</td>
<td>467</td>
</tr>
<tr>
<td>ARG4</td>
<td>559</td>
<td>571</td>
</tr>
<tr>
<td>ARG5</td>
<td>308</td>
<td>473</td>
</tr>
<tr>
<td>ARGLIST</td>
<td>91</td>
<td>108</td>
</tr>
<tr>
<td>ARGTXT</td>
<td>78</td>
<td>82</td>
</tr>
<tr>
<td>ARGTXT</td>
<td>511</td>
<td>520</td>
</tr>
<tr>
<td>ARGTXT</td>
<td>81</td>
<td>91</td>
</tr>
<tr>
<td>ASSGINA1</td>
<td>122</td>
<td>1126</td>
</tr>
<tr>
<td>ASSIGNA1</td>
<td>122</td>
<td>129</td>
</tr>
<tr>
<td>ATTRIBUTE1</td>
<td>161</td>
<td>1254</td>
</tr>
</tbody>
</table>

---

| R   | 269 | 407 | 717 | 720 | 724 | 725 | 729 | 740 | 744 | 747 | 751 | 752 | 756 | 902 | 906 | 915 | 969 | 1014 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| R   | 1073 | 1085 | 1107 | 1125 | 1321 | 1322 | 1333 | 1334 | 1335 | 1392 | 1395 | 1406 | 1512 | 1523 | 1533 | 1549 |
| R   | 1573 | 1585 | 1623 | 1725 | 1726 |
| RREGMPASS | 830 | 1761 |
| RCONT | 102 | 103 | 302 | 336 | 347 | 365 | 386 | 392 | 406 | 420 | 428 | 436 | 451 | 468 | 1512 | 1523 | 1533 | 1549 |
| RCONT | 1573 | 1585 | 1623 | 1725 | 1726 |
| RCONT | 830 | 1761 |
| RCONT | 102 | 103 | 302 | 336 | 347 | 365 | 386 | 392 | 406 | 420 | 428 | 436 | 451 | 468 | 1512 | 1523 | 1533 | 1549 |
| RCONT | 1573 | 1585 | 1623 | 1725 | 1726 |
| RCONT | 830 | 1761 |

