SECTION 1

CPU MEMORY

CPU memory is a physical address space from >00000 to >7FFFF. The last 64K bytes of this space are reserved for system roms, etc., so that the range of ram will be from >00000 to >6FFFF. The ram from >00000 to >0FFFF will be built into the console. Additional ram may be added in increments of 32 or 64 K, with the requirement that they be contiguous.

The system software will determine the amount of ram present, and store the top-of-memory in a two word pointer CMTOP. All system software will assume that the address space will eventually be expanded to 32 bits (4 billion bytes.)

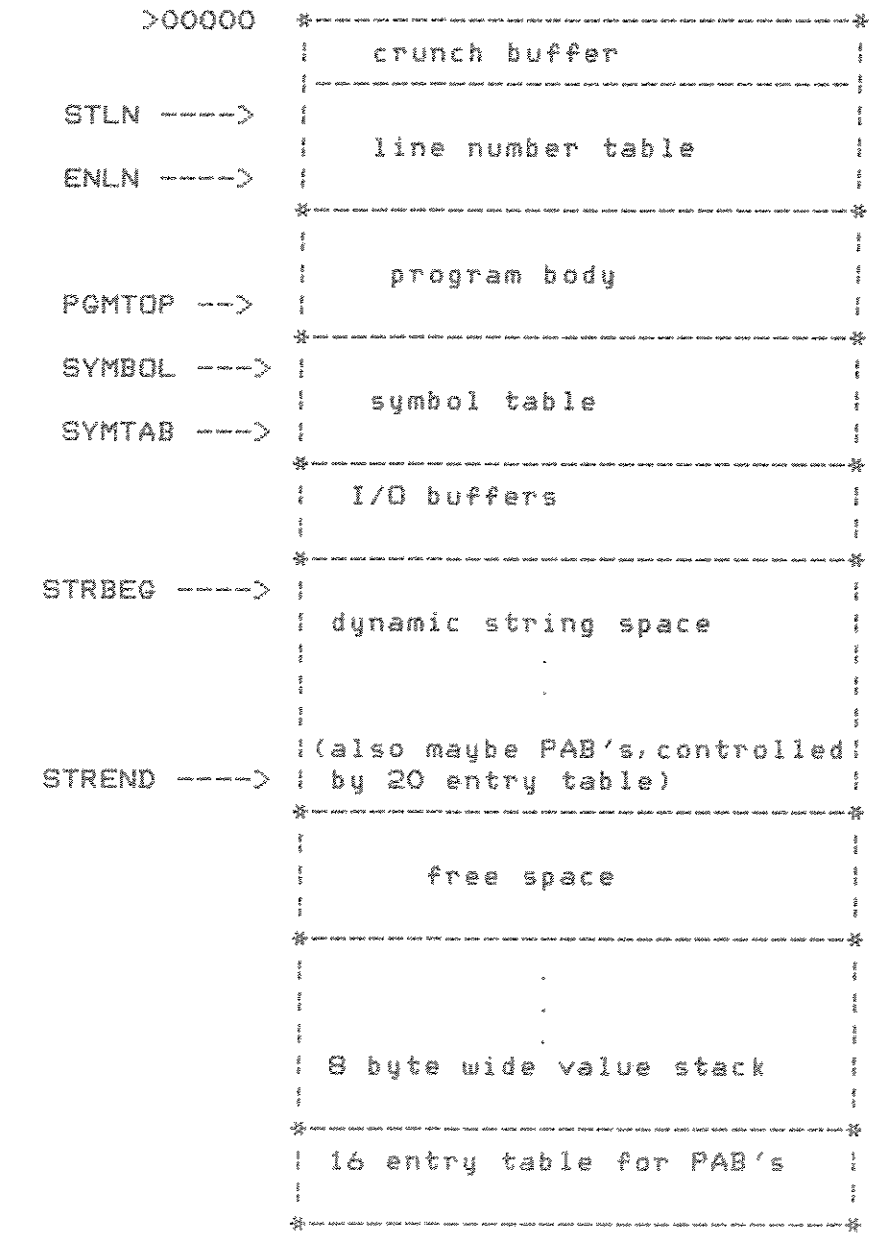
Peripheral devices will be able to allocate blocks of physical ram in a similar way to that in which they currently allocate VDP ram. The logical memory page at >A000 is available at powerup time for this purpose. The Device should allocate memory in the following way:

```
  CMTOP---> <last free byte>
CMTOP+1---> OLD CMTOP(4 BYTES)
            IDENTIFICATION BYTE (USUALLY FROM CRU BASE)
            <work space for this block>
```

