The individual characters within a string are indexed from 1 to the length of the string. A string variable may not be indexed beyond its current dynamic length.

String variables may be compared (=, <>, >, <, >=, <=) to other string variables, no matter what the current dynamic length of either. If the lengths of two strings being compared are unequal, the shorter string is extended to the length of the longer by appending blanks. Comparison is based on the ASCII collating sequence.

A common use of string variables in UCSD Pascal is reading file names from the console device. When a string variable is used as a parameter to READ or READLN, all characters up to the end-of-line character (carriage return) in the source file will be assigned to the string variable. In reading string variables, the single statement READLN(S1,S2) is equivalent to the two-statement sequence:

```
READ(S1);
READLN(S2);
```

3.6.17 WRITE and WRITELN

The procedures WRITE and WRITELN follow the conventions of Standard Pascal except when applied to a variable of type BOOLEAN. UCSD Pascal does not support the output of the words TRUE or FALSE when writing out the value of a boolean variable.

For writing variables of type STRING, see Section 3.1.3, String Intrinsics. When a string variable is written without specifying a field width, the actual number of characters written is equal to the dynamic length of the string. If the field width specified is longer than the dynamic length, leading blanks are inserted. If the field width is smaller, excess characters will be truncated on the right.

3.6.18 Implementation Size Limits

The maximum size limitation of UCSD Pascal are:

1. Maximum number of bytes of object code in a procedure or function is 1200. Maximum number of words for local variables in a procedure or function is 16383.

2. Maximum number of characters in a string variable is 255.

3. Maximum number of elements in a set is 255 * 16 = 4080.

4. Maximum number of segment procedures and functions is 16, of which nine are reserved for the Pascal system and seven are available to the user.

5. Maximum number of procedures or functions within a segment is 127.

3.6.19 Extended Comparisons

UCSD Pascal permits = and <> comparisons of any array or record structure.

3.7 INTRODUCTION TO THE PASCAL MACHINE

The following sections discuss aspects of software control of the Pascal MICROENGINE. Topics covered include code file representation, program execution, operating system structure, bootstrapping of the operating system, concurrent task (process) representation, and task control primitives.

3.7.1 Operating System Structure

The Pascal compiler emits code which runs directly on the WD9000 microprocessor. The compiler, screen editor, operating system, and all utilities are themselves written in Pascal and use this instruction set.

Figure 3-18 is a skeleton version of a large Pascal program, the operating system, here in after referred to as "The Program". This document is a top-down description of the realization of that program on the UCSD Pascal system. We will make occasional use of a helpful coincidence; The Program is the framework of the portion of the UCSD Pascal environment that's written in Pascal.

If The Program were expanded to a complete Pascal system, it would consist of several thousand lines of Pascal and compile to more than 50,000 bytes of code--too big to fit all at once into the memory of a small machine (current definition of small). USCD Pascal has therefore been extended so that a programmer can explicitly partition a program into segments; only some of which need be resident in main memory at a time. The syntax of this extension is shown in Figure 3-12. (Any syntactic objects not defined explicitly there retain their standard interpretation as defined by Jensen & Wirth: Pascal User Manual and Report.)

```
<program> ::= <program heading> <segment block> .

<segment block> ::= <label declaration part>
     <constant declaration part> <type definition part>
     <variable declaration part> <segment declaration part>
     <segment body>

<segment declaration part> ::= SEGMENT <procedure heading>
     <segment block>; | SEGMENT <function heading>
     <segment block>;

<segment body> ::= <procedure and function declaration part>
     <statement part>
```

Figure 3-12. Segment Declaration Syntax

Segment declaration syntax (Figure 3-12) requires that all nested segments be declared before the ordinary procedures or functions of the segment body. Thus, a code segment can be completely generated before processing of code for the next segment starts. This is not a functional limitation, since forward declarations can be used to allow nested segments to reference procedures in an outer segment body. Similarly, segment procedures and functions can themselves be declared forward.