---

<table>
<thead>
<tr>
<th>B</th>
<th>126</th>
<th>1261</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>135</td>
<td>766</td>
</tr>
<tr>
<td>Variable</td>
<td>Value</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>90.99</td>
<td></td>
</tr>
<tr>
<td>CALL1</td>
<td>120° 141°</td>
<td></td>
</tr>
<tr>
<td>CALLERBONDIS</td>
<td>257° 1095° 1111°</td>
<td></td>
</tr>
<tr>
<td>CALLPOS1</td>
<td>131° 1718°</td>
<td></td>
</tr>
<tr>
<td>CAPOS1</td>
<td>120° 137°</td>
<td></td>
</tr>
<tr>
<td>CHN</td>
<td>95° 92° 99° 99°</td>
<td></td>
</tr>
<tr>
<td>CHECK</td>
<td>276° 837°</td>
<td></td>
</tr>
<tr>
<td>CHK_2</td>
<td>208° 1080° 1122° 1266° 1386°</td>
<td></td>
</tr>
<tr>
<td>CHR</td>
<td>82° 485°</td>
<td></td>
</tr>
<tr>
<td>CODE</td>
<td>39° 869°</td>
<td></td>
</tr>
<tr>
<td>CODE_ERROR</td>
<td>256° 494°</td>
<td></td>
</tr>
<tr>
<td>COMN_LIMIT</td>
<td>428° 404°</td>
<td></td>
</tr>
<tr>
<td>COMPARE1</td>
<td>126° 1276°</td>
<td></td>
</tr>
<tr>
<td>COMPSTRUCT</td>
<td>126° 1318°</td>
<td></td>
</tr>
<tr>
<td>CONCURRENT</td>
<td>243° 1027°</td>
<td></td>
</tr>
<tr>
<td>CONSID</td>
<td>932° 1008° 1027°</td>
<td></td>
</tr>
<tr>
<td>CONSTANT1</td>
<td>130° 1630°</td>
<td></td>
</tr>
<tr>
<td>CONSTANTS</td>
<td>72° 268° 844° 851° 853° 866° 866° 866° 1391° 1637° 1637° 1641° 1646° 1649° 1649° 1650°</td>
<td></td>
</tr>
<tr>
<td>CONTIABLE</td>
<td>55° 82° 151° 82°</td>
<td></td>
</tr>
<tr>
<td>CONTENTS</td>
<td>51° 81° 82°</td>
<td></td>
</tr>
<tr>
<td>CONTINUE1</td>
<td>166°</td>
<td></td>
</tr>
<tr>
<td>COPY1</td>
<td>161° 1287°</td>
<td></td>
</tr>
<tr>
<td>COPY2</td>
<td>122° 1153°</td>
<td></td>
</tr>
<tr>
<td>CRP_2</td>
<td>195°</td>
<td></td>
</tr>
<tr>
<td>CRM_2</td>
<td>195°</td>
<td></td>
</tr>
<tr>
<td>CFI_2</td>
<td>195° 1226° 1711°</td>
<td></td>
</tr>
<tr>
<td>CLE_2</td>
<td>195°</td>
<td></td>
</tr>
<tr>
<td>CII_2</td>
<td>195°</td>
<td></td>
</tr>
<tr>
<td>CII_2</td>
<td>195° 1008° 1422° 1735°</td>
<td></td>
</tr>
<tr>
<td>DATA</td>
<td>928° 905° 1000°</td>
<td></td>
</tr>
<tr>
<td>DATASIZE</td>
<td>936° 962° 1014°</td>
<td></td>
</tr>
<tr>
<td>DATASIQUEL</td>
<td>56° 287° 847° 883° 883° 1447° 1465° 1486° 1492° 1506° 1520° 1528° 1745°</td>
<td></td>
</tr>
<tr>
<td>DB</td>
<td>269° 486° 725° 752°</td>
<td></td>
</tr>
<tr>
<td>DECREMENT1</td>
<td>130° 1671°</td>
<td></td>
</tr>
<tr>
<td>DEPLABEL1</td>
<td>127° 1352°</td>
<td></td>
</tr>
<tr>
<td>DELAY1</td>
<td>166°</td>
<td></td>
</tr>
<tr>
<td>DIF_2</td>
<td>213° 1202°</td>
<td></td>
</tr>
<tr>
<td>DIGIT</td>
<td>415° 417° 419° 419° 420° 422° 423° 424°</td>
<td></td>
</tr>
<tr>
<td>DFLM</td>
<td>666° 675° 680° 681° 692° 693° 709° 717° 720° 724° 725° 729° 740° 744° 747° 751° 752° 756°</td>
<td></td>
</tr>
<tr>
<td>DIV1</td>
<td>125° 1213°</td>
<td></td>
</tr>
<tr>
<td>DONE</td>
<td>278° 1044° 1733° 1756°</td>
<td></td>
</tr>
<tr>
<td>DBP1_2</td>
<td>214° 1093° 1106° 1267° 1664° 1673° 1751°</td>
<td></td>
</tr>
<tr>
<td>DBP2_2</td>
<td>207°</td>
<td></td>
</tr>
<tr>
<td>DPTOS1</td>
<td>132° 1751°</td>
<td></td>
</tr>
<tr>
<td>DTL_2</td>
<td>198° 1216°</td>
<td></td>
</tr>
<tr>
<td>DTR_2</td>
<td>206° 1211°</td>
<td></td>
</tr>
<tr>
<td>EF1_2</td>
<td>210° 1095° 1111°</td>
<td></td>
</tr>
<tr>
<td>EMPT1</td>
<td>161° 1249°</td>
<td></td>
</tr>
<tr>
<td>ENDPASS</td>
<td>864° 1763°</td>
<td></td>
</tr>
<tr>
<td>ENTER</td>
<td>802° 860° 1394° 1396° 1398° 1406° 1407° 1465° 1466° 1466° 1502° 1506° 1520° 1528° 1546° 1597° 1611° 1646° 1650°</td>
<td></td>
</tr>
<tr>
<td>ENTER1</td>
<td>120° 1934°</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Value</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------</td>
<td></td>
</tr>
<tr>
<td>LDC_2</td>
<td>191</td>
<td></td>
</tr>
<tr>
<td>LNE_2</td>
<td>197</td>
<td></td>
</tr>
<tr>
<td>LUL_2</td>
<td>192, 717, 723, 1898, 1512, 1533</td>
<td></td>
</tr>
<tr>
<td>LML_2</td>
<td>210, 771, 775, 1665, 1674</td>
<td></td>
</tr>
<tr>
<td>LPO_2</td>
<td>192, 720, 729</td>
<td></td>
</tr>
<tr>
<td>LDP_2</td>
<td>208</td>
<td></td>
</tr>
<tr>
<td>LENGTH</td>
<td>639, 641, 641, 646, 647</td>
<td></td>
</tr>
<tr>
<td>LQUNT_2</td>
<td>208, 1323, 1326</td>
<td></td>
</tr>
<tr>
<td>LEQ1_2</td>
<td>207, 1283, 1286</td>
<td></td>
</tr>
<tr>
<td>LEQMAH_2</td>
<td>203, 1308</td>
<td></td>
</tr>
<tr>
<td>LQRAV_2</td>
<td>200, 1295, 1298</td>
<td></td>
</tr>
<tr>
<td>LESS</td>
<td>155, 1281, 1293, 1321</td>
<td></td>
</tr>
<tr>
<td>LQUSH_2</td>
<td>201</td>
<td></td>
</tr>
<tr>
<td>LINE</td>
<td>306, 598, 855, 901, 1347, 1450, 1522, 1617</td>
<td></td>
</tr>
<tr>
<td>LITE</td>
<td>273, 310, 312, 319, 362, 883, 853, 856, 866, 867, 871, 872, 876, 1391, 1756, 1764</td>
<td></td>
</tr>
<tr>
<td>LGTOPTION</td>
<td>38, 59</td>
<td></td>
</tr>
<tr>
<td>LRAV</td>
<td>192, 771, 751</td>
<td></td>
</tr>
<tr>
<td>LQNT_2</td>
<td>215, 1167, 1281, 1283, 1293, 1295, 1297, 1307, 1321, 1323, 1325</td>
<td></td>
</tr>
<tr>
<td>LOCATION</td>
<td>303, 808, 488, 488, 488, 488, 489, 490, 495, 587, 588, 855, 878, 1354, 1404, 1443</td>
<td></td>
</tr>
<tr>
<td>LUB_2</td>
<td>193, 725</td>
<td></td>
</tr>
<tr>
<td>LVR_2</td>
<td>199, 1179</td>
<td></td>
</tr>
<tr>
<td>LWDORD</td>
<td>889, 801</td>
<td></td>
</tr>
<tr>
<td>LFR_2</td>
<td>190, 905, 1450, 1724</td>
<td></td>
</tr>
<tr>
<td>LPL_2</td>
<td>197</td>
<td></td>
</tr>
<tr>
<td>LPLIM</td>
<td>230</td>
<td></td>
</tr>
</tbody>
</table>

