Computer Science Department
114 Lind Hall
Institute of Technology
University of Minnesota
Minneapolis, Minnesota 55455

A Primer on the SMILE Microprogram Load and Test System

by
H. K. Berg and N. Samari Kermani

Technical Report 78-11
July 1978

Cover design courtesy of Ruth and Jay Leavitt

A Primer on the SMILE* Microprogram Load and Test System

by

Helmut K. Berg and Nemattolah Samari Kermani
Department of Computer Science
University of Minnesota

Abstract

This report is a practical introduction to the use of SMILE, a system for microprogram load and examination. It is meant to familiarize new users with SMILE as one of the microprogram development aids in the microprogramming laboratory, and as a reference for advanced users. The organization of the SMILE system, input to be prepared by the user, and the interface with other microprogram development aids and the UNIX operating system are presented in the form of a tutorial. The report covers the basics needed for the use of SMILE, such as typing commands, operating the system from the processor console, and interpreting system responses. The concept and use of the SMILE system is demonstrated by an example.

---

*SMILE, a System for MIcroprogram Load and Examination, was developed at the Technical University Berlin, Institut für Softwaretechnik und Theoretische Informatik, Fachgebiet Betriebssysteme.

1. Introduction

SMILE is a system for loading and examining PDP-11/40E user microprograms. It was developed at the Technical University Berlin [1]. SMILE runs on the bare PDP-11/40E hardware and is bootstrapped from a magnetic tape. The original SMILE version is bootstrapped from a DECTAPE, TC11 [2], which is not available in our PDP-11/40E system configuration. The SMILE version we refer to in this report is a modification of the original version which is bootstrapped from the available magnetic tape, TM11[2]. The SMILE tape is generated in a post-processing step following the assembly of a user microprogram with the MICRO/40 assembler [3].

The PDP-11/40E was developed at Carnegie-Mellon University [3,4]. It is a standard PDP-11/40 computer that has been extended by the following hardware features:

- 1K 80-bit words of random access (RAM) control store for storing user microprograms.
- 32 80-bit words of read-only (PROM) control store for bootstrapping microprograms.
- a 16-word stack for temporary data storage.
- a shift and mask unit and a carry control unit which extend the data manipulation capabilities of the basic PDP-11/40 processor.

The 3-Rivers Computer Corporation offers these hardware accessories as a writable control store option (WCS 11/40) for the PDP-11/40. All normal PDP-11/40 features are unaffected by the WCS 11/40. The design of this extension allows user microprograms access to all functional hardware units and data paths in the basic PDP-11/40 processor and in the WCS 11/40. Introductions to the microprogramming of the PDP-11/40E are given in [3,5].

The SMILE system is the control store loading facility in our PDP-11/40E microprogramming support system. Additionally, a microassembler [3,6] and a microsimulator [3,7] are available. These facilities constitute a typical microprogramming support system. More sophisticated support systems may also include test set generators, external hardware accessories for microporgram instrumentation, microprogram verification system, etc. As microinstructions affect all hardware resources in a computer and hence, faulty microprograms result in an erroneously operating machine, it is important that a microprogramming support system provides sufficient tools for microcode validation.

The basic approaches to program validation are formal correctness proofs and testing. Proofs of formal correctness, which attempt to show the absence of errors, are not supported by our PDP-11/40E microprogramming system. Microcode validation by testing is constrained by the fact that testing, in general, can only show the presence of errors, but not their absence. Microprogramming errors may occur at the microoperation level, the microinstruction level, and the microprogram level. The SMILE system provides testing facilities at the microprogram level.

In general, we can distinguish between static program tests, via program analysis, and dynamic program test, via program execution. Syntax checks as performed by the MICRO/40 assembler may be considered as a type of static testing. Dynamic microcode test methods can be classified into soft (off-line) testing and hard (on-line) testing. The PDP-11/40E microcode simulator [7] is an off-line testing system that allows the examination of simulated microinstruction executions. The ability to detect dynamic, hardware-dependent timing errors with an off-line tester depends on the homogeneity of the mapping of the machine hardware into the tester software. Therefore, on-line testing techniques are generally better suited for discovering dynamic errors in microprograms. The SMILE system allows on-line testing at the microprogram level. The microprogrammer can specify a PDP-11 machine language test program whose execution calls upon the execution of microprograms from the writable control store. The effect of microprogram execution in the physical machine can then be observed by investigating the appropriate processor registers and main memory locations. In this respect, the SMILE on-line test facilities at the microprogram level complement off-line simulator tests at the microinstruction level. However, these testing tools do not lend themselves to the location of dynamic errors at the microoperation level. Although the PDP-11/40E microinstruction format facilitates the detection of microoperation timing conflicts in the MICRO/40 assembly process [6], the provision of an on-line test system at the microoperation level is considered necessary. Therefore, the use of a logic state analyzer for this purpose has been investigated [8].

This report is a practical introduction to the use of the SMILE system. We first describe (section 2) the organization of the system. This description deviates from the original SMILE documentation [1] with respect to the modifications which became necessary to install the system in our microprogramming laboratory. It includes guidelines for the operation of the SMILE system.

The discussion in sections 2 and 3 is based on a simple example. With this example, the use of SMILE in microprogram development is demonstrated by a complete terminal session presented in section 3.

2. SMILE Organization

In this section, we describe the SMILE tape, as generated after microprogram assembly, and its constituent files. The discussion is based on the example microprogram [1], called fastc.mic, shown in Fig.1. This microprogram implements two machine language subroutines that handle the environment switch for subroutines calls in the "C" programming language of UNIX. It saves and restores registers that are used for parameter passing in subroutines calls. Further details of fastc.mic will be introduced as needed.

```
require defs.mic

begin noop
.=2001; d_210; b_d
d_rir-b !compare instruction
skipzero
d_211; b_d

set

  start          !check for other instr
    d_rir-b
    skipzero
    noop

    set

        goto 150

      start      !211 instr
        d_r5     !r1<-r5
        r1_d
        d,ba_r1-2; r1_d !pop r4
        dati; clkoff
        r4_unibus
        ba,d_r1-2; r1_d !pop r3
        dati; clkoff
        r3_unibus
        ba,d_r1-2; r1_d !pop r2
        dati; clkoff
        r2_unibus
        ba,d_r5 !sp<-r5
        r6_d; dati; clkoff
        r5_unibus        !r5<-(sp)+
        d_r6+2; r6_d
        ba_r6; dati      !rts pc
        d_r6+2; r6_d; clkoff
        r7_unibus; but 16
        goto 16
      end

    tes
  end

start          !210 instr
      d,ba_r6-2; r6_d !push r5
      d_r5; dato; clkoff
      d_r7; p3                !r5<-r7
      r5_d
      r0_d      !r0<-r5
      d_r6      !r5<-r6
      r5_d
      d,ba_r6-2; r6_d !push r4
      d_r4; dato; clkoff
      d,ba_r6-2; r6_d !push r3
      d_r3; dato; clkoff
      d,ba_r6-2; r6_d !push r2
      d_r2; dato; clkoff
      d_r6-2; r6_d     !r6<-r6-2
      d_r0      !r7<-r0
      r7_d; but 16
      goto 16
  end

tes

finis
```

