9. MACHINE LANGUAGE ROUTINES

This chapter is for the small number of muLISP users who may find it necessary to write machine language routines and call them from muLISP. There are three major reasons for writing such routines: a) to control some computer hardware or peripheral device; b) to communicate with the host operating system; c) to implement some critical user-defined functions in machine language for efficiency reasons. The services provided by the Hardware Interface Functions (see Chapter 5) are intended to minimize as much as possible the need to write machine language routines.

This chapter describes how to write and link to machine language routines if you should need to do so. It assumes that you are thoroughly familiar with the muLISP data structures as described in Chapter 4 of this manual and with the architecture and assembly language of the 8086 microprocessor.

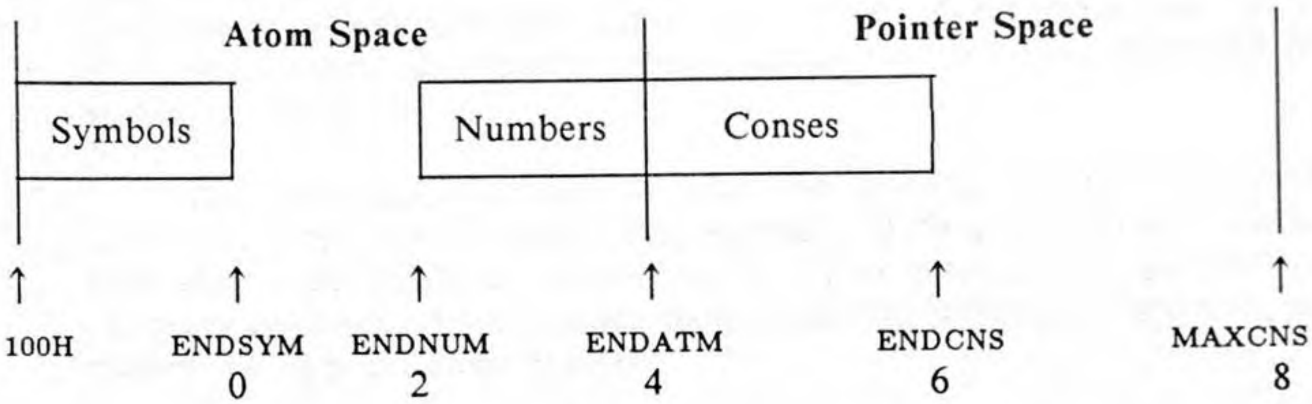
Appendix B to this manual is an assembly language listing of the muLISP **base page**. The base page consists of the first 300H bytes of MULISP.COM. It contains the default values for several system variables (e.g. computer type, line length, radix base, etc.), the default readtable, the macro character function table, the ten pseudo-registers, the line editor command table, and space for entry points into user-defined special form routines.

The commented listings show the addresses of these items in the event you need to change their default value. The change can be done and a revised MULISP.COM can be saved using the assembly language debug program supplied with your computer's operating system.

Appendix C to this manual is an assembly listing of the muLISP **global storage area**. This area contains the variables (i.e. two byte pointers) that delimit the ends of the various muLISP data spaces and other system variables. These variables can be accessed from assembly language programs beginning at an offset of 0 in the data segment.

9.1 DATA REPRESENTATION

In order to write machine language routines, you must have an understanding of how and where muLISP stores data objects in memory. The following diagram illustrates where the three types of objects are stored:



The diagram also gives the names of the variables that point to the current end of the various regions. These pointers are maintained by muLISP in the global storage area at the offsets indicated under their name. As an efficiency measure, the register DX contains the same value as ENDATM.

Using these variables, it is easy for a machine language routine to determine the type of a data object (a symbol, a number, or a cons) merely from its address. For example, if SI points to the object, the following code jumps to one of three routines based upon the object's type:

```
ENDSYM  EQU     0
ENDNUM  EQU     2
ENDATM  EQU     4

        CMP     SI,WORD PTR DS:[ENDSYM]
        JB      LABEL1          ;Jump if SI is a symbol
        CMP     SI,DX           ;Note that DX = ENDATM
        JB      LABEL2          ;Jump if SI is a number
        JMP     LABEL3          ;SI is a cons
```

Unfortunately the 8086 divides its 20 bit address space into **segments** which can only be a maximum of 64 Kilobytes each. The pointer cells making up a muLISP data object are stored at the same offset in different segments. The object's car cell is stored in the segment pointed to by the DS segment register. Its cdr cell is stored in the segment pointed to by the ES segment register. If the object is an atom it has a third and fourth cell which are stored in the definition and print name pointer segments respectively. The 16 bit paragraph addresses of these segments are stored in the variables DEFSEG and PNPSEG in the global storage area.