**-H-**

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HAB1</td>
<td>108</td>
</tr>
<tr>
<td>HABE</td>
<td>105, 848, 977</td>
</tr>
<tr>
<td>MAXARG</td>
<td>89, 91</td>
</tr>
<tr>
<td>MAXDOUT</td>
<td>26, 415, 422, 424</td>
</tr>
<tr>
<td>MAXINPAC</td>
<td>61, 60, 286, 891</td>
</tr>
<tr>
<td>MAXTEMP</td>
<td>301, 612, 612, 623, 623, 623, 623, 643, 643, 1440, 1504, 1595</td>
</tr>
<tr>
<td>MAXWARN</td>
<td>40, 51, 793, 798, 806, 809, 818, 821</td>
</tr>
<tr>
<td>MAX_FUNC</td>
<td>162, 1229</td>
</tr>
<tr>
<td>MAX_PROC</td>
<td>168</td>
</tr>
<tr>
<td>MESSAGE1</td>
<td>130, 1657</td>
</tr>
<tr>
<td>MESSAG2</td>
<td>217, 556</td>
</tr>
<tr>
<td>MININDEX</td>
<td>93, 956</td>
</tr>
<tr>
<td>MIN_FUNC</td>
<td>162, 1229</td>
</tr>
<tr>
<td>MIN_PROC</td>
<td>168</td>
</tr>
<tr>
<td>MODE</td>
<td>125, 1220</td>
</tr>
<tr>
<td>MODE0</td>
<td>666, 673, 711, 713, 717, 720, 724, 725, 729, 736, 738, 740, 744, 747, 751, 752, 756</td>
</tr>
<tr>
<td>MODE1</td>
<td>130, 674, 739</td>
</tr>
<tr>
<td>MODE2</td>
<td>110, 678, 716, 743, 1335, 1445, 1548</td>
</tr>
<tr>
<td>MODE10</td>
<td>110, 707, 731, 758, 1539, 1592</td>
</tr>
<tr>
<td>MODE11</td>
<td>150, 696, 728, 755, 1526, 1588</td>
</tr>
<tr>
<td>MODE12</td>
<td>151, 670, 716, 743, 1340, 1504, 1580</td>
</tr>
<tr>
<td>MODE2</td>
<td>141, 678, 719, 746, 1453, 1552</td>
</tr>
<tr>
<td>MODE3</td>
<td>142, 683, 722, 749, 1335, 1419, 1463, 1575</td>
</tr>
<tr>
<td>MODE4</td>
<td>143, 678, 716, 743, 1340, 1475, 1480, 1578</td>
</tr>
<tr>
<td>MODE5</td>
<td>144, 678, 716, 743, 1340, 1503, 1579</td>
</tr>
<tr>
<td>MODE6</td>
<td>145, 696, 728, 755, 1518, 1584, 1703</td>
</tr>
<tr>
<td>MODE7</td>
<td>146, 696, 728, 755, 1476, 1586</td>
</tr>
<tr>
<td>MODE8</td>
<td>147, 696, 728, 755, 1525, 1587</td>
</tr>
<tr>
<td>MODE9</td>
<td>148</td>
</tr>
<tr>
<td>MODL1</td>
<td>194, 1221</td>
</tr>
<tr>
<td>MIV_2</td>
<td>207, 1155</td>
</tr>
<tr>
<td>MFI_2</td>
<td>194, 1209</td>
</tr>
<tr>
<td>Variable</td>
<td>Value</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>MPR_2</td>
<td>206 1210</td>
</tr>
<tr>
<td>MUCOGSIZE</td>
<td>281 916</td>
</tr>
<tr>
<td>N1010</td>
<td>124 1206</td>
</tr>
<tr>
<td>NR</td>
<td>129 1183</td>
</tr>
<tr>
<td>NR_2</td>
<td>202 1285</td>
</tr>
<tr>
<td>NRW</td>
<td>792 797</td>
</tr>
<tr>
<td>NRN</td>
<td>792 797</td>
</tr>
<tr>
<td>NRW1</td>
<td>123 1158</td>
</tr>
<tr>
<td>NRW1_81</td>
<td>129 1614</td>
</tr>
<tr>
<td>NRW1_82</td>
<td>217 900</td>
</tr>
<tr>
<td>NRM</td>
<td>946 979</td>
</tr>
<tr>
<td>NEXT_PORTION</td>
<td>50 797</td>
</tr>
<tr>
<td>NEXIT_PASS</td>
<td>219 1764</td>
</tr>
<tr>
<td>NFI_2</td>
<td>219 1697</td>
</tr>
<tr>
<td>NCL_2</td>
<td>219 1186</td>
</tr>
<tr>
<td>NCL_2_2</td>
<td>215 1187</td>
</tr>
<tr>
<td>NHTYPE</td>
<td>79 83</td>
</tr>
<tr>
<td>NOP_2</td>
<td>198 970</td>
</tr>
<tr>
<td>NOT</td>
<td>123 1166</td>
</tr>
<tr>
<td>NQ2</td>
<td>155 1285</td>
</tr>
<tr>
<td>NOUTEQT</td>
<td>155 1285</td>
</tr>
<tr>
<td>NOUTEQUAL</td>
<td>155 1285</td>
</tr>
<tr>
<td>NOUTLESS</td>
<td>155 1285</td>
</tr>
<tr>
<td>NOUT</td>
<td>269 891</td>
</tr>
<tr>
<td>NOT_O</td>
<td>100 207</td>
</tr>
<tr>
<td>NOT</td>
<td>101 208</td>
</tr>
<tr>
<td>NUMBER</td>
<td>277 808</td>
</tr>
<tr>
<td>NODIDOPTION</td>
<td>39 839</td>
</tr>
<tr>
<td>NODNENTRY</td>
<td>929 986</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>318 337</td>
</tr>
<tr>
<td>OLDFIELDTOP</td>
<td>941 977</td>
</tr>
<tr>
<td>ONEND</td>
<td>246 489</td>
</tr>
<tr>
<td>OP</td>
<td>307 616</td>
</tr>
<tr>
<td>OPTIONS</td>
<td>59 71</td>
</tr>
<tr>
<td>OT</td>
<td>269 106</td>
</tr>
<tr>
<td>OR</td>
<td>123 1176</td>
</tr>
<tr>
<td>OR IF</td>
<td>420</td>
</tr>
<tr>
<td>OUTFILE</td>
<td>250 352</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>102 103</td>
</tr>
<tr>
<td>PAGE</td>
<td>96 96</td>
</tr>
<tr>
<td>PARMLNGTH</td>
<td>322 384</td>
</tr>
<tr>
<td>PAGES_IN</td>
<td>322 384</td>
</tr>
<tr>
<td>PAGES_OUT</td>
<td>322 384</td>
</tr>
<tr>
<td>PARAM</td>
<td>108 334</td>
</tr>
<tr>
<td>PARA_LENGTH</td>
<td>298 1437</td>
</tr>
<tr>
<td>PASS</td>
<td>80a 597</td>
</tr>
<tr>
<td>PC REQ</td>
<td>23g 282</td>
</tr>
<tr>
<td>PICK_PUSHCON</td>
<td>63g 777</td>
</tr>
</tbody>
</table>