Figure 1: fastc.mic

---

1) This microprogram was developed by K. Bullis, J. Bjoin, and T. Lunzer as a course project for (H.K. Berg) CSci 5299, Microprogramming, Winter Quarter 1978.

2.1 Interface with MICRO/40

User microcode for the PDP-11/40E is written in the MICRO/40 assembly language. For a detailed description of MICRO/40, the reader is referred to [6]. Microcode source files are generated using the UNIX text editor [9]. MICRO/40 microprogram source files must have names of the form

<name>.mic,

where <name> is any legal name, e.g., fastc. To assemble a microprogram source file, the UNIX command [10]

mic [opt] <name>.mic

is given. For use with the SMILE system, the options for assembler calls, opt, are not significant. The default assembler call,

mic <name>.mic ,

which is used with SMILE, generates three files, namely <name>.lst, <name>.bin, and <name>.tab(<name>.tab is only used by the microsimulator [7]).

<name>.lst

This file is a listing of the microprogram object code in the 80-bit PDP-11/40E microinstruction format, followed by a list of mnemonic labels and their associated control store addresses.

<name>.bin

This file contains the binary version of the assembled microcode. It is generated from an internal file, <name>.s, in which a pseudo-readable form of the object microcode is stored in UNIX assembler format. The binary version of the assembled microcode, <name>.bin, is generated using the UNIX assembler as a post-processor of MICRO/40.

When the assembly process is completed, the SMILE system can be invoked to load the object microcode into the writable control store, and to test the microcode at the microprogram level. As there is no protection mechanism built into the micro-architecture of the PDP-11/40E, the on-line execution of partially validated user microprograms has the potential to harm the system software. Therefore, the SMILE system is designed to run on the bare machine hardware. To this end, SMILE is bootstrapped from a magnetic tape. This tape is generated under the UNIX operating system. After the generation of the SMILE tape, the operating system is shut off (demount the system disk) to run SMILE. Following the execution of the SMILE tape, the user microcode is stored in the writable control store and the machine is available for normal operation. For instance, machine instructions may be executed which

call upon the execution of user microprograms, or an operating system (e.g., UNIX) may be bootstrapped.

Following microcode assembly under UNIX, the SMILE tape is generated by the UNIX command

smile <name>.bin [<test program name>].

<test program name> is a file which contains the binary representation of a machine language program to be used for testing the user microcode at the microprogram level. The specification of a test program is optional. If the SMILE system is only to be used for loading microcode into the writable control, the SMILE tape can be generated by the UNIX command

smile <name>.bin

which does not refer to a test program. The SMILE tape consists of the following four files:

- the system loader,
- the RAM loader,
- the object code of the user microprogram (<name>.bin),
- the object code of the machine language test program (<test program name>).

2.2 Organization of the SMILE Tape

The UNIX command 'smile' is a program that generates the SMILE tape. This program is written in the "C" programming language of UNIX [11]. It checks the status of the magnetic tape drive, TM11. If the tape drive is not ready for writing, the error message,

cannot open /DEV/TM11/ ,

is printed and the program halts. Otherwise, the subroutines TWRITE is called to write each of the constituent SMILE files on the magnetic tape. As the user microprogram and the test program are variable in size, TWRITE attaches the file headers (specifying the size) to these two files. Additionally, the program symbol table is included in the test program file. For each file that is written on the tape, the subroutine TWRITE prints the following message,

<file name>: <# bytes>/<max# bytes>bytes <# blocks>/<max# blocks>blocks

where <# bytes> and <# blocks> are the numbers of bytes and blocks, respectively, in the file to be written, and <max# bytes> and <max# blocks> specify the allowed maximum size of the file <file name>. The block size on the TM11 7-track tapes used in our microprogramming laboratory is 512 bytes, which corresponds to 256

words. If the subroutine TWRITE is for some reason unable to properly write on the magnetic tape, the program halts after printing one of the error messages listed in Table 1.

cannot open /DEV/TM11/	If the device is not ready for writing.
cannot open <file name>	If one of the SMILE files cannot be read from the UNIX file system.
too many bytes	If <# bytes > exceeds <max# bytes>.
too many blocks	If <# blocks> exceeds <max# blocks>.
read error on <file name>	If a read error occurs when reading from the UNIX file system.
write error on tape	If a write error occurs when writing onto the tape.
more blks written than read	If the number of blocks written on the tape is greater than the number of blocks read from the UNIX file system.

Table 1: Smile Command Error Messages.

After the program 'smile' has been executed, the generated magnetic tape has the format shown in Fig. 2.

Block 0: As the bootstrap program in our system skips block 0, the first block on the tape contains no information.

Block 1: This block contains the SMILE system loader. The system loader's size is 146 bytes and hence, bytes 222_8 to 777_8 of this block are not used.

Block 2: This block contains the RAM loader. Its size is 226 bytes, leaving bytes 342_8 to 777_8 of this block empty.