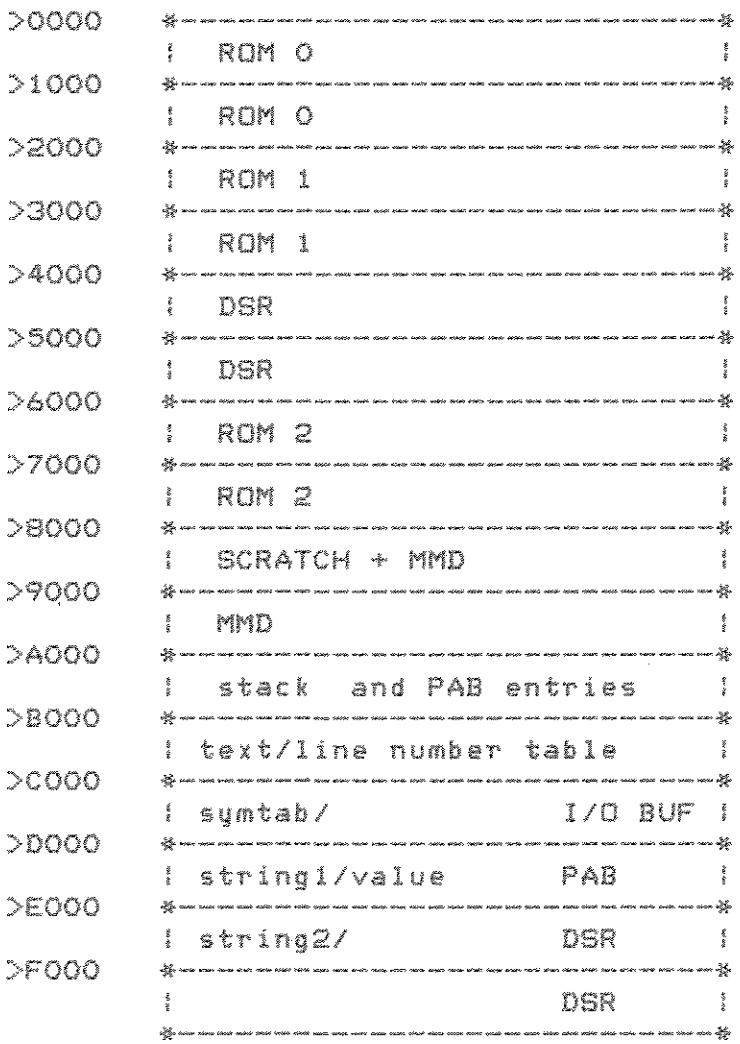
Since the "old CMTOP" is a pointer to a previously allocatred block, the blocks will form a linked list, with double word links.

When Basic is called, there will be some amount of memory available, as determined by (CMTOP)+1. Since this available memory may be larger than 64K bytes, (1) Basic will be required to keep some 32-bit pointers, and (2) some data structures will be limited to 64K bytes so that "local" 16 bit pointers may be used.

Memory will be used as follows:



LOGICAL MEMORY. Logical memory will have some fixed assignments, and some variable ones. The "normal" assignment of windows will be as follows:



THE LINE NUMBER TABLE AND THE PROGRAM BODY. This block will begin at the first available byte after the interpreter variable area. It will extend no further than >OFFFF. This means that (1) the program size is limited to about 63.5K bytes, and (2) The pointers STLN, ENLN, PGMTOP, and the statement pointers within the line number table may be stored as 16-bit physical pointers, with the most-significant 16 bits implied to be zero.