**B-37**
<table>
<thead>
<tr>
<th>REALMEN</th>
<th>1042</th>
<th>1046</th>
<th>1736</th>
</tr>
</thead>
<tbody>
<tr>
<td>REALTIME</td>
<td>29</td>
<td>613</td>
<td>771</td>
</tr>
<tr>
<td>REALTIME1</td>
<td>164</td>
<td>235</td>
<td>1585</td>
</tr>
<tr>
<td>REALTYPE</td>
<td>870</td>
<td>1015</td>
<td>1219</td>
</tr>
<tr>
<td>RELEASE</td>
<td>445</td>
<td>420</td>
<td>421</td>
</tr>
<tr>
<td>REM</td>
<td>167</td>
<td>621</td>
<td>620</td>
</tr>
<tr>
<td>REPOINT</td>
<td>73</td>
<td>153</td>
<td>1546</td>
</tr>
<tr>
<td>RETURN</td>
<td>122</td>
<td>154</td>
<td>1543</td>
</tr>
<tr>
<td>BRM_2</td>
<td>285</td>
<td>1015</td>
<td>1518</td>
</tr>
<tr>
<td>BW</td>
<td>156</td>
<td>915</td>
<td>1500</td>
</tr>
<tr>
<td>BTW</td>
<td>942</td>
<td>951</td>
<td>957</td>
</tr>
</tbody>
</table>

The table continues with other entries...
<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>115</td>
</tr>
<tr>
<td>TABLE</td>
<td>40</td>
</tr>
<tr>
<td>TABLEPART</td>
<td>53</td>
</tr>
<tr>
<td>TABLEPRT</td>
<td>48</td>
</tr>
<tr>
<td>TABLESPRT</td>
<td>58</td>
</tr>
<tr>
<td>TAG</td>
<td>82</td>
</tr>
<tr>
<td>TACK_BRO</td>
<td>229</td>
</tr>
<tr>
<td>TEMP</td>
<td>633</td>
</tr>
<tr>
<td>TEST</td>
<td>276</td>
</tr>
<tr>
<td>TESTOPT</td>
<td>38</td>
</tr>
<tr>
<td>TEXT</td>
<td>326</td>
</tr>
<tr>
<td>TEXT_LENGTH</td>
<td>90</td>
</tr>
<tr>
<td>TEXT_TF</td>
<td>92</td>
</tr>
<tr>
<td>THRU_PASS</td>
<td>249</td>
</tr>
<tr>
<td>THREEWORDS</td>
<td>247</td>
</tr>
<tr>
<td>TMC</td>
<td>205</td>
</tr>
<tr>
<td>TOP</td>
<td>195</td>
</tr>
<tr>
<td>TOS</td>
<td>990</td>
</tr>
<tr>
<td>TOPR</td>
<td>926</td>
</tr>
<tr>
<td>TRUE</td>
<td>243</td>
</tr>
<tr>
<td>TRUNC1</td>
<td>160</td>
</tr>
<tr>
<td>TWORDS</td>
<td>247</td>
</tr>
<tr>
<td>TYPICODE</td>
<td>269</td>
</tr>
<tr>
<td>U</td>
<td></td>
</tr>
<tr>
<td>UFL</td>
<td>269</td>
</tr>
<tr>
<td>UFJ</td>
<td>193</td>
</tr>
<tr>
<td>URL</td>
<td>212</td>
</tr>
<tr>
<td>UPGLOC</td>
<td>482</td>
</tr>
<tr>
<td>USQNL</td>
<td>231</td>
</tr>
<tr>
<td>V</td>
<td></td>
</tr>
<tr>
<td>VALUE</td>
<td>400</td>
</tr>
<tr>
<td>VARIKEY</td>
<td>121</td>
</tr>
<tr>
<td>VARTYP</td>
<td>767</td>
</tr>
<tr>
<td>W</td>
<td></td>
</tr>
<tr>
<td>WAIT1</td>
<td>167</td>
</tr>
<tr>
<td>WAIT2</td>
<td>213</td>
</tr>
<tr>
<td>WORD</td>
<td>888</td>
</tr>
</tbody>
</table>
**WORDLENGTH**

<table>
<thead>
<tr>
<th>20</th>
<th>610</th>
<th>616</th>
<th>675</th>
<th>690</th>
<th>681</th>
<th>692</th>
<th>693</th>
<th>708</th>
<th>709</th>
<th>711</th>
<th>717</th>
<th>777</th>
<th>778</th>
<th>844</th>
<th>845</th>
<th>915</th>
<th>917</th>
<th>1073</th>
<th>1085</th>
</tr>
</thead>
<tbody>
<tr>
<td>1107</td>
<td>1135</td>
<td>1140</td>
<td>1155</td>
<td>1171</td>
<td>1172</td>
<td>1177</td>
<td>1192</td>
<td>1200</td>
<td>1208</td>
<td>1215</td>
<td>1238</td>
<td>1279</td>
<td>1336</td>
<td>1341</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**WORDS_IN**

| 322 | 346 | 369 | 375 | 377 | 377 | 378 |

**WORDS_OUT**

| 323 | 351 | 351 | 351 | 399 | 400 | 401 | 408 |

**WORDTYPE**

| 135 | 768 | 1056 | 1131 | 1171 | 1178 | 1185 | 1192 | 1200 | 1208 | 1215 | 1238 | 1279 | 1336 | 1341 |

**WRITE**

| 99 | 329 | 330 | 331 | 333 | 333 | 334 | 335 | 336 | 337 | 338 | 339 | 340 | 341 | 342 | 343 | 344 | 345 | 346 |

**WRITE1**

| 500 | 501 | 502 | 504 | 506 | 508 | 510 | 512 | 514 | 516 | 518 | 520 | 522 | 524 | 526 | 528 | 530 | 532 | 534 |

**WRITE2**

| 711 | 698 | 699 | 700 | 701 | 702 | 703 | 704 | 705 | 706 | 707 | 708 | 709 | 710 | 711 | 712 | 713 | 714 | 715 |

**WRITE3**

| 529 | 725 | 726 | 1006 | 1008 | 1222 | 1235 |

**WRITE4**

| 541 |

**WRITE5**

| 558 |

**WRITEBAD**

| 578 | 596 | 597 | 598 | 599 | 600 | 601 | 602 | 603 | 604 | 605 | 606 | 607 | 608 | 609 | 610 | 611 | 612 | 613 |

**WRITELOCATIO**

| 585 | 1364 | 1369 |

**WRITE_FIL**

| 381 | 507 | 517 | 517 | 532 | 533 | 533 | 534 | 535 | 536 | 537 | 538 | 539 | 540 | 541 | 542 | 543 | 544 | 545 |