Blocks 3 to i: The object code of the user microprogram, <name>.bin, to be loaded into the writable control store is stored in these blocks. As the size of this file is variable, a file header is attached to it. This file header has the format of the UNIX object file header [9]. The following two of the eight header words are of interest. Word 2 specifies the size of the file in number of bytes. The microassembler stores the size of the microcode file in blocks into word 7 which is unused in UNIX. For the microcode file, <max# blocks> = 21 and hence, <max# bytes> = 10752. The file header is stored in a separate block. The remaining 20 blocks correspond to 5K of 16-bit words

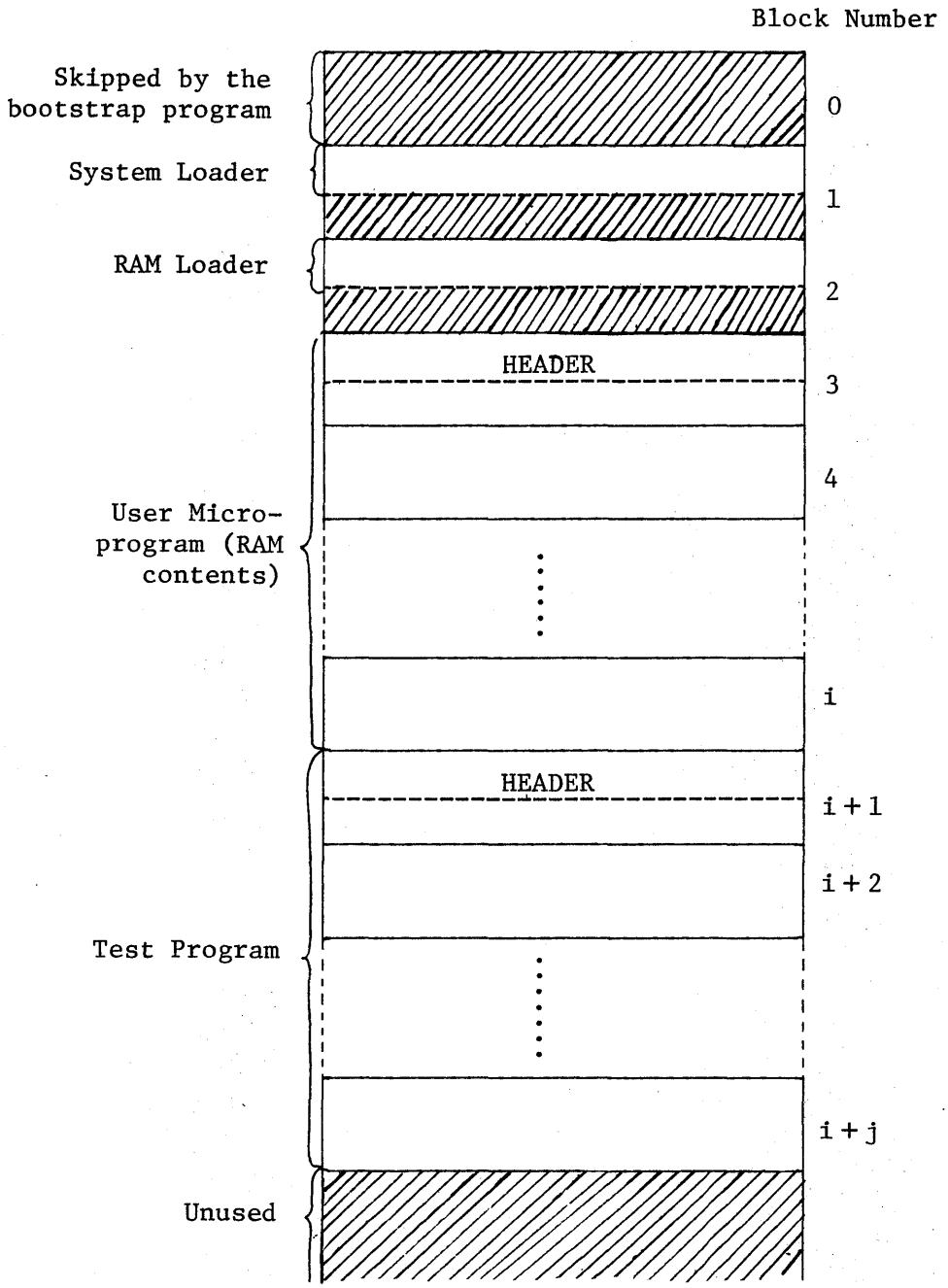

Figure 2: SMILE Tape layout (the shaded areas do not contain information)

for the user microcode (cf. Fig.4 in the following subsection).

Blocks i+1 to i+j: These blocks contain the object code of the machine language test program, <test program name>, and the associated program symbol table. As this file is of variable size, a file header is attached. Its size is specified in the same way as for the microcode file; Additionally, the size of the program symbol table in bytes is stored in word 5 of the file header. For the test program, we have <max# blocks> = 89 and <max# bytes> = 19952. In this file, the file header is not stored in a separate block and hence, the test program can be up to 22.5K 16-bit words long (cf. Fig. 4 in the following subsection).

The SMILE tape for our example microprogram as generated by the command

smile fastc.bin a.out

is shown in Fig. 3. The UNIX file a.out contains the binary object code of the test program. This file exists as long as no other program is assembled or compiled, since object programs generated under UNIX are always stored in this file. If it is desirable to preserve the object code of a test program, the file a.out can be saved into a user file, <name of test program>, by the UNIX command

mv a.out <name of test program>.

Block 1
System
Loader

Byte Number (Octal) ↓

```
0001000 000000 012706 133000 012705 172522 012700 000001 012701
0001020 000400 004767 000126 000000 005200 012701 133000 005061
0001040 000020 004767 000106 016102 000014 060002 005200 016101
0001060 000020 004767 000044 000000 012701 000760 005061 000014
0001100 004767 000050 016102 000014 060002 005200 062701 001000
0001120 004767 000006 000000 000137 000400 020002 002401 000207
0001140 004767 000010 005200 062701 001000 000767 010165 000004
0001160 012765 177000 000002 012715 060003 004767 000002 000207
0001200 032715 100200 001775 100401 000207 013700 172520 000000
0001220 000207 000000 000000 000000 000000 000000 000000 000000
0001240 000000
```

As the system loader is of fixed size (146 bytes), no file header is attached

Block 2
RAM
Loader

Byte Number (Octal) ↓

```
0002000 013746 133772 004767 000060 010002 010003 013716 133774
0002020 062716 000010 004767 000040 005726 005005 062205 005505
0002040 020200 103774 020537 133776 001422 012700 000001 004767
0002060 000246 010537 133776 000744 016600 000002 162700 020000
0002100 010001 006201 006201 060100 062700 134000 000207 032737
0002120 000001 177570 001005 012700 170000 004767 000172 000767
0002140 013746 177776 052737 000200 177776 005005 013701 133772
0002160 012704 000652 011300 005002 000007 010702 000007 020023
0002200 001031 060005 005505 062401 020427 000662 101763 020137
0002220 133774 003756 012637 177776 020537 133776 001421 012700
0002240 000002 012746 000400 000167 000056 000002 000002 000002
0002260 037772 140010 012700 000010 004767 000034 005743 000732
0002300 012746 000700 005000 012746 001000 000167 000012 000000
0002320 000000 000000 000000 000000 012737 000007 177566 000000
0002340 000207 000000 000000 000000 000000 000000 000000 000000
0002360 000000
```