Knowing this, it is easy for machine language routines to access the contents of an object's cells. For example, if SI points to a symbol, the following code loads a pointer to its value into AX, a pointer to its property list into BX, a pointer to its function definition into CX, and a pointer to its print name into DI:

```
DEFSEG  EQU     10H
PNPSEG  EQU     12H

        MOV     AX,[SI]         ;AX points to symbol's value
        MOV     BX,ES:[SI]      ;BX points to symbol's plist
        PUSH    DS
        MOV     DS,WORD PTR DS:[DEFSEG]
        MOV     CX,[SI]         ;CX points to symbol's defn
        POP     DS
        PUSH    DS
        MOV     DS,WORD PTR DS:[PNPSEG]
        MOV     DI,[SI]         ;DI points to symbol's pname
        POP     DS
```

Since the symbol's value and property list must be bona fide muLISP objects, they can be accessed as described earlier. If the definition pointer in CX is odd, it contains the offset of a machine language routine in the code segment (CS). If CX is even, it contains the offset of D-code, which is stored in the stack segment (SS). The print name pointer in DI contains the offset of the ASCII print name string which is stored in the print name string segment (PNSSEG).

The contents of the four cells making up a number can be found by the same code as shown above for obtaining a symbol's four cells. The vector cell of a big integer contains the offset of the number vector which is stored in the number vector segment (VECSEG). Note that VECSEG and PNSSEG are variables maintained in the muLISP global storage area.

9.2 ARGUMENT PASSING

User-defined machine language routines can be either **eval functions** or **special forms** (see Evaluation Functions section of Chapter 5). The entry point for eval function routines must be on an *odd* address. The entry point for special forms routines can be on an even or odd address since they are only referenced indirectly via the jump table located in the base page.

Arguments to functions are evaluated, and the address of the resulting object is passed to the machine language routine above the top of the variable (BP) stack. On entry, CX will contain two times the number of actual arguments that were in the call to the function. If the following routine is called at the beginning of a function, it loads SI, DI, and BX with the first three arguments to the function. If there are not enough arguments, it loads the registers with NIL:

```
NIL      EQU     100H         ;Offset of NIL is a constant

GETTHR:  CMP     CX,6
         JB      GETTWO       ;Jump if less than 3 arguments
         MOV     SI,[BP]      ;SI: argument one
         MOV     DI,[BP+2]    ;DI: argument two
         MOV     BX,[BP+4]    ;BX: argument three
         RET

GETTWO:  MOV     BX,NIL       ;BX: NIL
         CMP     CX,4
         JB      GETONE       ;Jump if less than 2 arguments
         MOV     SI,[BP]      ;SI: argument one
         MOV     DI,[BP+2]    ;DI: argument two
         RET

GETONE:  MOV     DI,NIL       ;DI: NIL
         JCXZ    GETZRO       ;Jump if no arguments
         MOV     SI,[BP]      ;SI: argument one
         RET

GETZRO:  MOV     SI,NIL       ;SI: NIL
         RET
```

Since the arguments to special forms are not evaluated, the address of the actual argument list is passed to the routine in the SI register. Since the special form could have been invoked either from a compiled function or from evaluating an S-expression, the list can be either D-code or a linked list. You must write separate routines to handle each case. The offset address of the two routines is then stored in a two entry jump table. Space for 10 special form jump tables has been set aside in the base page for user-defined special forms. The address of the D-code handler is stored in the first entry; the address of the S-expression handler in the second.

The value returned by a machine language routine is the object pointed to by the DI register. DI must contain the offset address of a bona fide muLISP data object. If the routine does not need to return a value, DI should still be set to the offset of NIL which is 100H.