**INDEX**

| 259 | 852 | 867 | 1392 | 1395 | 1396 | 1397 | 1397 | 1398 | 1405 | 1406 |

**LJPIFFST**

| 55 | 282 | 385 | 881 | 901 | 936 | 969 | 1396 | 1398 | 1406 |

**LJP**

| 218 | 969 | 1391 |

**LJ_OFFSET**

| 72 | 845 | 853 | 867 |

**END**

518 IDENTIFIERS 2678 TOTAL REFERENCES
745 COLLISIONS.
APPENDIX C

SOURCE CODE FOR MEPASCAL COMPILER DRIVER: MEPASCAL
10 TYPE FULLWORD = INTEGER;
11     INTEGER = SHORTINTEGER;
17
13 CONST HL = '(:10:);  FF = '(:12:);  CR = '(:13:);  EN = '(:25:);
18
15 CONST PAGELENGTH = 512;
16 TYPE PAGE = ARRAY (:1..PAGELENGTH.) OF CHAR;
17
18 CONST LINESLENGTH = 132;
19 TYPE LINE = ARRAY (:1..LINESLENGTH.) OF CHAR;
20
21 CONST IDLENGTH = 12;
22 TYPE IDENTIFIER = ARRAY (:1..IDLENGTH.) OF CHAR;
23
24 TYPE FILE = 1..2;
25
26 TYPE FILEKIND = (EMPTY, SCRATCH, ASCII, SECODE, CONCODE);
27
28 TYPE FILEATTR = RECORD
29     KIND: FILEKIND;
30     ADDR: INTEGER;
31     PROTECTED: BOOLEAN;
32     CONTROL: ARRAY (:1..5.) OF INTEGER
33 END;
34
35 TYPE IODEVICE =
36     (TYPEDEVICE, DISKDEVICE, TAPEDEVICE, PRINTDEVICE, CARDDEVICE);
37
38 TYPE IOOPERATION = (INPUT, OUTPUT, MOVE, CONTROL);
39
40 TYPE IOARG = (WRITETO, REMIND, UPSIZE, BACKSPACE);
41
42 TYPE IORESULT =
43     (COMPLETE, INTERVENTION, TRANSMISSION, FAILURE,
44     ENDFILE, ENDMEDIUM, STARTMEDIUM);
45
46 TYPE IOPARAM = RECORD
47     OPERATION: IOOPERATION;
48     STATUS: IORESULT;
49     ARG1: IOARG
50     PND;
51
52 TYPE TASKKIND = (INPUTTASK, JOBTASK, OUTPUTTASK);
53
54 TYPE ARITAG =
55     (WILTYPE, DATATYPE, INITYPE, IDTYPE, PRTYPE);
56
57 TYPE POINTER = #INTEGER;
58
59 TYPE PASSPTR = &PASSLINK;
PROCEDURE RUN(ID: IDENTIFIER; VAR PAR: ARGLIST;
  VAR LINE: INTEGER; VAR RESULT: PRO Ges); * * 22 * 82
PROCEDURE BREAKLINE(LINE: INTEGER);
PROCEDURE P(PARAM: LINE);
"THE PARAMETERS OF THE COMPILED PASSES"
HAVE THE FOLLOWING MEANING:
LIST(1.) : BOOLEAN (COMPILED OK)
LIST(2.) : POINTER (HEAP POINTER)
LIST(3.) : INTEGER (CODE LENGTH)
TYPE CHARSET = SET OF CHAR;
TYPE PASS = 0..17;
VAR
OK, PAST.OK: BOOLEAN; SOURCE, DEST, OBJECT: IDENTIFIER;
CODLENGTH: INTEGER;
WHERE: (MEMORY, ONDISK);
REPORT: (MAIN, OUT);
I: INTEGER;
LIST: ARGLIST;
PARAM: 1..LINELENGTH;
TESTPASS: SET OF PASSES;
HPSOURCE: BOOLEAN;
PROCEDURE INITWRITE;
BEGIN
IDENTIFY('PASCAL:(;10):');
REPORT := MAIN;
END;
PROCEDURE WRITECHAR(C: CHAR);
BEGIN
IF REPORT <> MAIN
  THEN WRITE(C);
DISPLAY(C);
END;
PROCEDURE WRI TEXT(TEXT: LINE);
CONST NUL = '\(10)';
VAR I, C, CHAR:)
180 BEGIN
181 i := 1; c := 'A'; 
182 WHILE c <> 'A' DO 
183 BEGIN 
184 WRITECHAR(c); 
185 i := i + 1; c := 'A'; 
186 END; 
187 END; 
188 
189 PROCEDURE WRITEINT(INT, LENGTH: INTEGER); 
190 VAR NUMBER: ARRAY [.1..6] OF CHAR; 
191 DIGIT, REM, i: INTEGER; 
192 BEGIN 
193 DIGIT := 0; REM := INT; 
194 REPEAT 
195 DIGIT := DIGIT + 1; 
196 NUMBER.(DIGIT) := 
197 CHAR(ASC(NUMBER MOD 10) + ORD('0')); 
198 REM := REM DIV 10; 
199 UNTIL REM = 0; 
200 FOR i := 1 TO LENGTH - DIGIT - 1 DO 
201 WRITECHAR(' '); 
202 IF INT < 0 THEN WRITECHAR('-'); 
203 ELSE WRITECHAR('+'); 
204 FOR i := DIGIT DOWNTO 1 DO 
205 WRITECHAR(NUMBER.(i)); 
206 END; 
207 
208 PROCEDURE WRITEID(ID: IDENTIFIER); 
209 VAR i: INTEGER; c: CHAR; 
210 BEGIN 
211 FOR i := 1 TO IDLENGTH DO 
212 BEGIN 
213 c := ID.(i); 
214 IF c <> ' ' THEN WRITECHAR(c); 
215 END; 
216 END; 
217 
218 PROCEDURE CONVRESULT(RESULT: PROGRESULT; VAR ID: IDENTIFIER); 
219 BEGIN 
220 CASE RESULT OF 
221 TERMINATED: ID := 'TERMINATED'; 
222 OVERFLOW: ID := 'OVERFLOW'; 
223 POINTERERROR: ID := 'POINTERERROR'; 
224 RUNERROR: ID := 'RUNERROR'; 
225 VARIANTERROR: ID := 'VARIANTERROR'; 