(continued)

... RAM loader is of fixed size (226 bytes), no file header is attached

Block 3
Header
fastc.bin

```
                  Byte
                  Number
                  (octal)        #bytes                                                    #blocks
                    ↓               ↓                                                         ↓
File Header  →    0003000 000407 001662 000000 000000 000000 000000 000002 000001
                  0003020 134000 134670 000000 000000 000000 000000 000000 000000
                  0003040 000000
Control      →    0003760 000000 000000 000000 000000 000000 020000 020540 060615
Information                                                     ↑      ↑      ↑
                                                             ramfrst ramlast chcksum
```

Block 4
fastc.bin

```
                  Byte
                  Number
                  (Octal)
                    ↓
                  0004000 000000 000000 040000 002376 000000 100000 000000 141400
                  0004020 006371 000210 000033 003000 100410 002366 000000 100026
                  0004040 003057 146610 002342 000000 000000 000000 040000 000227
                  0004060 000000 000025 000000 100400 002364 000000 000033 003000
                  0004100 100410 002370 000000 005000 000000 040000 002367 000000
                  0004120 100000 000000 141400 006375 000211 005000 000000 040000
                  0004140 002365 000000 000000 000000 040000 002373 000000 100021
                  0004160 000000 046000 002363 000000 100021 003057 146610 002362
                  0004200 000000 000000 000000 060020 002361 000000 040024 000000
                  0004220 046000 002360 000000 100021 003057 146610 002357 000000
                  0004240 000000 000000 060020 002356 000000 040023 000000 046000
                  0004260 002355 000000 100021 003057 146610 002354 000000 000000
                  0004300 000000 060020 002353 000000 040022 000000 046000 002352
                  0004320 000000 020025 000000 100600 002351 000000 100026 000000
                  0004340 066020 002350 000000 040025 000000 046000 002347 000000
                  0004360 100026 004457 146400 002346 000000 020026 000000 040220
                  0004400 002345 000000 100026 004457 166400 002344 000000 047027
                  0004420 000000 046000 002343 000000 000000 000000 040000 000361
                  0004440 000000 000025 000000 120520 002341 000000 000027 000000
                  0004460 140400 002340 000000 100025 000000 046000 002337 000000
                  0004500 100020 000000 046000 002336 000000 000026 000000 100400
                  0004520 002335 000000 100025 000000 046000 002334 000000 100026
                  0004540 003057 146610 002333 000000 000024 000000 120520 002332
                  0004560 000000 100026 003057 146610 002331 000000 000023 000000
                  0004600 120520 002330 000000 100026 003057 146610 002327 000000
                  0004620 000022 000000 120520 002326 000000 100026 003057 146410
                  0004640 002325 000000 000020 000000 100400 002324 000000 107027
                  0004660 000000 046000 002323 000000 000000 000000 040000 000361
                  0004700 000000
Control      →    0004760 000000 000000 000000 000000 000000 020000 020540 060615
Information                                                     ↑      ↑      ↑
                                                             ramfrst ramlast chcksum
```

Block 5
a.out

```
                  Byte                                 length
                  Number                               symbol
                  (Octal)        #bytes                table           #blocks
                    ↓               ↓                    ↓                ↓
File Header  →    0005000 000407 000054 000000 000000 000000 000000 000001 000000
                  0005020 012700 000000 012701 000001 012702 000002 012703 000003
                  0005040 012704 000004 012705 000005 012706 133000 000210 000000
                  0005060 005000 005001 005002 005004 000211 000000
```

This test program file does not not contain a symbol table

Figure 3: Smile Tape Example

2.3 System Loader

The system loader is the first file on the SMILE tape (block 1). It is brought into main memory by bootstrapping the SMILE tape. The main memory locations 0_8 to 376_8 are reserved for the SMILE system loader. The bootstrap program places it into locations 0_8 to 110_8 and then transfers control to the first instruction of the system loader and halts. The system loader is invoked by pressing the CONTINUE switch at the processor console. It loads the RAM loader, the microcode, and the test program into main memory, thus, creating the main memory map as depicted in Fig. 4. To load these files requires that the system loader is reinvoked for each file by pressing the CONTINUE switch. The implementation of the system loader is based on the fixed SMILE tape format as discussed in the preceeding subsection. It is written in PDP-11/40 assembly language and makes no use of the memory management [12]. Hence, the main memory map shown in Fig. 4. is a description of the lower 28K words of physical memory.

The SMILE system loader first reads the RAM loader (block 2) from the magnetic tape into main memory locations 400_8 to 741_8. In the main memroy map, locations 400_8 to 756_8 are reserved for the RAM loader.

Next, the first block of the microcode file, which contains only the file header and some control information (cf. Fig. 3), is read from the magnetic tape. From this block, the size of the object microcode is calculated. Then, the rest of the microcode file is read into main memory. Location 134000_8 to 157776_8 are reserved for this purpose. The maximum length of the object microcode corresponds to the maximum writable control store capacity of 1K 80-bit words. The 10K bytes are mapped into the highest 5K of 16-bit main memory words (23K to 28K) as follows. The increment of control store addresses from one 80-bit RAM-word to the next is 010_8[5], whereas the increment of main memory addresses from one 80-bit RAM-word to the next is 10 (decimal) bytes. Hence, we have

$$\text{memoffset} = \text{ramoffset} \cdot 5/4 \cdot$$

with the control store address of the first 80-bit ram0 $=2000_8$, we have

$$\text{ramoffset} = \text{ramadr} - \text{ram0}.$$

If we denote the main memory address of ram0 by ram_c, and with ram_c = 134000_8, the main memory address corresponding to a given control store address is calculated as follows

$$\begin{aligned} \text{memadr} &= \text{ram\_c} + \text{memoffset} \\ &= \text{ram\_c} + (\text{ramadr}-\text{ram0}) \cdot 5/4 \cdot \end{aligned}$$