SECTION 2

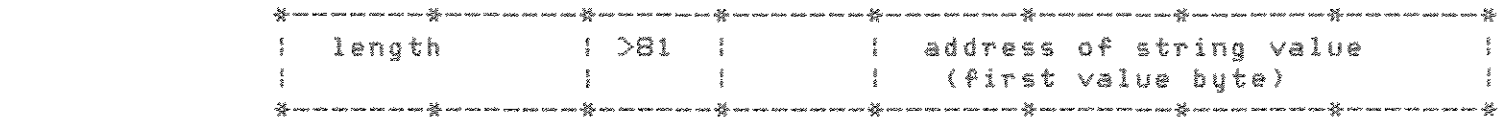
THE VALUE STACK

The value stack will have a maximum size of 4K bytes. This means that we can map in STKBEG at the beginning of a 4K page and have access to all of the stack. Also, STKTOP becomes a single word that we increment or decrement by 8 whenever we pop or push. (We must also maintain a logical top of stack pointer the same way.) Floating point numeric values will be 8-byte internal format numbers. The radix 100 notation assures that the third byte will be in the range of 0 to 99 (>00 to >63). This third byte is thus used as a tag field to identify the type of the stack entry.

2.1 String entries

(1) string associated with variable. Refer to symbol table entry to find value.

(2) temporary string.

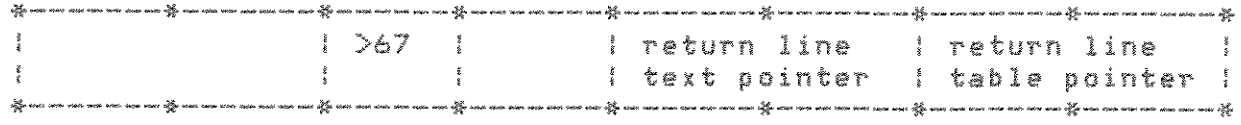


The owner will either be a symbol table entry, or the stack entry for a temporary string. A string can have only one owner for garbage collection purposes. Note that the string space is not limited to 64K bytes, and that strings are not limited to 255 bytes.

CAUTION

A string entry is considerably different than in 99/4 console or extended basic, and many code changes may be necessary.

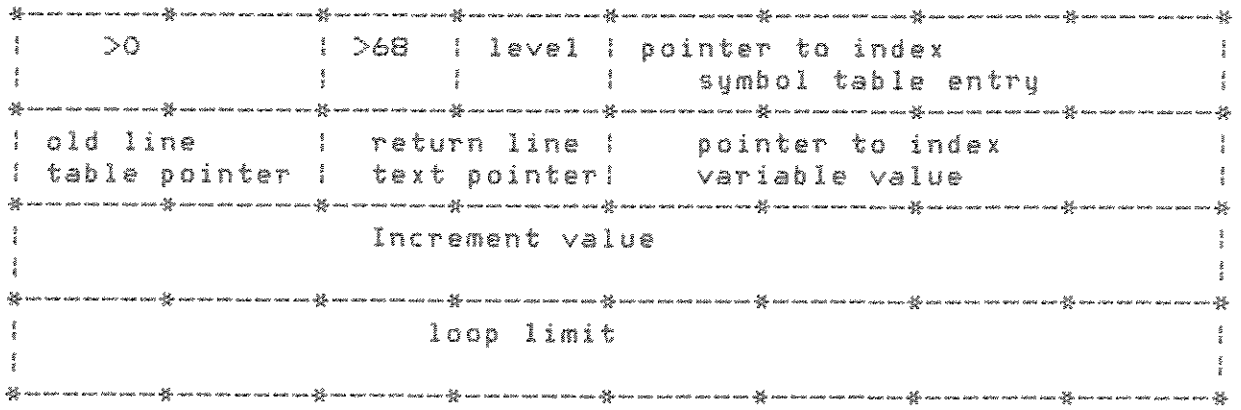
2.2 Gosub entry



This is copied from extended basic and supports multiple statement lines.

2.3 FOR statement

(1) with a floating point index.



NOTE

The index value pointer could be a single word, as an offset within the symbol table, since only scalar variables are permitted.