9.3 CALLING PRIMITIVE ROUTINES

User-defined routines can call the large number of primitively defined muLISP function and special form routines. These primitive routines use the same **calling convention** to receive arguments and return values as that described for user-defined routines. The offset address in the code segment of the functions can be determined from muLISP using the function GETD:

```
$ (SETQ *PRINT-BASE* 16)                    ;Print in hexadecimal
10

$ (GETD 'CONS)                              ;Get CONS's entry point
0B27

$ (GETD 'LOOP)                              ;Get LOOP's jump table
1B8

$ (SETQ *PRINT-BASE* 10)                    ;Print in decimal
10
```

Note that user-defined routines must save the DX and the four 8086 segment registers, so that they can be restored to their original value before a call is made to a primitive routine and before control is returned to muLISP.

Calls to muLISP routines that do consing or generate numbers can invoke the garbage collector. Garbage collections will destroy unreferenced muLISP data objects and can change the address of referenced objects. To ensure that pointers to objects are saved and updated during a garbage collection, they must be pushed onto the muLISP variable stack. The BP register is the variable stack pointer. The following code segment illustrates how the pointer in BX is pushed on and later popped off the variable (BP) stack for a call to the muLISP CONS routine:

```
CONS    EQU     0B27H

        MOV     [BP],BX         ;Push BX on BP stack
        INC     BP              ;Increment BP stack pointer
        INC     BP
        MOV     AX,OFFSET CONS+3
        EVEN                    ;Force CALL to an odd address
        NOP
        CALL    AX              ;Set DI to CONS (SI, DI)
        DEC     BP              ;Decrement BP stack pointer
        DEC     BP
        MOV     BX,[BP]         ;Pop updated BX from BP stack
```

Note the use of the EVEN and NOP opcodes immediately preceding the call to CONS. The assembly language instruction CALL AX is a two byte instruction. Therefore, if the call instruction begins on an odd address, the return address that is pushed on the SP stack will be odd.

During a reallocating garbage collection references on the SP stack to D-code must be updated when the D-code is moved during reallocation. To distinguish D-code pointers from return addresses on the SP stack, the reallocator relies on the fact that D-code pointers are even whereas return addresses are odd. Therefore, it is necessary to ensure that a return address (or anything else for that matter) that is pushed on the SP stack is odd *if* a garbage collection can occur while the return address is on the stack.

9.4 LOADING AND LINKING

A three step process is used to load and link user-defined machine language routines into muLISP. First, the function ALLOCATE is called to set aside enough space in the code segment for the routine. Next, the function BINARY-LOAD is called to actually load the routine into memory. Finally, the function PUTD is called to link a symbol's function definition cell to the routine.

If *bytes* is a positive integer, (ALLOCATE *bytes*) frees *n* bytes of memory in the code segment and returns the offset of the base of the newly allocated memory. Note that if ALLOCATE returns an integer, it will be an even integer.

If *offset* is a positive integer less than 65536 and the file on the drive and directory specified by *filename* exists, (BINARY-LOAD *filename offset*) loads *filename* into the muLISP code segment at *offset* and returns the number of bytes loaded. If no file name extension is specified in *filename*, BINARY-LOAD assumes a file name extension of BIN. Normally the *offset* in a call to BINARY-LOAD is the offset address returned by the call to ALLOCATE plus 1 to make it odd.

If *name* is a symbol and *n* is the odd entry address (i.e. its offset in the code segment) of a user-defined machine language function routine, (PUTD *name n*) links *name* to the function. If *n* is the even offset address of a special form jump table in the base page, (PUTD *name n*) links *name* to the special form routines pointed to by the table.

The following function (LOAD-LINK *name filename bytes*) is a useful utility that automates the process for loading and linking user-defined machine language routines:

```
(DEFUN LOAD-LINK (NAME FILE-NAME BYTES
    OFFSET )
  ((SETQ OFFSET (ALLOCATE BYTES))          ;Allocate memory
    ((BINARY-LOAD FILE-NAME (ADD1 OFFSET)) ;Load file
      (PUTD NAME (ADD1 OFFSET)) ) ) )      ;Link function
```

As an example of the major points discussed in this chapter, consider the problem of converting the following user-defined function into a machine language routine:

```
(DEFUN COPY (OBJ)
  ((ATOM OBJ) OBJ)
  (CONS (COPY (CAR OBJ)) (COPY (CDR OBJ))) )
```

COPY is equivalent to the primitive function COPY-TREE. Appendix D to this manual is a listing of the equivalent 8086 assembly language routine for COPY. If the routine is assembled and turned into the file COPY.BIN, it can be loaded and linked to the symbol COPY as follows:

```
$ (LOAD-LINK COPY B:COPY 60)          ;Load and link COPY
COPY

$ (COPY '(A (B . C) D))               ;Call the new function COPY
(A (B . C) D)
```

A. IBM PC EXTENDED FUNCTION KEYS

The IBM PC keyboard has a set of **extended function keys**. They include the ten function keys located on the left or top of the keyboard and the cursor control keys located on the right side of the keyboard. See the IBM PC's Guide to Operations Manual and the BASIC Programming Manual for details.