Using this relationship, the user may perform smaller modifications of the

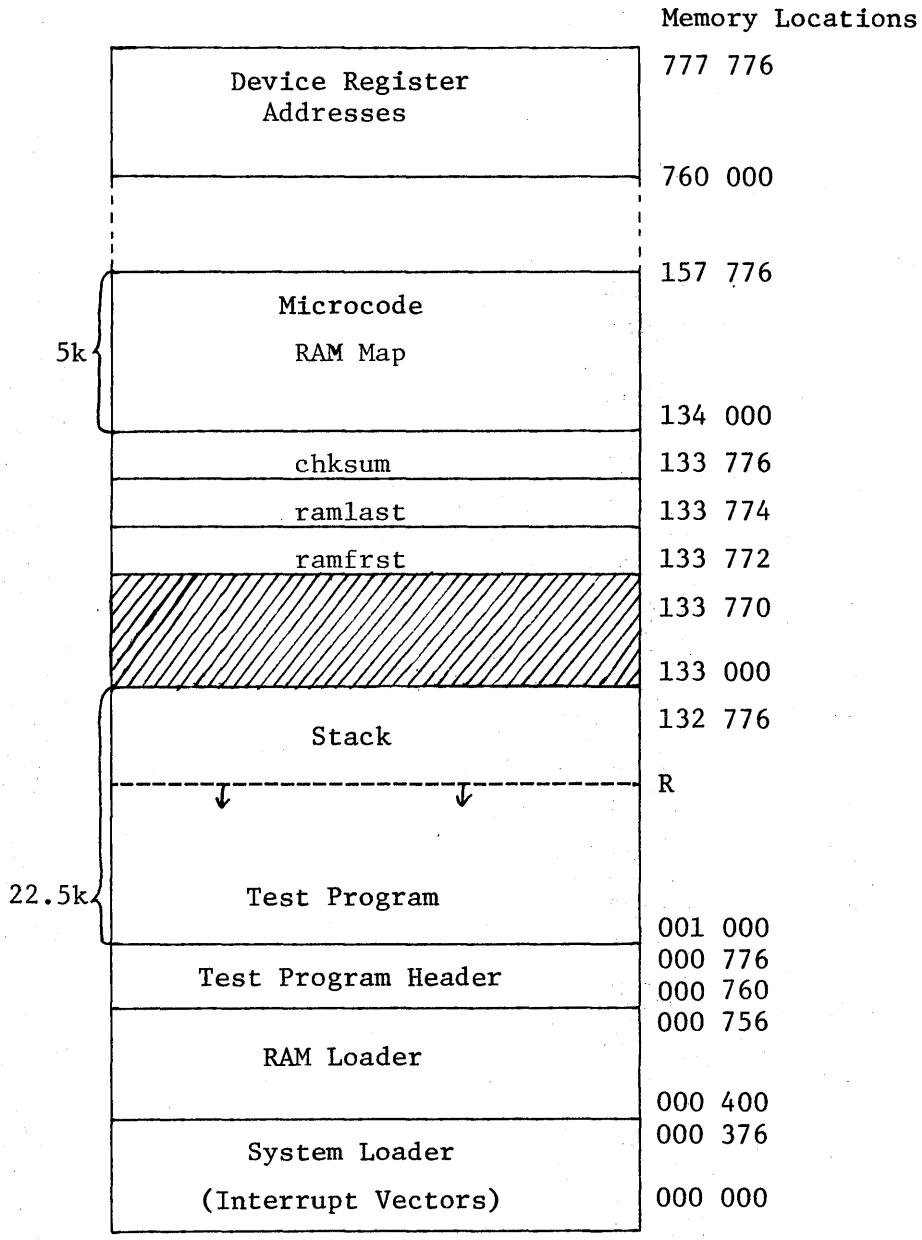

Figure 4: Main Memory Layout

object microcode from the processor console, without reinvoking the micro-assembler and generating a new SMILE tape. The bit assignment of the micro-operation fields in the 80-bit RAM word are discussed in [3,5].

The last eight words of the first and last block in the microcode file on the tape contain control information (cf. Fig. 3). The first five of these control words are empty. Words 6 and 7 specify the first (ramfrst) and last (ramlast) control store addresses to be loaded with the microcode in the file. The addresses ramfrst and ramlast are stored in the main memory locations 133772_8 and 133774_8 respectively. Word 8 of the control information contains a check sum (chksum) which is stored in location 133776_8 for later use by the RAM loader.

Finally, the system loader transfers the object code of the test program from the magnetic tape into main memory. In this case, the file header and the object code contained in the first block of the file (cf. Fig. 3) are stored in main memory starting at location $000\ 760_8$. That is, the file header is stored in locations $000\ 760_8$ to $000\ 776_8$ and the object code of the test program starts at location $001\ 000_8$. The size of the test program file is calculated from the information in the file header, and then, the rest of this file is read from the magnetic tape.

Locations 1000_8 to 132776_8 are reserved for the storage of the test program and its program symbol table. Test program file is stored, starting at main memory location 1000_8. However, the test program file must not completely fill the reserved 22.5K main memory words as main memory location 132776_8 to R are used for the stacks in the system loader and the RAM loader. This stack should also be used by the test program (if this program contains stack instructions). It is the user's responsibility to prevent overwritting of the test program by the stack. As a general rule, the reservation of ten main memory words for the stacks in the system loader and RAM loader is sufficient.

Main memory locations $760\ 000_8$ to $777\ 776_8$ as shown in Fig. 4 are reserved for PDP-11 device register addresses. These locations correspond to the 16-bit addresses $160\ 000_8$ to $177\ 776_8$ which are automatically relocated by setting the address bits 16 and 17 in the 18-bit UNIBUS address to 1, if the address bits 13, 14, and 15 are 1. With the memory management option [12], the main memory space may be extended from 28K as shown in Fig. 4 up to 124K words (locations $160\ 000_8$ to $757\ 776_8$).

If an error occurs when the system loader reads the SMILE tape, an error subroutine is called which stores the address of the currently written main

memory word into the processor register R[0]. Then, the program halts and the content of R[0] is displayed on the data display at the processor console. Thus, using the main memory map given in Fig. 4, it can be determined how much of the SMILE tape had been read at the time the error occured. When the system loader terminates after loading the entire SMILE tape, it is no longer needed. Hence, the test program can use its memory space for the storage of interrupt vectors [12](cf. subsection 2.5).