Segmenting a program does not change its meaning in any fundamental sense. When a segment is called, the operating system checks to see if it is present in memory due to a previous invocation. If it is, control is transferred and execution proceeds: if not, the appropriate code segment must be loaded from disk before the transfer of control takes place. When no more active invocations of the segment exist, its code is removed from memory. Clearly, a program should be segmented in such a way that (non-recursive) segment calls are infrequent; otherwise, much time could be lost in unproductive thrashing (particularly on a system with low performance disk).

```
----------------------------------------------------------------------------

                    ------------------------
location           |          31            |
                    - - - - - - - - - - - - -    PASCALSYSTEM
size               |           8            |
                    ------------------------
                   |           0            |
                    - - - - - - - - - - - - -    USERPROGRAM
                   |           0            |
                    ------------------------
                   |          11            |
                    - - - - - - - - - - - - -    SYSCODE
                   |        4399            |
                    ------------------------
                   |          24            |
                    - - - - - - - - - - - - -    CSPCODE
                   |        3153            |
                    ------------------------
                   |           1            |
                    - - - - - - - - - - - - -    PRINTERROR
                   |         656            |
                    ------------------------
                   |           A            |
                    - - - - - - - - - - - - -    GETCMD
                   |         506            |
                    ------------------------

----------------------------------------------------------------------------
```

Figure 3-13. The Segment Dictionary

The code file of The Program is a sequence of code segments preceded by a segment dictionary. Code segments consist of a sequence of blocks, a 512-byte disk allocation quantum, and each code segment begins on a block boundary. The ordering (from low address to high address) is determined by the order that one encounters segment procedure bodies in passing through The Program.

The segment dictionary in the first block of a code file contains an entry for each code segment in the file. The entry includes the disk location and size (in words) for the segment. The disk location is given as relative to the beginning of the segment dictionary (which is also the beginning of the code file) and is given in number of blocks. This information is kept in the segment vector during the execution of the code file, and is used in the loading of non-present segments when they are needed. Figure 3-13 details the layout of the table and shows representative contents for the Pascal system code file.

A code segment contains the code for the body of each of its procedures, including the segment procedure, itself. Figure 3-14 is a detailed diagram of a code segment. Each of a code segment's procedures are assigned a procedure number, starting at 1 for the segment procedure, and ranging as high as 255 (current temporary limit of 127). All references to a procedure are made via its number. Translation from procedure number to location in the code segment is accomplished with the procedure dictionary at the end of the segment. This dictionary is an array indexed by the procedure number. Each array element is a segment base pointer to the code for the corresponding procedure. Since zero is not a valid procedure number, the zero'th entry of the dictionary is used to store the segment number (even byte) and number of procedures (odd byte). The outer block code is generated and appears last.

```
--------------------------------------------------------------------------------

                              high addresses
                      odd                          even
                ______________________________________________
                |  Number of procedures  |  Segment Number    |
                |     in dictionary      |                    |
                |---------------------------------------------|
                |  Procedure #1                               |--|
                | - - - - - - - - - - - - - - - - - - - - - - |  |
          |----|  Procedure #2                                |  |
          |     | - - - - - - - - - - rest of - - - - - - - - |  |
          | |--| - - - - - - -procedure dictionary - - - - - -|  |
          | |   |---------------------------------------------|  |
          | |   |                                             |  |
          | |   |              outer block code               |<-|
          | |   |                                             |
          | |   |---------------------------------------------|
          | |   |  other procedures of the Pascal system      |
          | |   |---------------------------------------------|
          | |->|  Procedure #3                       code      |
          |     |---------------------------------------------|
          |--->|  Procedure #2                       code      |
                |---------------------------------------------|
                |          Number of words in segment         |
                |_____________________________________________|

                              low addresses

--------------------------------------------------------------------------------
```

Figure 3-14. A Code Segment

A more detailed diagram of a single procedure code section is seen in Figure 3-15. It consists of two parts: the procedure code itself, and a table of attributes of the procedure. These attributes are:

EXIT IC: This is a segment-base-relative byte pointer to the beginning of the block of procedure instructions which must be executed to terminate procedure properly.