226 HEAPLIMIT: ID := 'HEAPLIMIT'; 
227 STACKLIMIT: ID := 'STACKLIMIT'; 
228 CONSLIMIT: ID := 'CONSLIMIT'; 
229 TIMELIMIT: ID := 'TIMELIMIT'; 
230 CALLERROR: ID := 'CALLERROR'; 
231 END; 
232 END; 
233 
234 PROCEDURE WRITEMESULT 
235 (ID: IDENTIFIER; LINE: INTEGER; RESULT: PROGRESULT); 
236 VAR ARG: IDENTIFIER; 
237 BEGIN 
238 WRITECHAR(INL); 
239 WRITEINT(ID); 
240
240  WRITETEXT('L1INE (10:1)');  177
241  WRITETEXT('LINE, I');  169 235
242  WRITETEXT('I');  170
243  CONFRESULT('RESULT, AND');  218 235 216
244  WRITETEXT('AND');  708 236
245  WRITETEXT('N');  170 13
246  OK := (RESULT = TERMINATED);  151 235 87
247  END;
248
249  PROCEDURE ERROR(TEXT: LINE);  235
250  BEGIN
251  WRITETEXT;  164
252  WRITETEXT(TEXT);  177 249
253  OK := FALSE;  151 *
254  END;
255
256  FUNCTION NEXTCHAR: CHAR;
257  BEGIN
258  PARAMPTR := PARAMPTR + 1;
259  NEXTCHAR := PARAM[PARAMPTR];
260  END;
261
262  PROCEDURE SCANDRAPS;
263  VAR SKIPS, ALPHAS, NUMERICS: SET OF CHAR;
264  ID: IDENTIFIER;
265  C: CHAR;
266  N, IDPTR: INTEGER;
267  BEGIN
268  SKIPS := [' ', ' ', ' '];  263
269  ALPHAS := [A] TO [Z];  263 265 263 263 265
270  NUMERICS := [0] TO [9];  263 265 263 263 265
271  C := PARAM[1];  266 263 159
272  WHILE IDPTR > LENGTH THEN BEGIN
273  WHILE C IN SKIPS DO C := NEXTCHAR;
274  IF C IN ALPHAS THEN BEGIN
275  ID := C;
276  WHILE C IN (ALPHAS OR NUMERICS) DO BEGIN
277  IF IDPTR < IDLENGTH THEN BEGIN
278  IDPTR := IDPTR + 1;
279  END ELSE C := NEXTCHAR;
280  "IF ID[1]=" THEN OPTIONS := OPTIONS+[ID[2]]
281  ELSE OPTIONS := OPTIONS+[ID[1]];
282  ID := 'M';
283  IF ID='M' THEN OPTIONS := OPTIONS+[ID[1]]
284  THEN FOR N := 1 TO 7 DO TESTPASS := TESTPASS+1;
285  END
286  ELSE IF C IN NUMERICS THEN BEGIN
287  N := 0;
288  WHILE C IN NUMERICS DO BEGIN
289  N := N+ID-ORD('0');
290  END;
291  TESTPASS := TESTPASS+[N];
292  END
293  ELSE IF C IN CHAR THEN BEGIN
294  END "WHILE";
295  END;
296
297  PROCEDURE CHECK10;
298  VAR C: CHAR;
299  BEGIN
300 "COMPLETE SOURCE TEXT INPUT/OUTPUT:
301 IF NOT FAST_EOM THEN REPEAT READ(C) UNTIL C=EN;
302 WRITE(EN);
303 END;
304
305 PROCEDURE INITIALIZE;
306 BEGIN
307 WRITECHAR (NL);
308 TESTPASS:=#; NOSOURCE:=FALSE;
309 WITH LIST(.1) DO 157
310 BEGIN TAG:= BOOLTYPE; BOOL:= FALSE END;
311 WITH LIST(.2) DO 157
312 BEGIN TAG:= PTRTYPE; PTR:=Nil END;
313 WITH LIST(.3) DO 157
314 BEGIN TAG:= INTTYPE; INT:= 0 END;
315 SCAMPERS;
316 INITWRITE;
317 WRITEEXIT;
318 'HEPASCAL COMPILER (MICROENGINE P-Code) (:10;)(:10;):
319 END;
320
321 PROCEDURE CALLPASS(PASSNO: INTEGER; ID: IDENTIFIER);
322 VAR LINE: INTEGER; RESULT: PROGRESS;
323 BEGIN
324 LIST(.1).BOOL:= FALSE;
325 RUNID, LIST, LINE, RESULT);
326 IF RESULT <> TERMINATED THEN
327 BEGIN
328 REPORT:= OUT;
329 WRITERESULT(ID, LINE, RESULT);
330 END ELSE
331 BEGIN
332 OR:= LIST(.1).BOOL;
333 CORRELATION:= LIST(.3).INT;
334 IF NOT OR THEN 151
335 ERROR('COMPILATION ERRORS(:10;)(:10;):
336 IF OR THEN BEGIN
337 WRITEID(ID); WRITEEXIT(OK(:10;)(:10;));
338 END;
339 END;
340 END;
341 END;
342
343 PROCEDURE RUN_RUNHON (PASS: INTEGER);
344 VAR
345 L:INTEGER;
346 B: PROGRESS;
347 BEGIN
348 LIST[10].TAG := INTTYPE;
349 LIST[10].INT := PASS;
350 IF (PASS = 5) OR (PASS = 6)
351 THEN RUN ('MEMHEN: ', LIST, L, B);
352 CASE PASS OF
353 1: WRITEEXIT ('NO MEMHEN 1(:10;)(:10;):');
354 2: WRITEEXIT ('NO MEMHEN 2(:10;)(:10;):');
355 3: WRITEEXIT ('NO MEMHEN 3(:10;)(:10;):');
356 4: WRITEEXIT ('NO MEMHEN 4(:10;)(:10;):');
357 5: WRITEEXIT ('NO MEMHEN 5 (DONE(:10;)(:10;):');
358 END.
6: WRITEEXIT ('MEMBER 6  DORM(10:)(0:1)');
7: WRITEEXIT ('NO MEMBER 7(10:)(0:1)');
END;