(2) with an integer index.

not >0	>6B	level	pointer to index symbol table entry
old line table pointer	return line text pointer		pointer to index variable value
Increment value			
Limit value			

2.4 User-defined function

return line text pointer	>6A	funct. type	old symbol table pointer
return line table pointer			

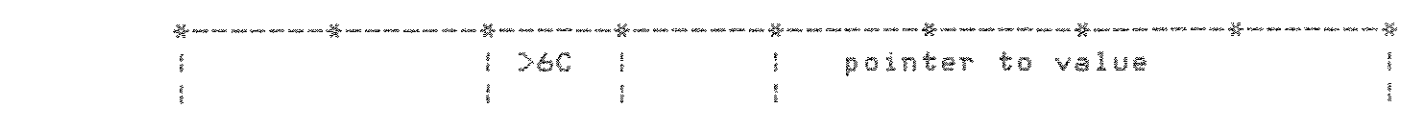
2.5 Integer value

value	>6B		>00000000

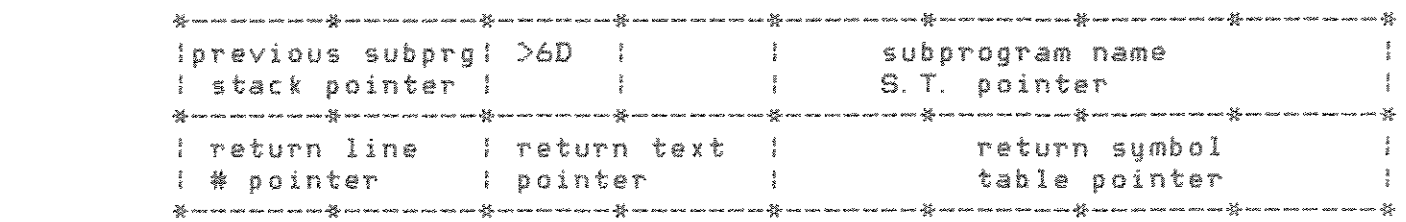
2.6 Integer variable

value	>6B		Pointer to value in S.T.

2.7 Real variable



2.8 Subprogram, Basic

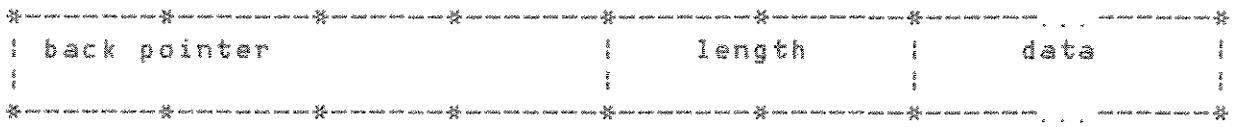


SECTION(THE PAB TABLE) ATA The PAB table is located at the high end of the value stack. ATA It contains 16 16-byte entries. Each entry contains two value ATA stack-type temporary string pointers. The first is for the PAB, and ATA the second for the buffer. The PAB and buffer themselves are allocated ATA from string space, with back pointers to the PAB table. ATA An active file entry appears as: ATA BEGIN MARGINED INSERT ATA ATA *-------
--*----------*----------*----------*----------*----------*----------* ATA ! !
>81 ! file #! address of PAB string ! ATA ! ! ! ! ! ATA *--------
--------------------*----------*----------*----------*----------* ATA ! !
>81 ! file #! address of buffer string ! ATA ! ! ! ! ! ATA *------
--*----------*----------*----------*----------*----------*----------* ATA
ATA END INSERT ATA An unused file entry appears as 16 zero byes. ATA

SECTION 3

THE STRING SPACE

The string space will have to be managed so that it can contain back pointers to either the stack or the symbol table, which may not be in the same 64K. In addition, the symbol table itself may be larger than 64K. Thus back pointers must be double word physical pointers. Only one length word is provided, at the position required for garbage collection. All pointers to strings should point to the first data byte. The data may be of either even or odd length.



SECTION 4

THE SYMBOL TABLE

The symbol table is not limited to 64K bytes, so all pointers to it must be double words. In order to be able to calculate array element positions, it is necessary to limit the value space of each array to 64K bytes. Thus a string array may have 16K elements, a real array may have 8K elements, and an integer array may have 32K elements.