2.4 RAM Loader

After the entire SMILE tape has been transferred into main memory, the system loader transfers control to the first instruction of the RAM loader at location 400_8 and halts. The RAM loader is invoked by pressing the CONTINUE switch at the processor console. It loads the object code of the user microprogram from main memory into the writable control store locations specified by ramfrst (main memory location 133 772_8) and ramlast (main memory location 133 774_8). The implementation of the loader is based on the main memory map of the 80-bit RAM-words as discussed in the preceeding subsection. It is written in the PDP-11/40 assembly language.

The control store loading process is controlled by check sum tests and immediate rereads from the control store. These tests are preformed to ensure that the microcode loaded into the control store is equivalent to the microcode generated by the microassembler. To this end, the RAM loader first calculates a check sum of the main memory words to be transferred into the control store and compares it with the check sum that was produced by the microassembler (chksum in main memory location 133 776_8). If the two check sums agree, the microcode can be loaded into the control store. Otherwise, the RAM loader issues an error message and halts. The user may request continuation of the loading process by pressing the CONTINUE switch at the processor console. In this case, the checksum calculated by the RAM loader is written into location 133 776_8 and the check sum test is repeated.

This option has been built into the RAM loader to allow for immediate modifications of the user microprogram. The checksum of the original object microcode is automatically replaced by a checksum for the modified object microcode. Hence, it is possible to make smaller modifications in the object code of the user microprogram, without invoking MICRO/40 under UNIX for reassembling a modified source microprogram and generating a new SMILE tape. Such modification may even be made after the microprogram has been tested. In this case, control must manually (at the processor console) be transferred to the beginning of the Ram loader.

For this reason, the main memory locations of the RAM loader should not be changed by the test program.

During the control store loading process, the RAM loader immediately rereads each transferred 16-bit word from the control store and compares it with the appropriate main memory word. If the two 16-bit patterns disagree, the WRITE operation can be repeated by pressing the CONTINUE switch. Furthermore, after the entire RAM map has been written into the control store, the control store content is reread and a checksum is calculated. If this checksum disagrees with the check sum stored in location $133\ 776_8$, the checksum in location $133\ 776_8$ is checked by recalculating a new checksum for the microcode in main memory and a new control store loading process can be initiated by pressing the CONTINUE switch.

The RAM loader informs the user of the status of the control store loading process by generating the encoded error messages listed in Table 2. To alert the user, the console 'peeps' and the message code is displayed on the data display. At this time, the program halts at location $000\ 740_8$ which is displayed on the address display. When the CONTINUE switch is pressed, the program continues execution according to the displayed message code. The actions performed after the program is restarted are described in Table 2.

The RAM loader has a built-in safeguard against unintentional modifications of the writable control store. This protection mechanism is based on the fact that all hardware bootstrap programs start at even (word) addresses. Therefore, in contrast to the general rule, it is required that the bit0-switch (lowest order bit) is set to 1 (odd address) when the control store loading process is initiated. The position of the bit0-switch is tested after the first check sum test has successfully been passed. When the bit0-switch is set to 0, the program halts until the switch is set to 1 and it is reinitiated by pressing the CONTINUE switch.

Code 000 000

The RAM has been loaded without error. When the CONTINUE switch is pressed, the testprogram will be executed, starting from location 001000_8

Code 000 001

The checksum stored in location 133776_8 disagrees with the checksum calculated by the RAM loader. When the CONTINUE switch is pressed, the RAM loader checksum will be stored in location 133776_8 and the checksum test is repeated.

Code 000 002

The checksum calculated upon immediate reread of the written object microcode from the RAM disagrees with the checksum stored in location 133776_8. When the CONTINUE switch is pressed, a new RAM load is tried.

Code 000 010

The immediately reread 16-bit word disagrees with the 16-bit word written into the RAM. When the CONTINUE switch is pressed, a new WRITE/READ for the same 16-bit word is tried.

Code 017 000

A control store loading process has been initiated with the bit0-switch set to 0. When the CONTINUE switch is pressed, the test of the bit0-switch is repeated.

Table 2: RAM Loader Message Codes

2.5 Test Program

The test program for testing user microcode at the microprogram level is a PDP-11/40 machine language object program. The source of the test program is written in the UNIX assembly language. This is indicated by the suffix "s" in test program names of the form <name>.s. The object code of the test program is to be generated and stored in the file a.out by the UNIX command

as <name>.s ,

before the generation of the SMILE tape (with the microcode object file <name>.bin) is invoked by the UNIX command

smile <name>.bin a.out .

As the system loader loads the test program into consecutive main memory locations, starting from location 001 000_8 (cf. Fig. 4), instead of 000 000_8, the first instruction in the test program must set the assembler's relocation counter (..) to this address. The appropriate instruction is

.. = 1000 .

This initialization is necessary to have correct program-counter-relative addresses produced by the UNIX assembler.

When the RAM loader terminates with a message code 000 000 (cf. Table 2), pressing the CONTINUE switch invokes the execution of the test program. All instructions in the test program which have an unused PDP-11/40 op-code transfer control to one of two entry points into the control store [3,5]. Decoding of such an op-code by the user microcode in the writable control store calls upon the execution of the microprogram which defines the unused op-code. In the assembly language, unused op-codes may either be represented by an octal number or may be assigned a mnemonic representation, <code>, as follows [11].

(a) No Operand Instructions

<code> = <octal number>

(b) Single Operand Instructions

<code> = <octal number> ^tst

(c) Double Operand Instructions

<code> = <octal number> ^mov

Note that <octal number> must always represent an unused PDP-11/40 op-code [5,12].

The source code of a test program, called testn.s, for our example microprogram (cf. Fig. 1) is shown in Fig. 5.

```
..=1000
        mov     $0,r0
        mov     $1,r1
        mov     $2.r2
        mov     $3,r3
        mov     $4,r4
        mov     $5,r5
        mov     $133000,sp
        210
        0       / halt

/ test for 211
        clr  ro
        clr  r1
        clr  r2
        clr  r3
        clr  r4
        211
        0       / halt
```

Figure 5: testn.s

The first segment of testn.s tests the (unused) instruction 210_8 which is implemented by the microcode in the right column of Fig. 1. This instruction

saves registers R[2], R[3], and R[4] by pushing them onto a stack (for a subroutine call). The second segment of testn.s is a test for the (unused) instruction 211_8 whose implementation is given in the left column of Fig. 1. Instruction 211_8 restores registers R[2], R[3], and R[4] by popping them off the stack (for a return from subroutine). Instructions 210_8 and 211_8 are no operand instructions and are represented by octal numbers in testn.s.