REINITIALIZE;
IF NOXREF THEN CALLPASS(1,'MEMPASS1');
ELSE CALLPASS(1,'MEMPASS1');
PAST_PERM = LIST(5).ROOT;
IF (OK AND (1 IN TESTPASS)) THEN RUN_MEM(1);
IF (OK AND (2 IN TESTPASS)) THEN RUN_MEM(2);
IF (OK AND (3 IN TESTPASS)) THEN RUN_MEM(3);
IF (OK AND (4 IN TESTPASS)) THEN RUN_MEM(4);
IF (OK AND (5 IN TESTPASS)) THEN RUN_MEM(5);
IF (OK AND (6 IN TESTPASS)) THEN RUN_MEM(6);
IF (OK AND (7 IN TESTPASS)) THEN RUN_MEM(7);
END.
<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAILURE</td>
<td>43</td>
</tr>
<tr>
<td>FALSE</td>
<td>753</td>
</tr>
<tr>
<td>FF</td>
<td>13</td>
</tr>
<tr>
<td>FILER</td>
<td>28</td>
</tr>
<tr>
<td>FILERATTR</td>
<td>20</td>
</tr>
<tr>
<td>FILERIND</td>
<td>26</td>
</tr>
<tr>
<td>FOUND</td>
<td>93</td>
</tr>
<tr>
<td>FULLWORD</td>
<td>10</td>
</tr>
<tr>
<td>GRT</td>
<td>95</td>
</tr>
<tr>
<td>HEADER</td>
<td>102</td>
</tr>
<tr>
<td>HEAPLIMIT</td>
<td>88, 226</td>
</tr>
<tr>
<td>IDENTIFIER</td>
<td>22</td>
</tr>
<tr>
<td>IDENTIFY</td>
<td>102</td>
</tr>
<tr>
<td>IDLENTH</td>
<td>21</td>
</tr>
<tr>
<td>IDPTR</td>
<td>266</td>
</tr>
<tr>
<td>INTYPE</td>
<td>55</td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>305</td>
</tr>
<tr>
<td>INITWRITE</td>
<td>164</td>
</tr>
<tr>
<td>INP</td>
<td>84</td>
</tr>
<tr>
<td>INPUT</td>
<td>38</td>
</tr>
<tr>
<td>INTASK</td>
<td>52</td>
</tr>
<tr>
<td>INTEGER</td>
<td>10</td>
</tr>
<tr>
<td>INTERFACE</td>
<td>65</td>
</tr>
<tr>
<td>INTERVENTION</td>
<td>43</td>
</tr>
<tr>
<td>IOAM</td>
<td>40</td>
</tr>
<tr>
<td>IODEVICE</td>
<td>35</td>
</tr>
<tr>
<td>IOPERATION</td>
<td>38</td>
</tr>
<tr>
<td>IOPARAM</td>
<td>86</td>
</tr>
<tr>
<td>IORESULT</td>
<td>42</td>
</tr>
<tr>
<td>JOBNAME</td>
<td>52</td>
</tr>
<tr>
<td>K_</td>
<td>29</td>
</tr>
<tr>
<td>LABELS</td>
<td>386</td>
</tr>
<tr>
<td>LABELS</td>
<td>353</td>
</tr>
<tr>
<td>LENGTH</td>
<td>38</td>
</tr>
<tr>
<td>LINE</td>
<td>19</td>
</tr>
<tr>
<td>LINELENGTH</td>
<td>18</td>
</tr>
<tr>
<td>LIST</td>
<td>157</td>
</tr>
<tr>
<td>LISTOPTD</td>
<td>61</td>
</tr>
<tr>
<td>PROTECTED</td>
<td>31*</td>
</tr>
<tr>
<td>PTR</td>
<td>78 312=</td>
</tr>
<tr>
<td>FEETPR</td>
<td>55 78* 312</td>
</tr>
<tr>
<td>FUT</td>
<td>36*</td>
</tr>
<tr>
<td>-R-</td>
<td></td>
</tr>
<tr>
<td>RANGEERROR</td>
<td>87 228</td>
</tr>
<tr>
<td>READ</td>
<td>90* 301</td>
</tr>
<tr>
<td>RELEASE</td>
<td>100*</td>
</tr>
<tr>
<td>REN</td>
<td>191* 193= 197 198* 199</td>
</tr>
<tr>
<td>REPORT</td>
<td>159* 167= 172 328=</td>
</tr>
<tr>
<td>RESETPOINT</td>
<td>67*</td>
</tr>
<tr>
<td>RESULT</td>
<td>122* 218= 220 235= 243 246 322= 325 326 329</td>
</tr>
<tr>
<td>REVIND</td>
<td>90</td>
</tr>
<tr>
<td>RUN</td>
<td>121* 325 353</td>
</tr>
<tr>
<td>RUN_MEN</td>
<td>344* 372 376 380 384 388 392 396</td>
</tr>
<tr>
<td>-S-</td>
<td></td>
</tr>
<tr>
<td>SCANFARMS</td>
<td>262* 315</td>
</tr>
<tr>
<td>SCRAMBLE</td>
<td>26</td>
</tr>
<tr>
<td>SEGMENT</td>
<td>26</td>
</tr>
<tr>
<td>SHORTINTEGER</td>
<td>11</td>
</tr>
<tr>
<td>SKIPS</td>
<td>263* 268= 273</td>
</tr>
<tr>
<td>SOURC</td>
<td>151*</td>
</tr>
<tr>
<td>STACKLIMIT</td>
<td>88 227</td>
</tr>
<tr>
<td>STARTMEDIUM</td>
<td>68</td>
</tr>
<tr>
<td>STATUS</td>
<td>48*</td>
</tr>
<tr>
<td>SUMMARYOPTIO</td>
<td>61</td>
</tr>
<tr>
<td>-T-</td>
<td></td>
</tr>
<tr>
<td>TABLE</td>
<td>68*</td>
</tr>
<tr>
<td>TAG</td>
<td>73* 310= 312= 314= 349=</td>
</tr>
<tr>
<td>TABLEDEVICE</td>
<td>36</td>
</tr>
<tr>
<td>TACHETING</td>
<td>52*</td>
</tr>
<tr>
<td>TERMINATED</td>
<td>87 221 246 326</td>
</tr>
<tr>
<td>TEOF</td>
<td>61</td>
</tr>
<tr>
<td>TEOFPO</td>
<td>160* 204= 204= 201= 201= 300= 311 375 379 383 387 391 395</td>
</tr>
<tr>
<td>TEXT</td>
<td>177* 181 185 299* 252</td>
</tr>
<tr>
<td>THETH</td>
<td>88 229</td>
</tr>
<tr>
<td>TOP</td>
<td>99* 100*</td>
</tr>
<tr>
<td>TRANSMISSION</td>
<td>43</td>
</tr>
<tr>
<td>TRUE</td>
<td>282</td>
</tr>
<tr>
<td>TYPEDevice</td>
<td>36</td>
</tr>
<tr>
<td>-U-</td>
<td></td>
</tr>
<tr>
<td>USPACE</td>
<td>40</td>
</tr>
<tr>
<td>-V-</td>
<td></td>
</tr>
<tr>
<td>VARIANTERROR</td>
<td>87 225</td>
</tr>
<tr>
<td>-W-</td>
<td></td>
</tr>
<tr>
<td>WHERE</td>
<td>153*</td>
</tr>
<tr>
<td>WRITE</td>
<td>91* 173 302</td>
</tr>
<tr>
<td>WRITECHAR</td>
<td>170* 184 201 202 203 205 214 238 242 245 307</td>
</tr>
<tr>
<td>WRITEDEF</td>
<td>30</td>
</tr>
<tr>
<td>WRITEIND</td>
<td>208* 239 248 318</td>
</tr>
<tr>
<td>WRITEINT</td>
<td>105* 241</td>
</tr>
<tr>
<td>WRITEDQUERY</td>
<td>218* 329</td>
</tr>
<tr>
<td>WRITETEXT</td>
<td>177* 240 252 317 338 355 356 357 358 359 360 361</td>
</tr>
</tbody>
</table>
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 ---
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;