1. The first two bits of a symbol table entry indicates the symbol type.

Type	
0	floating point
1	string
2	integer
3	subprogram (extended only)

2. The third bit is a shared flag (extended only) to indicate that the variable is a formal parameter in a subprogram.

3. The forth bit indicates a user-defined function.

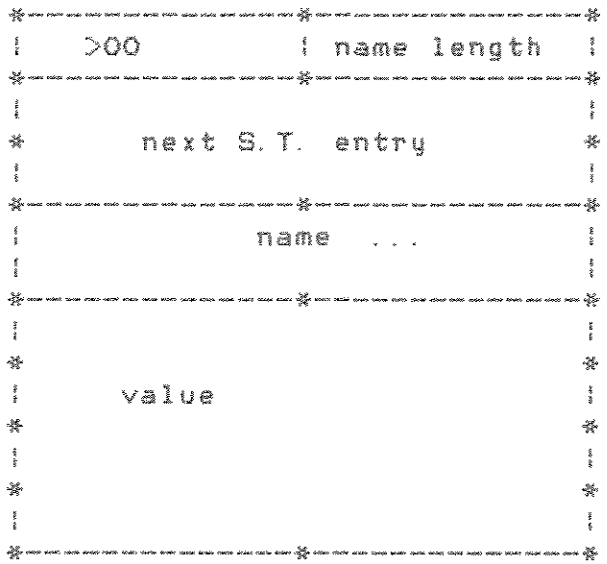
4. The fifth through eighth bits indicate the number of diminsions for an array.

5. The second and third words of the entry are a physical pointer to the next symbol table entry in the chain.

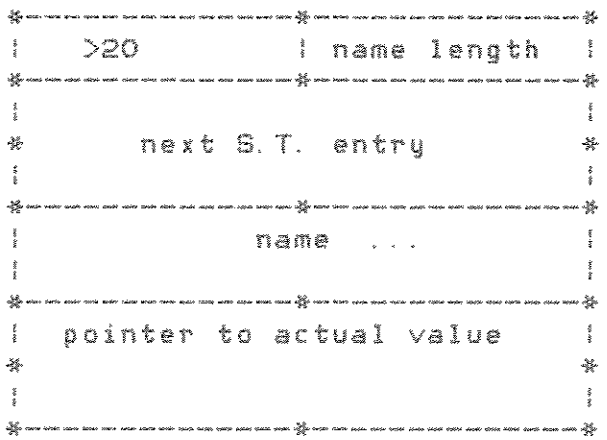
6. The name begins in the forth word. An extra byte is added, if necessary, to keep the following value space on word boundaries.

7. Value information follows the name. The name length must be used to find it. (Because of the amount of symbol table searching done, it is better to keep the name in a fixed position.)