The test for instruction 210_8 loads constants into registers R[0] to R[5] and sets the stack pointer sp (R[6]). It then executes instruction 210_8. A halt instruction (op-code 000_8) follows. Hence, when the execution of testn.s halts, the register contents can be examined by displaying them on the data display at the processor console. Pressing the CONTINUE switch invokes the execution of the test for instruction 211_8. The appropriate section of testn.s clears registers R[0] to R[4], executes instruction 211_8, and halts. Again, the register contents can be examined. The microcoded implementation of instructions 210_8 and 210_8 can then be checked at the microroutine level by comparing the register contents obtained after the execution of instruction 211_8 with the original content of the registers.

When the test program is first initiated, following the execution of the RAM loader, the stack pointer sp is automatically set to the value $133\ 000_8$. Before the first value is pushed onto the stack, sp is decremented by 2 and hence, the last element in the stack is always stored at location $132\ 776_8$ (cf. Fig. 4). The SMILE system automatically initializes the stack for use by the test program. However, if the test program is restarted from the processor console, the stack pointer sp is not reset. In this case, care must be taken that stack instructions in the test program are not affected by "garbage" in the bottom of the stack. Therefore, it is advisable to always reset the stack pointer sp in the test program, as done in testn.s (cf. Fig. 5). Moreover, this measure may also help to prevent unintentional overwriting of the test program by the stack. Nevertheless, the user must always make sure that the test program does never overlap with the test program stack. For this purpose, the SMILE system retains the header of the test program file in locations $000\ 760_8$ to $000\ 776_8$, such that the memory locations occupied by the test program can always be calculated from the byte count stored in location $000\ 762_8$ (second word in the file header).

As mentioned before, the test program may use locations 0_8 to 376_8 (which are generally reserved for the system loader) for the storage of interrupt or trap vectors. However, as the SMILE system runs on the bare machine, interrupt or trap vectors to be used in the test program must be

initialized in the program. Furthermore, the test program must also handle interrupts or traps. The general structure of an interrupt/trap vector is depicted in Fig. 6. The address assignment for PDP-11/40 interrupts vectors can be found in [12].

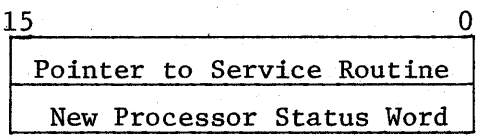

Figure 6: Interrput/Trap Vector

An example for the use of a trap vector in a SMILE test program is demonstrated by the test program, test.s, as shown in Fig. 7.

```
..=1000
        mov     $a,*$10
        mov     $3,*$12
        mov     $133000,sp
        212
        0
a:      0
```

Figure 7: test.s

This test program is used with the example microprogram, fastc.mic, as depicted in Fig. 1. It tests if illegal instructions (unused op-codes other than 210_8 or 211_8) are recognized by the decoding in fastc.mic and trapped correctly. The first instruction of test.s stores the address of label 'a' into location 010_8 (location of the interrupt vector for 'Reserved Instruction' [12]). The second instruction assigns the (arbitrary) value, 3 to the associated processor status word entry at location 012_8. If instruction 212_8 is trapped properly, test.s halts at label 'a' with the value 3 stored in the processor status register. If the microprogram fastc.mic does not recognize instruction 212_8 as an illegal instruction, test.s halts at the halt statement immediately following instruction 212_8.

An example for the implementation of an interrupt service routine is shown in Fig. 8.

```
                    :
                    :
            mov     $int,*$10
                    /set interrupt vector for reserved
                     instruction
int:        mov     (sp), r0
             0
                    /display the address of the interrupt
                     vector that caused the halt on the
                     data display
            rti
```

Figure 8: Interrupt Handler for Reserved Instructions

The last instructions of an interrupt/trap service routine should be halt and rti (return from interrupt). rti restores the program counter and the processor status word from the processor stack. Thus, after the cause of an interrupt or trap has been investigated, the test program execution may be continued by pressing the CONTINUE switch.

3. SMILE Operation

In this section, a terminal session is presented which demonstrates assembling, loading, and testing of the example microporgram fastc.mic. The microprogram and the test program testn.s are assumed to be stored under the directory /uprog/. System commands and responses start at the left margin of the page. System responses end with the prompt %.

3.1 Preparation of the SMILE Files

The microcode file and the test program file are prepared before the SMILE tape is generated.

```
mic  /uprog/fastc.mic
```

The microporgram fastc.mic is assembled.

```
/uprog/fastc.mic
91 lines read.
%
```

Now, the files fastc.lst, fastc.bin, and fastc.tab may be printed

```
strip fastc.bin
%
```

The symbol table is stripped off the file fastc.bin, as it is not needed by the SMILE system.

```
as  /uprog/testn.s
%
```

The test program testn.s is assembled by the UNIX assembler, and the object code is stored in the file a.out.

```
mv  a.out testn
%
```

The object code of the test program is moved from a.out to the file named testn.

3.2 Mounting the Tape

When the microcode object file, the test program object file, the system loader, and the RAM loader are allocated under the user working directory, a magnetic tape is mounted.

1. Mount the tape on the tape drive
2. Push the POWER switch into the on-position. The WRITE ENABLE light turns on.
3. Push the LOAD switch.
4. Push the ON LINE switch. The READ and SELECT lights turn on.
5. The tape drive is ready.

3.3 Generation of the SMILE Tape

The UNIX command "smile" is assumed to be stored under the directory /uprog/bin/.

```
/uprog/bin/smile fastc.bin testn
```

This command generates the SMILE tape which contains the system loader, the RAM loader, the object microcode fastc.bin, and the object code and the symbol table of the test program testn.

```
fastc.bin:    962/10752 bytes 2/21 blocks
testn:        60/-19952 bytes  1/89 blocks
done
%
```

For the microcode and the test program, the system specifies the number of bytes and blocks to be written on the tape and the allowed maximum size of these two files. Then, the completion of the SMILE command execution is indicated.

3.4 System Halt

When the SMILE tape is generated, the system is halted to load the tape into the bare machine.

```
halt
```

Halt the system

```
halt the system?  y
```

The system reassures the halt command. With the user response "y", the system halts and prompts

```
UNIX halted.
@ unix
```

After the system halts, the load switch on both disks must be pushed to the LOAD position such that the read/write heads are fully retracted and the spindle stops rotating.