If you are running muLISP on an IBM PC, programs can detect that the user has pressed one of the extended function keys if READ-BYTE returns a 255. The *next* byte returned by READ-BYTE indicates which extended function key was pressed as shown in this table.

Byte	Key	Byte	Key	Byte	Key	Byte	Key
15	SHIFT |<—	48	ALT-B	85	SHIFT-F2	109	ALT-F6
16	ALT-Q	49	ALT-N	86	SHIFT-F3	110	ALT-F7
17	ALT-W	50	ALT-M	87	SHIFT-F4	111	ALT-F8
18	ALT-E	59	F1	88	SHIFT-F5	112	ALT-F9
19	ALT-R	60	F2	89	SHIFT-F6	113	ALT-F10
20	ALT-T	61	F3	90	SHIFT-F7	114	CTRL-PrtSc
21	ALT-Y	62	F4	91	SHIFT-F8	115	CTRL <—
22	ALT-U	63	F5	92	SHIFT-F9	116	CTRL —>
23	ALT-I	64	F6	93	SHIFT-F10	117	CTRL-End
24	ALT-O	65	F7	94	CTRL-F1	118	CTRL-PgDn
25	ALT-P	66	F8	95	CTRL-F2	119	CTRL-Home
30	ALT-A	67	F9	96	CTRL-F3	120	ALT-1
31	ALT-S	68	F10	97	CTRL-F4	121	ALT-2
32	ALT-D	71	Home	98	CTRL-F5	122	ALT-3
33	ALT-F	72	↑	99	CTRL-F6	123	ALT-4
34	ALT-G	73	PgUp	100	CTRL-F7	124	ALT-5
35	ALT-H	75	<—	101	CTRL-F8	125	ALT-6
36	ALT-J	77	—>	102	CTRL-F9	126	ALT-7
37	ALT-K	79	End	103	CTRL-F10	127	ALT-8
38	ALT-L	80	↓	104	ALT-F1	128	ALT-9
44	ALT-Z	81	PgDn	105	ALT-F2	129	ALT-0
45	ALT-X	82	Ins	106	ALT-F3	130	ALT -
46	ALT-C	83	Del	107	ALT-F4	131	ALT-=
47	ALT-V	84	SHIFT-F1	108	ALT-F5	132	CTRL-PgUp

B. muLISP BASE PAGE

Appendix B is an annotated assembly language listing of the muLISP base page. It shows the initial or default value for many system variables. The first two columns in each row of the listing are the hexadecimal and decimal offsets in the code segment of the variables.

```
                ORG     100H     ;muLISP base address
0100    256     JMP     LISP     ;Jump to entry point

                DS      125      ;Reserved space, do not use
```

The following **Special Form Jump Table** provides space for up to 10 pairs of pointers to user-defined special forms. The first entry of each pair points to D-code; the second points to the S-expr routine. See Chapter 9 for details on implementing user-defined special forms.

```
0180    384     DW      0,0
0184    388     DW      0,0
0188    392     DW      0,0
018C    396     DW      0,0
0190    400     DW      0,0
0194    404     DW      0,0
0198    408     DW      0,0
019C    412     DW      0,0
01A0    416     DW      0,0
01A4    420     DW      0,0

01A8    424     DS      88       ;Reserved space, do not use
```

The following **Default Read Table** specifies the default character type for the 256 different ASCII characters. The bits in each byte in the table describe the type of the corresponding ASCII character as follows:

Bit	Mask	Character type	Bit	Mask	Character type
0	1	Whitespace	4	10H	Interrupt
1	2	Single escape	5	20H	Terminating macro
2	4	Multiple escape	6	40H	Nonterminating macro
3	8	Break	7	80H	Comment macro

```
;             @ A B C D E F G H I J K L M N O
0200 512  DB  0,0,0,0,0,0,0,0,0,1,1,0,1,1,0,0

;             P Q R S T U V W X Y Z  [ \ ] ^ _
0210 528  DB  0,0,0,0,0,0,0,0,0,0,0,16,0,0,0,0

;               ! " # $ % & ' ( ) * + , - . /
0220 544  DB  1,8,40,8,8,8,8,40,40,40,8,8,40,8,8,8

;             0 1 2 3 4 5 6 7 8 9 :  ; < = > ?
0230 560  DB  0,0,0,0,0,0,0,0,0,8,128,8,8,8,8

;             @ A B C D E F G H I J K L M N O
0240 576  DB  8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

;             P Q R S T U V W X Y Z [  \  ] ^ _
0250 592  DB  0,0,0,0,0,0,0,0,0,0,0,8,10,40,8,8

;             ' a b c d e f g h i j k l m n o
0260 608  DB  8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

;             p q r s t u v w x y z {  | } ~ RUB
0270 624  DB  0,0,0,0,0,0,0,0,0,0,0,8,12,8,8,0

0280 640  DB  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

0290 656  DB  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

02A0 672  DB  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

02B0 688  DB  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

02C0 704  DB  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

02D0 720  DB  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

02E0 736  DB  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

02F0 752  DB  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

The following **Default Macro Character Function Table** contains pointers to the default definitions for macro characters. The first word of the table is the number of entries in the table. There is room for up to 10 entries. An entry consists of a byte containing the ASCII code for a character followed by a word pointing to its macro definition; odd pointers imply machine language routines, even imply D-code.

```
0300  768  DW  4          ;Entries in table
0302  770  DB  "'"        ;Macro character
           DW  RDQUOTE    ;Macro function pointer
0305  773  DB  '('
           DW  RDLIST
0308  776  DB  ';'
           DW  RDCOMMENT
030B  779  DB  '"'
           DW  RDSTRING
030E  782  DB  0
           DW  0
0311  785  DB  0
           DW  0
0314  788  DB  0
           DW  0
0317  791  DB  0
           DW  0
031A  794  DB  0
           DW  0
031D  797  DB  0
           DW  0
```

The following **Pseudo-registers** are used by REGISTER & INTERRUPT.

```
0320  800  DW  0    ;0   AX
0322  802  DW  0    ;1   BX
0324  804  DW  0    ;2   CX
0326  806  DW  0    ;3   DX
0328  808  DW  0    ;4   SI
032A  810  DW  0    ;5   DI
032C  812  DW  0    ;6   BP
032E  814  DW  0    ;7   DS
0330  816  DW  0    ;8   ES
0332  818  DW  0    ;9   FLAGS
```

If the following minimum number of bytes are not free after a garbage collection and data space reallocation, a "Memory Full" error break occurs.

```
0334   820   DW    200H    ;Minimum atom space bytes
0336   822   DW    200H    ;Minimum pointer space bytes
0338   824   DW    400H    ;Minimum stack space bytes
```

The data space reallocator allocates stack space so that the total free space to stack free space is the following.

```
033A   826   DW    32      ;Total free to stack free space

033C   828   DW    1       ;Default precision setting (single)
033E   830   DW    0       ;Default underflow setting (disabled)
0340   832   DW    7       ;Default value of *PRINT-POINT*

0342   834   DW    CSREG   ;Initial value of the CS register
```

A data space reallocation occurs after a garbage collection if, for any space, the ratio of the size of the space to the optimum size of the space is greater than [0344H]/[0346H] or less than [0346H]/[0344H].

```
0344   836   DW    4       ;Reallocation sensitivity
0346   838   DW    3       ;Reallocation sensitivity
```

muLISP will *not* use memory above the paragraph address in the following word. If it is zero, muLISP uses all available memory.

```
0348   840   DW    0       ;Limit on muLISP memory size
```

The following word is the default file and printer line length.

```
034A   842   DW    79      ;Default file line length
```

The following bytes are the initial radix base used for I/O.

```
034C   844   DB    10      ;Initial value of *READ-BASE*
034D   845   DB    10      ;Initial value of *PRINT-BASE*
```

The following byte is used as a mask on character input from files to strip parity bit.

```
034E   846   DB    7FH     ;File input character mask
```

The following byte is the character output by muLISP to sound the console bell. Set byte to 0 to deactivate bell.

```
034F   847    DB     7          ;Console bell character
```

The following bytes are the characters used by PRIN1 to delimit special characters when the control variable *PRINT-ESCAPE* is nonNIL.