>00. Floating point numeric scalar

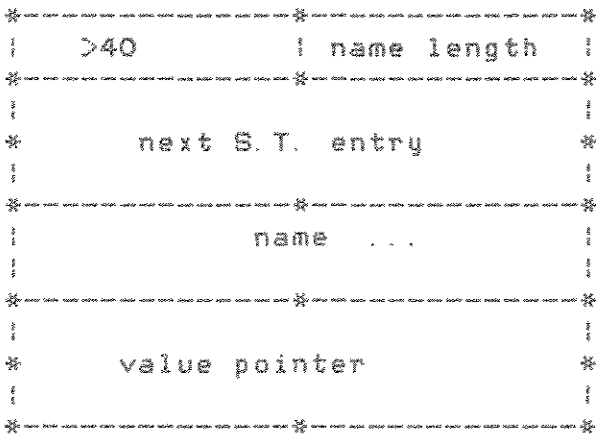


>20. Floating point numeric scalar, shared

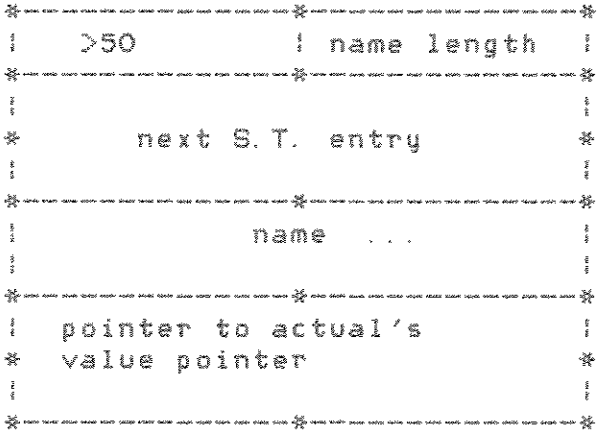


>30. not supported

>40. String scalar

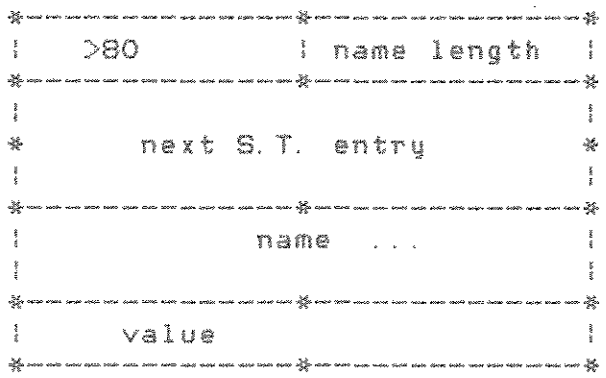


>60. String scalar, shared

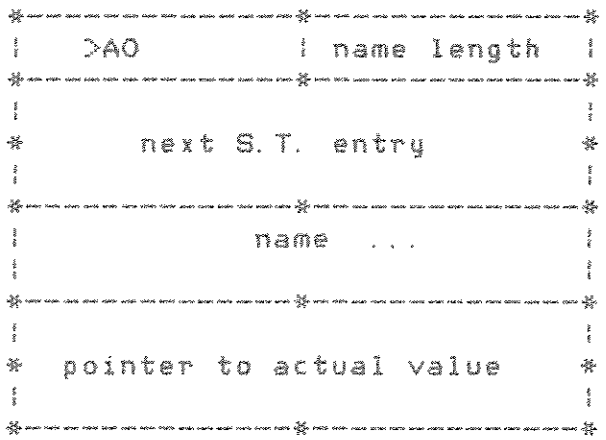


>70. not supported

>80. Integer numeric scalar

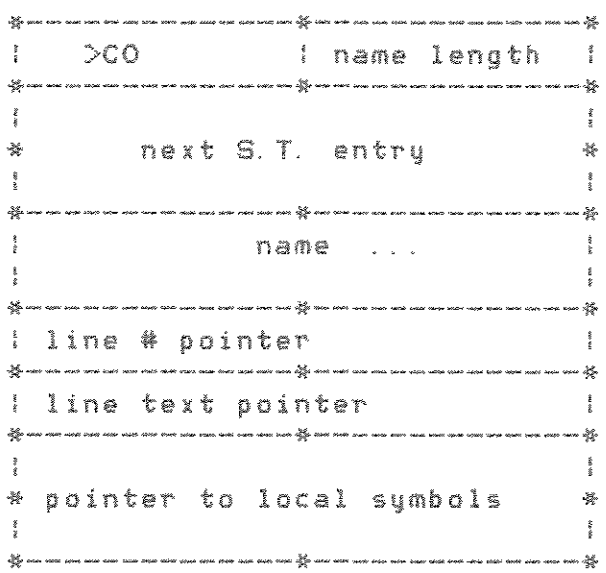


>A0. Integer scalar, shared



>B0. not supported

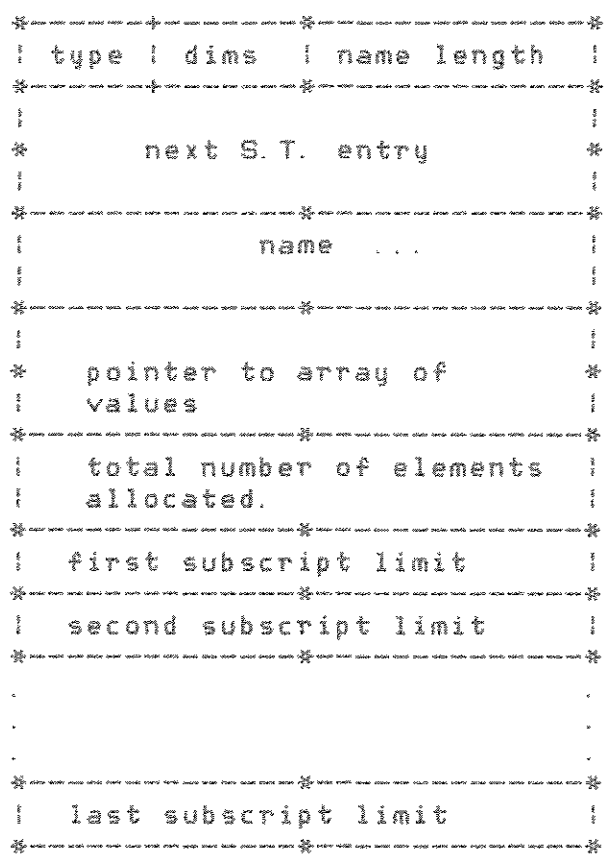
>C0. Subprogram.



