Volume 2, Issue 2 March 1980

OSI Items

OSI

Items

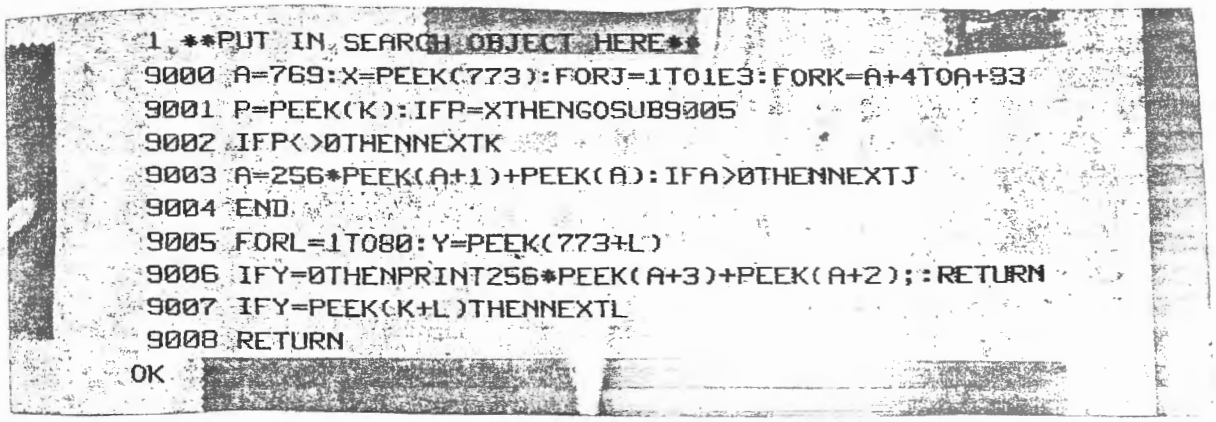
The C4P MF

Anouncement: Up to this issue, OSI-tems has been a collection of articles and software by just a small number of people. There are (too) many of you who just collect the (free) issues. All of you must have, at one time or another, written programs or aquired information that may be of interest to other people. This journal is in sore need of software, articles and applications. So if you wish this journal to continue, please drop off or send your material to this store.

Table of Contents

This issue prepared by: Mike Bassman
39-65 52 st. tel. (212) 639-3530
Woodside, NY 11377

FIND for OSI- Terry Terrence



```
1 **PUT IN SEARCH OBJECT HERE**
9000 A=769:X=PEEK(773):FORJ=1TO1E3:FORK=A+4TOA+93
9001 P=PEEK(K):IFP=XTHENGOSUB9005
9002 IFP<>0THENNEXTK
9003 A=256*PEEK(A+1)+PEEK(A):IFA>0THENNEXTJ
9004 END
9005 FORL=1TO80:Y=PEEK(773+L)
9006 IFY=0THENPRINT256*PEEK(A+3)+PEEK(A+2);:RETURN
9007 IFY=PEEK(K+L)THENNEXTL
9008 RETURN
OK
```

Total memory usage as written (no spaces)- 191 bytes.

Find will search for a variable, operator or a text expression in your program. To use Find, load it along with the program to be searched. Find can be anywhere within your program so long as there are no line number conflicts. Enter the expression to be searched for as line 1, for example:

1 IF

Now, evoke by RUN 9000. All line numbers containing that expression, including line 1 and lines within Find, will be printed on the screen. If the expression appears twice in one line it will be printed twice. While Find will search unambiguously for expressions which are stored as tokens, it cannot distinguish between variables and text expressions. If you searched for, for example:

1 S0

you would get any variable S0 or any time the letters S0 appeared as text even if they are imbedded in a word.

Comprint 912-serial printer review-Mike Bassman

The Comprint 912-S is a printer that I am currently using with my Challenger 1P. It uses electrosensitive 8½ in. wide paper and allows up to 80 characters per line. The paper is more expensive than plain paper and somewhat more difficult to obtain, but no ribbons, inks, daisy wheels or other consumables have to be bought. Characters are printed in a 9 by 12 dot matrix which provides true lower case descenders. There is no interfacing required or driving software; it plugs right into the RS232 serial (cassette) port. The major problem with the printer is with the paper it uses. The silvery

```
9003 A=256*PEEK(A+1)+PEEK(A):IFA>0THENNEXTJ
9004 END
9005 FORL=1TO80:Y=PEEK(773+L)
9006 IFY=0THENPRINT256*PEEK(A+3)+PEEK(A+2);:RETURN
9007 IFY=PEEK(K+L)THENNEXTL
9008 RETURN
OK
```

Total memory usage as written (no spaces)- 191 bytes.