```
0350   848    DB     '\'        ;Single escape character
0351   849    DB     '|'        ;Multiple escape character
```

The following byte controls the frequency at which the console is checked for escape char. Value can range from 1 for most frequent to 255 for least.

```
0352   850    DB     100        ;Keyboard interrupt sensitivity
```

If the following byte is nonzero, whitespace characters that terminate a READ will be unread so that they can be reread by next READ.

```
0353   851    DB     -1         ;Read preserving white space flag
```

If the following byte is nonzero, an "End-Of-File" error break occurs when the EOF character is encountered while reading a file. If zero, EOF error break only occurs when the physical EOF is reached.

```
0354   852    DB     -1         ;Logical EOF flag
```

The following byte is recognized as the logical EOF character if the above logical EOF flag byte is nonzero.

```
0355   853    DB     1AH        ;EOF character
```

The following byte is the exit code returned by muLISP if the function SYSTEM is called without arguments or if the break option "System" is chosen.

```
0356   854    DB     0          ;Default exit code
```

The following byte stores the computer type number from the following table. It is used by the muLISP console screen and graphics functions.

0 = Unknown computer type	5 = Zenith Z-100 or VT-52
1 = Generic MS-DOS computer	6 = Hewlett-Packard HP-150
2 = IBM PC or "look-alike"	7 = Hewlett-Packard HP-110
3 = ANSI or VT-100 Terminal	8 = NEC APC or ADM-3A
4 = TI Professional Computer	9 = NEC PC-9801

```
0357   855    DB     0          ;Computer type byte
0358   856    DS     4          ;Reserved space, do not use
```

If the following byte is nonzero, the muLISP pseudo-code compiler is included in COM files created by the function SAVE. Only COM files *without* the pseudo-code compiler can be sold as muLISP Runtime Systems. Contact the Soft Warehouse, Inc. for licensing details.

```
035C   860    DB     0          ;Preserve D-code compiler flag
```

The following string is the muLISP read-eval-print loop prompt.

```
035D   861    DB     '$ ',0,0,0,0 ;muLISP prompt string
```

The following string is the default source file name extension recognized by the function LOAD.

```
0363   867    DB     'LSP'      ;Default source file name extension
```

If the following byte is nonzero and muLISP is running on an IBM PC or compatible computer, the cursor temporarily becomes a full block during garbage collections and data space reallocations.

```
0366   870    DB     -1         ;GC cursor indication flag
```

If the following byte is nonzero, input of a character whose high order bit is set is recognized as the first byte of a two-byte KANJI character.

```
0367   871    DB     0          ;Recognize KANJI characters flag
```

If the following byte is nonzero, the header message (see Chapter 2) is displayed when muLISP begins execution.

```
0368   872    DB     -1         ;Display logon header message flag
```

The following character is used by the functions PLOT-DOT, PLOT-LINE, and PLOT-CIRCLE for plotting points when in a text video mode (see Section 5.21).

```
0369   873    DB     '.'       ;Dot used for char-mode graphics
```

The following characters are used by the line-editor to indicate that more text is to the right and/or left of the displayed part of the line.

```
036A   874    DB     '<'       ;More left indicator character
036B   875    DB     '>'       ;More right indicator character
```

If the following byte is nonzero and muLISP is running on an IBM PC or compatible computer in a graphics video mode, muLISP uses its own internal character generator table to display characters 128 through 255. The table located from 400H to 7FFH can be modified by as desired.

```
036C   876    DB     -1        ;Use internal char generator flag
```

The following byte is used as a counter by the function TONE to determine the duration of the tone it produces. If the byte is zero, when muLISP starts up it uses the system clock to determine an appropriate value for the byte.

```
036D   877    DB     0         ;Timing counter byte
```

If one or more of the following words are nonzero, the size of the muLISP console window (see Section 5.21) is limited accordingly.

```
36F  036E   878    DW     0         ;Limit on base row
371  0370   880    DW     0         ;Limit on base column
373  0372   882    DW     0         ;Limit on rows
375  0374   884    DW     0         ;Limit on columns
```

The following byte is the default color used by the functions PLOT-DOT, PLOT-LINE, and PLOT-CIRCLE for plotting points.