3.5 Bootstrapping the SMILE tape

The SMILE tape is bootstrapped from the processor console.

1. Set address 773 130_8 at the switch register.
2. Push LOAD ADRS switch.
3. Push START switch to load the system loader into main memory.
4. CPU halts.
5. Push CONTINUE switch to start the system loader.
6. CPU halts.
7. Push CONTINUE switch to load the RAM loader.
8. CPU halts.
9. Push CONTINUE switch to load user microcode file
10. CPU halts.
11. Push CONTINUE switch to load the test program.
12. CPU halts.
13. Push CONTINUE switch to transfer control to the first instruction of the RAM loader.
14. Console "peeps", the console address register displays 000740_8, and the console data register displays 170000_8 (cf. Table 2).

3.6 Control Store Loading

The control store loading process is controlled according to the RAM loader messages listed in Table 2.

1. Set bit0-switch to 1.
2. Push CONTINUE switch
3. When the console "peeps' follow Table 2 until the RAM loader holds at location 000740_8 and the code $000\,000_8$ is displayed on the console data display.

3.7 Test

When the control store has successfully been loaded, pushing the CONTINUE switch invokes the execution of the test program. When the test program executes a 'halt' instruction, register contents and main memory locations can manually be examined at the processor console, (the test program might also include an output routine that prints the desired register contents on the DEC-writer).

1. CPU halts.
2. Set register or main memory address at the switch register.
3. Push LOAD ADRS switch.
4. Address in switch register is displayed on the address display.
5. Push EXAM switch.
6. Content of specified location is displayed on the data display.
7. Push CONTINUE switch (if more test program instr. are to be executed).

For the test program, testn.s, (cf. Fig. 5) the following register contents are obtained after the execution of instruction 210_8.

Address	Register	Content	Comment
777 700	R[0]	001036	Address of halt instruction
777 702	R[1]	1	
777 704	R[2]	2	
777 706	R[3]	3	
777 710	R[4]	4	
777 712	R[6]	132776	Stack start address
777 714	R[6]	132766	Stack pointer
777 716	R[7]	001040	Current program counter

The stack content observed after the execution of instruction 210_8 is given below.

Address	Content	Comment
132 776	5	Content of R[5]
132 774	4	Content of R[4]
132 772	3	Content of R[3]
132 770	2	Content of R[2]
132 766	0	Scratch value

After the execution of instruction 211_8 registers R[1] to R[4] contain the following values.

Address	Register	Content	Comment
777 702	R[1]	132770	Stack address of R[2]
777 704	R[2]	2	Restored from location 132 770
777 706	R[3]	3	Restored from location 132 772
777 710	R[4]	4	Restored from location 132 774

3.8 Bootstrapping UNIX

When the user microcode has been loaded into the control store and has been examined by the test program, it may be desirable to bootstrap the operating system. The UNIX bootstrap requires the EIS (extended instruction set) microcode in the writable control store. If the user microcode contains the EIS microcode, bootstrapping UNIX starts from step 4 of the bootstrap sequence given below. Otherwise, the user microcode in the control store is overwritten by the EIS microcode, when the following bootstrap sequence is carried out.

1. Mount the EIS SMILE tape (labelled EIS).
2. Bootstrap the EIS SMILE tape (cf. subsection 3.5).
3. Load the EIS microcode into the control store (cf. subsection 3.6).
4. Load the UNIX disk on disk drive 0.
5. Push LOAD switch at disk drive 0 to RUN position.
6. Wait for the RDY or the ON CYL light to come on.
7. Set address 773 100_8 at the switch register.
8. Push LOAD ADRS switch.
9. Set ENABLE/HALT switch to ENABLE
10. Push START switch.
11. The system responds with @ on the DEC-writer.
12. Type 'unix' and push return key.
13. The system types an identification and asks for the date to be typed.
14. Set the date.
15. UNIX is up and prompts with 'login:'.

Acknowledgement

The SMILE system was originally implemented and documented by J. Mueller of the Technical University Berlin. The authors wish to express their thanks to H. Mauersberg, also with the Technical University Berlin, for providing us with the SMILE system and for many helpful suggestions concerning the development of a microprogramming laboratory around a PDP-11/40E. They are also greatfully indepted to Profs. W. K. Giloi and W. R. Franta for initiating the microprogramming laboratory project at the University of Minnesota. The microprogramming laboratory is funded by University Computer Services, University of Minnesota.

References

[1] Mueller, J., "SMILE - Manual," Institut für Softwaretechnik und Theoretische Informatik, Fachgebiet Betriebssysteme, Technical University Berlin, December 1976.

[2] Digital, "PDP-11 Peripheral Handbook," Digital Equipment Corporation, 1975.

[3] Fuller, S. H.; Almes, G. T.; Broadley, W. H.; Forgy, C. L.; Karlton, P. L.; Lesser, V. R.; Teter, J. R.; "PDP-11/40E Microprogramming Reference Manual," Department of Computer Science, Carnegie-Mellon University, Tech. Report 16-January-1976.

[4] Teter, J.R., "PDP-11/40E Hardware Maintenance Manual," Department of Computer Science, Carnegie-Mellon University, September 1976, revised June 1977.

[5] Berg, H. K., "A PDP-11/40E Microprogramming Primer," Department of Computer Science, University of Minnesota, Tech. Report 78-8.

[6] Berg, H. K., Dekel, E., "MICRO/40 Assembler Primer," Department of Computer Science, University of Minnesota, Tech. Report 78-9.

[7] Berg, H. K., Blasing, B. E., "PDP-11/40E Microcode Simulator Primer," Department of Computer Science, University of Minnesota, Tech. Report 78-10.

[8] Berg, H. K., Covey, C. R., "A Primer on the Use of a Logic State Analyzer as a Microprogram Debugging Aid," Department of Computer Science, University of Minnesota, Tech. Report 78-12.

[9] UNIX Documentation Book I, "Introduction to UNIX," Computer Systems Laboratory, Department of Computer Science, University of Minnesota.

[10] UNIX Documentation Book II, "UNIX Programmer's Manual Section I - Commands," Computer Systems Laboratory, Department of Computer Science, University of Minnesota.

[11] UNIX Documentation Book III, "The "C" Programming Language," Computer Systems Laboratory, Department of Computer Science, University of Minnesota.

[12] Digital, "PDP-11 Processor Handbook," Digital Equipment Corporation 1973.