DATA SEGMENT SIZE: The data size is the size of the procedure data space (parameters and local variables) in words, excluding the markstack size.

```
--------------------------------------------------------------------------------

                high addresses
            ____________________________
            |                          |
            |        (exit code)       |
      |--->|                          |
      |    | - - - - - - - - - - - -  |
      |    |                          |
      |    |        Procedure         |
      |    |          Code            |      _________________
      |    |                          |      |               |
      |    |--------------------------|      |   Procedure   |
      |    |    Data Segment Size     |<-----|   Dictionary  |
      |    |--------------------------|      |    Pointer    |
      |----|         Exit IC          |      |_______________|
            |__________________________|

                low addresses

--------------------------------------------------------------------------------
```

Figure 3-15. Procedure Code Section

Figure 3-16 is a snapshot of system memory during the execution of a call to procedure GETCMD, which is the command processor of the operating system. SYSCOM serves as a communications area between the bootstrap and the operating system. The operating system tables consists of the TIB (task information block) and the segment vector, which is an array of information about active program segments. The Pascal heap is next in the memory layout; it grows toward high memory. The single stack growing down from high memory is used for 3 types of items: 1) temporary storage needed during expression evaluation; 2) a data segment containing local variables and parameters for each procedure activation; and 3) a code segment for each active segment procedure.

```
                         high addresses

                     _______________________
                    | Code Segment 0        |
                    | - - - - - - - - - - - |
                    | Code Segment 3        |
                    |   CSPCODE             |
                    | - - - - - - - - - - - |
                    | markstack             |
                    | - - - - - - - - - - - |
                    | Code Segment 2        |
                    |   SYSCODE             |
                    | - - - - - - - - - - - |
                    | markstack             |
                    | - - - - - - - - - - - |
                    | Code Segment 6        |
                    |   GETCMD              |
                    | - - - - - - - - - - - |
                    | markstack             |
                    | - - - - - - - - - - - |
                    | <Available Memory>    |
                    | - - - - - - - - - - - |
                    |                       |
                    |        HEAP           |
                    |                       |
                    | - - - - - - - - - - - |
                    |Operating System Tables|
                    |         and           |
                    |        SYSCOM         |
                    | - - - - - - - - - - - |
                    | Interrupt Vectors     |
                    | - - - - - - - - - - - |
                    |_______________________|

                          low addresses
```

Figure 3-16. System Memory During Operating System Execution

Consider the status of operations just before a procedure call. Conceptually, there are five pseudo-variables which point to locations in memory:

a STACK POINTER(SP): which points to the current top of the stack,

a MARK STACK POINTER(MP): which points to the "topmost" markstack in the stack, (remember that the stack grows down!), a SEGMENT POINTER(SEGB): which points to the base of the code for the currently active segment procedure,

an INSTRUCTION PROGRAM COUNTER(IPC): which contains the byte offset from the base of the code segment of the next instruction to be executed,

and (SPLOW): which points to the current top of the heap, and also serves as the stack limit pointer.

When a segment procedure is called, its code segment is loaded on the stack. The data segment is built on top of the stack. Figure 3-17 is a diagram of a data segment.

```
                          high addresses

                  _________________________<--|
                  |                       |   |
                  |-----------------------|   |
                  |                       |   |
                  |-----------------------|   |--> local variables
                  |                       |   |
                  |-----------------------|   |
                  |                       |   |
                  |-----------------------|<--|
                  |  MSFLAG  |  MSSEG     |   |
                  |-- - -- - -- - -- - -- |   |
                  |        MSIPC          |   |
                  |-- - -- - -- - -- - -- |   |--> markstack
                  |        MSDYNL         |   |
                  |-- - -- - -- - -- - -- |   |
         |----|   |        MSSTAT         |   |
         | MP |-->|-- - -- - -- - -- - -- |<--|
         |____|

                           low addresses
```

Figure 3-17. A Data Segment

In the upper portion of the data segment, space is allocated for variables local to the new procedure.

In the lower portion of the data segment is a "markstack". When a call to any procedure is made, the current values of the pseudo-variables, which characterize the operating environment of the calling procedure, are stored in the markstack of the called procedure. This is so that the execution state may be restored to pre-call conditions when control is returned to the calling procedure.