```
0376   886    DB     15        ;Default plot color
```

Each entry of the following **Line Editor Control Key Table** points to the machine language routine called by the muLISP line editor when a control key is pressed. No action is taken if the pointer is 0.

```
0380  896  DW  0        ;CTRL-@
0382  898  DW  LFTWRD   ;CTRL-A
0384  900  DW  0        ;CTRL-B
0386  902  DW  0        ;CTRL-C
0388  904  DW  RHTCHR   ;CTRL-D
038A  906  DW  0        ;CTRL-E
038C  908  DW  RHTWRD   ;CTRL-F
038E  910  DW  DELCHR   ;CTRL-G
0390  912  DW  LFTCHR   ;CTRL-H or BACKSPACE
0392  914  DW  RHTTAB   ;CTRL-I or TAB
0394  916  DW  0        ;CTRL-J or LINEFEED
0396  918  DW  0        ;CTRL-K
0398  920  DW  0        ;CTRL-L
039A  922  DW  0        ;CTRL-M or RETURN
039C  924  DW  0        ;CTRL-N
039E  926  DW  0        ;CTRL-O
03A0  928  DW  ESCMOD   ;CTRL-P
03A2  930  DW  CMDCHD   ;CTRL-Q
03A4  932  DW  0        ;CTRL-R
03A6  934  DW  LFTCHR   ;CTRL-S
03A8  936  DW  DELWRD   ;CTRL-T
03AA  938  DW  UNDLIN   ;CTRL-U
03AC  940  DW  INSTOG   ;CTRL-V
03AE  942  DW  0        ;CTRL-W
03B0  944  DW  0        ;CTRL-X
03B2  946  DW  DELLIN   ;CTRL-Y
03B4  948  DW  0        ;CTRL-Z
03B6  950  DW  0        ;CTRL-[ or ESC
03B8  952  DW  0        ;CTRL-\
03BA  954  DW  0        ;CTRL-]
03BC  956  DW  0        ;CTRL-^
03BE  958  DW  DELLFT   ;CTRL-_
03C0  960  DW  DELLFT   ;RUBOUT or DELETE
```

Each entry of the following **Line Editor Extended Function Key Table** contains the second byte returned by an extended function key (see Appendix A) followed by a pointer to the machine language routine called by the muLISP line editor when the extended function key is pressed. The first word of the table contains the number of entries in the table. There is enough room in the table for ten extended function key routines.

```
03C2   962   DW   7          ;Entries in table
03C4   964   DB   83         ;Del
03C5   965   DW   DELCHR     ;Delete char under cursor
03C7   967   DB   75         ;<—
03C8   968   DW   LFTCHR     ;Move left a char
03CA   970   DB   77         ;—>
03CB   971   DW   RHTCHR     ;Move right a char
03CD   973   DB   115        ;Ctrl <—
03CE   974   DW   LFTWRD     ;Move left a word
03D0   976   DB   116        ;Ctrl <—
03D1   977   DW   RHTWRD     ;Move right a word
03D3   979   DB   61         ;F3
03D4   980   DW   UNDLIN     ;Undelete last deleted line
03D6   982   DB   82         ;Ins
03D7   983   DW   INSTOG     ;Insert/replace mode toggle
03D9   985   DB   0
03DA   986   DW   0          ;Free space
03DC   988   DB   0
03DD   989   DW   0
03DF   991   DB   0
03E0   992   DW   0
03E2   994   DB   0
03E3   995   DW   0
03E5   997   DB   0
03E6   998   DW   0
```

C. muLISP GLOBAL STORAGE AREA

Appendix C is an annotated assembly language listing of the muLISP global storage area. It shows the offset address of the variables that delimit the current muLISP data spaces and other system variables. The first two columns in each row of the listing are the hexadecimal and decimal offsets in the data segment of the variables.