The diminsion bits are used as
(1) an "in use" flag
(2) a GPL vs. Basic vs. Assembly language selector

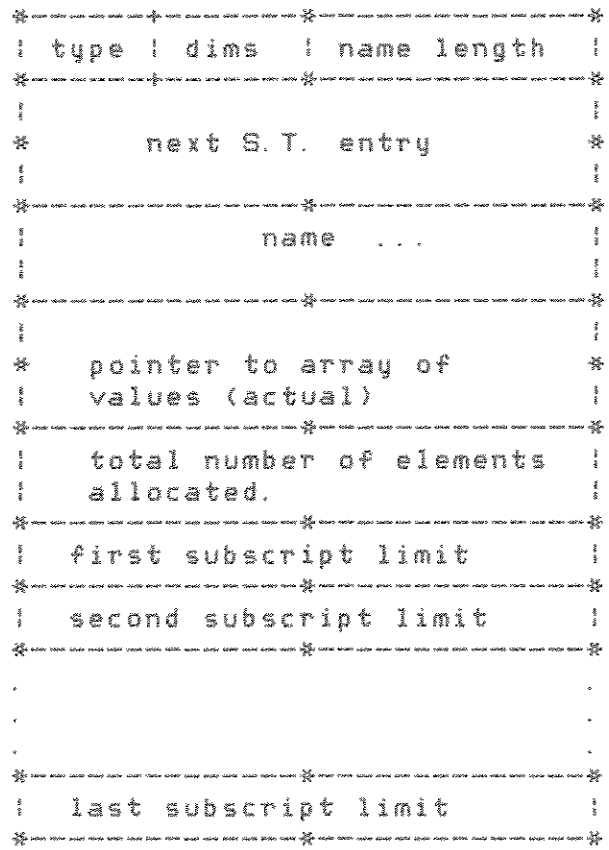
>D0, >E0, >F0. Not supported

>0X, >4X, >8X array, not shared.

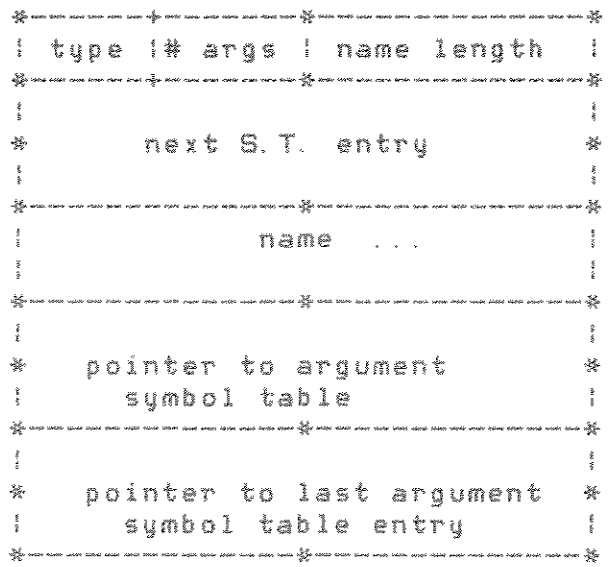


The array of values is a linearized array of entries, each of which is two bytes for integers, 4 bytes for strings, and 8 bytes for reals. The total number of elements allocated is saved, to permit a future extended basic to perform rediminsioning operations (either explicitly or automatically in support of MAT commands.)

>2X, >6X, >AX array, shared.



>10, >50, >90, User-defined function




line table pointer for def
line text pointer for def