SUB2=81; INDEX2=82; REAL2=83; STRING2=84;
LCONST2=85; MESSAGE2=86; NEW_LINE2=87; FWD_DEF2=88;
CHK_TYPE2=89; PROCF_DEF2=90; UNDEF2=91; PEND2=92;
CASE_JUMP2=93;
"MANAGER:"
"MGR"
MANAGER2=94; FROM2=95; REFERENCES2=96;
NIL2=97;
"MGR"
VTAG_DEF2=98; VPART_END2=99; VVARNT_END2=100; "VARNT"
VVARIANT2=101; "VARNT" D U P T O S 2 = 1 0 2 ; "MICROENGINE"
"DUPTOS2 DUPLICATES TOP-OF-STACK WORD"

"OTHER CONSTANTS"

SPLIT_SET_LENGTH = "$IF .SRC_16" 8; "$END" "WORDS/SPLIT SET"
"IF SRC_32" 4; "$END"
NOUN_MAX=999;

FUNCTION1: FUNCTION_;
GE1: BINARY(GE2);
GT1: BINARY(GT2);
INCLUDE1: BINARY(INCLUDE2);
INIT_NAME1: BEGIN INIT_NAME; PUTO(DUPTOS2) END; "MICROENGINE"
INITS_DEF1: INITS_DEF;
INITS_END1: POP_LEVEL;
INIT1: CALL(INIT2);
INTEGER1: INDEX(XINTEGER);
INF_ID1: INF_ID;

--- PASS 3 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;

CASE JUMP1=93;
MANAGER1=94; FROM1=95; REFERS1=96; "MGR*"
NIL1=97; "MGR*"
VTAG_DEF1=98; VPART_END1=99; VVARNT_END1=100; VVARIANT1=101;
DUPTOS1 = 102; "MICROENGINE"

"OUTPUT OPERATORS"

EOM2=1; BODY2=2; BODY_END2=3; ADDRESS2=4;
RESULT2=5; STORE2=6; CALL_PROC2=7; CONSPARM2=8;

VCOMP2=49; RCOMP2=50; SUB2=51; LCONST2=52;
MESSAGE2=53; NEW_LINE2=54; CHK_TYPE2=55; SAVEPARM2=56;
CALL_GEN2=57; NOT2=58; UNDEF2=59; RANGE2=60;
REFER2=61; "MGR*"
VVARNT2=62; "VARNT*" DUPTOS2=63; "MICROENGINE"

"STANDARD SPELLING/NOUN INDICES"

XUNDEF=0; XFALSE=1; XTRUE=2; XINTEGER=3;
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;

--- PASS 4 CODE MODIFICATIONS ---
"*************** 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 RINS IN EACH INTERFACE"
      ARRAY[1..MAXINTFAC] OF INTEGER;
  END;

*************** MICROENGINE ******************

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; SAVEPARAM1=56;
CALL_GEN1=57; NOT1=58; UNDEF1=59; RANGE1=60;
REFER1=61; "MGR*"
VARIANT1=62; "VARNT*" DUTFOS1=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; "MICROENGINE"
DUTFOS2=49;

"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"

INTERFACEPTR: IFPTR;  "MICROENGINE"
INTERFACE: IFINFO;  "MICROENGINE"

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

PROCEDURE NEXT_PASS (VAR LINK: PASSPTR);
BEGIN
    LINK@.INTERFACE := INTERFACEPTR;  "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(INTERFACEPTR); "GET SPACE FOR INTERFACE INFO RECORD"  
MARK(INTER_PASS_PTR@.RESETPOINT); "GET @ NEW HEAPTOP"  
INTERFACEPTR@.INTERFACES := 0; "NO INTERFACES DEFINED YET"

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

--- PASS 5 CODE MODIFICATIONS ---
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 INF_LEN:INTEGER;
BEGIN
READ_IFL(INF_LEN);

"MICROENGINE"

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

PUT0(CALLPROC2); PUT1(POP2,INF_LEN);
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;

--- PASS 5 CODE MODIFICATIONS ---
CASE_LIST1: CASE_LIST;
CHK_TYPE1: CHK_TYPE;
CONSTPARM1: CONSTPARM(FALSE);
DEF_LABEL1: DEF_LABEL;
DIV1: DIV_MOD(DIV2);
DUP2OS1: PUTO(DUP2OS2); "MICROENGINE"
EMPTY_SET1: EMPTY_SET;
EOM1: EOM;
EQ1: EQUALITY(EQUAL);
FALSEJUMP1: FALSE_JUMP;

--- PASS 5 CODE MODIFICATIONS ---
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.