```
                ORG     0           ;Object space variables
                BASSYM  EQU   100H  ;Base of symbols
0000    0       ENDSYM  DW    ?     ;End of symbols
0002    2       ENDNUM  DW    ?     ;End of numbers
0004    4       ENDATM  DW    ?     ;End of atom space
0006    6       ENDCNS  DW    ?     ;End of conses
0008    8       MAXCNS  DW    ?     ;Max size of cons space

000A   10               DW    ?     ;Reserved
000C   12               DW    ?     ;Reserved
000E   14               DW    ?     ;Reserved

                                    ;Segment pointer variables
0010   16       DEFSEG  DW    ?     ;Function/length cell seg
0012   18       PNPSEG  DW    ?     ;Print name/vector cell seg
0014   20       PNSSEG  DW    ?     ;Print name string segment
0016   22       VECSEG  DW    ?     ;Number vector segment

0018   24       USENAM  DW    ?     ;Reserved  Adresse du 1er symbole et l[e]
001A   26       ENDSYMGC DW   ?     ;Reserved  fin des symboles après tassement
001C   28       FIRSTSYMKILLED DW ? ;Reserved  1er symbole viré (ceux au [...] sont décalés)
001E   30               DW    ?     ;Reserved
0020   32       SPSAVE  DW    ?     ;Reserved  sauvegarde de SP

                                    ;Stack segment variables
                BASSTK  EQU   400H  ;Base of stack space
0022   34       BASCOD  DW    ?     ;End of stk/base of D-code
0024   36       ENDCOD  DW    ?     ;End of D-code
0026   38       MAXCOD  DW    ?     ;Maximum size of D-code
```

DS:


```
                                          ;Pname segment variables
0028  40    ENDSTR   DW   ?               ;End of print name strings
002A  42    MAXSTR   DW   ?               ;Max size of string space

                                          ;Vector segment variables
002C  44    ENDVEC   DW   ?               ;End of number vectors
002E  46    MAXVEC   DW   ?               ;Max size of vector space

                                          ;File control block variables
0030  48    BASFCB   DW   ?               ;Base of FCB space in CS
0032  50    IFCB     DW   ?               ;SIF FCB pointer
0034  52    OFCB     DW   ?               ;SOF FCB pointer
0036  54    ENDFCB   DW   ?               ;End of FCB space in CS

0038  56    THRVAL   DW   ?               ;Thrown value (0 = inactive)
003A  58    PRECSN   DW   ?               ;Current precision
003C  60    UNDFLO   DW   ?               ;Current underflow

003E  62    GCCTR    DW   ?               ;Garbage collection counter
0040  64    RACTR    DW   ?               ;Reallocation counter

0042  66    ORICS    DW   ?               ;Original code segment
```

44

46

48

4A ATOMFREE MINI

4C VECTORFREE MINI — devant provoquer un GC (?) — Valeurs de ATOMFREE, VectorFree, ConsFree après GC (voir en 81D9)

4E CONSFREE mini

5Ø temporaire : Sauvegarde de AX

52 Position dans table de gestion des erreurs numériques/routines

54

56 NIL si on saisit un entier, T si réel (.9) voir 5CB3 | signe par défaut

58 fraction 1

5A fraction 2

5C

5E longueur courante du Pname de travail

6Ø

62

D. SAMPLE ASSEMBLY LANGUAGE ROUTINE

Appendix D is an annotated assembly language listing of the function COPY that was discussed in Chapter 9.

```
        ASSUME CS:CSEG
CSEG    SEGMENT

ENDATM  EQU     04H             ;Offset of ENDATM in DS
CONS    EQU     0B27H           ;Offset of CONS in CS

                                ;FUNCTION: (COPY OBJ)
COPY:   JCXZ    RETNIL          ;Return NIL if no arguments
        MOV     SI,[BP]         ;SI: OBJ
COPYSI: CMP     SI,WORD PTR DS:[ENDATM]
        JB      COPY1           ;Jump if (ATOM OBJ)
        MOV     [BP],SI         ;Push OBJ on BP stack
        INC     BP
        INC     BP
        MOV     SI,[SI]         ;SI: (CAR OBJ)
        EVEN
        CALL    COPYSI          ;DI: (COPY (CAR OBJ))
        XCHG    DI,[BP-2]       ;Exchange OBJ with stack top
        MOV     SI,ES:[DI]      ;SI: (CDR OBJ)
        EVEN
        CALL    COPYSI          ;DI: (COPY (CDR OBJ))
        DEC     BP              ;Restore BP
        DEC     BP
        MOV     [BP+2],DI       ;Store DI on top of stack
        MOV     CX,4            ;Set CX to indicate 2 arguments
        MOV     AX,OFFSET CONS
        JMP     AX              ;DI: Cons copies of CAR & CDR

COPY1:  MOV     DI,SI           ;DI: OBJ
        RET

RETNIL: MOV     DI,100H         ;DI: NIL
        RET
CSEG    ENDS
        END     COPY
```