Find will search for a variable, operator or a text expression in your program. To use Find, load it along with the program to be searched. Find can be anywhere within your program so long as there are no line number conflicts. Enter the expression to be searched for as line 1, for example:

1 IF

Now, evoke by RUN 9000. All line numbers containing that expression, including line 1 and lines within Find, will be printed on the screen. If the expression appears twice in one line it will be printed twice. While Find will search unambiguously for expressions which are stored as tokens, it cannot distinguish between variables and text expressions. If you searched for, for example:

1 SO

you would get any variable SO or any time the letters SO appeared as text even if they are imbedded in a word.

Comprint 912-serial printer review-Mike Bassman

The Comprint 912-S is a printer that I am currently using with my Challenger 1P. It uses electrosensitive 8½ in. wide paper and allows up to 80 characters per line. The paper is more expensive than plain paper and somewhat more difficult to obtain, but no ribbons, inks, daisy wheels or other consumables have to be bought. Characters are printed in a 9 by 12 dot matrix which provides true lower case descenders. There is no interfacing required or driving software; it plugs right into the RS232 serial (cassette) port. The major problem with the printer is with the paper it uses. The silvery appearance looks strange and no one likes the 'feel' of the paper, even though nothing rubs off. Apart from this, it has an impressive array of features, the most important being selectable baud rates with speeds up to 225 CPS, a built in self test, a pagination mode to skip lines after each 11" page, a ¼k buffer and a bell. For the reasonable ($699) price, the Comprint 912-S is an ideal printer for the OSI hobbyist.

Machine Language Clock- Salomon Lederman

This is a machine language program to simulate a clock. It allows you to enter your own delay loops for clock accuracy, but I have found that the most accurate setting is 803 for the main delay loop and 100 for the fine tuning. You must then enter the starting time on a rotating display on the screen. This program can be used as a stopwatch by entering the starting time as ØØ:ØØ ØØ. To start the program, hit the 'G' key.

Clock program

```
1 A(0)=53705
2 DATA53706,53708,53709,53711,53712,53705,53705,53709,53709,53711
3 FORN=1TO5:READA(N):NEXTN:FORN=1TO5:READB(N):NEXTN
10 DATA238,5,8,162,5,139,0,3,221,6
20 DATA8,208,14,169,48,157,0,8,189,255,7
30 DATA24,105,1,157,255,7,224,0,240,4
40 DATA202,76,5,9,173,1,8,201,51,208
50 DATA07,173,0,8,201,49,208,10,169
60 DATA49,141,1,8,169,48,141,0,8,162,5
70 DATA188,12,8,189,0,8,153,201,209
80 DATA202,224,255,208,242,174,254,7,172,255,7
90 DATA169,255,24,233,1,201,0,208,250,136,192,0,208,245,202,224,0,208
100 DATA240,76,0,9
110 FORQ=2304TO2406:READN:POKEQ,N:NEXTQ
116 DATA50,58,54,58,54,58
117 DATA0,1,3,4,6,7
118 FORQ=2054TO2065:READN:POKEQ,N:NEXTQ
119 PRINT:PRINT:PRINT:PRINT
120 PRINT"INPUT LENGTH OF DELAY":INPUTD
130 D1=INT(D/256):D2=D-256*D1
140 POKE2046,D1:POKE2047,D2
150 PRINT:PRINT"INPUT FINE TUNING":INPUTF:POKE2385,F
180 POKE2386,F
200 FORX=53200TO54200:POKEX,32:NEXTX
220 POKE53707,58
230 POKE11,0:POKE12,253:X=USR(X)
240 S=PEEK(531):IFS=71THEN300
250 IFS<48ORS>57THEN230
260 FORN=1TO5:T=PEEK(A(N)):POKEB(N),T:NEXTN
270 POKE53712,S:POKE53707,58:GOTO230
300 POKE11,75:POKE12,9
```

the fine tuning. You must then enter the starting time on a rotating display on the screen. This program can be used as a stopwatch by entering the starting time as ØØ:ØØ ØØ. To start the program, hit the 'G' key.

Clock program

```
1 A(0)=53705
2 DATA53706,53708,53709,53711,53712,53705,53705,53703,53709,53711
3 FORN=1TO5:READA(N):NEXTN:FORN=1TO5:READB(N):NEXTN
10 DATA238,5,8,162,5,139,0,3,221,6
20 DATA8,208,14,169,48,157,0,8,189,255,7
30 DATA24,105,1,157,255,7,224,0,240,4
40 DATA202,76,5,9,173,1,8,201,51,208
50 DATA07,173,0,8,201,49,208,10,169
60 DATA49,141,1,8,169,48,141,0,8,162,5
70 DATA188,12,8,189,0,8,153,201,209
80 DATA202,224,255,208,242,174,254,7,172,255,7
90 DATA169,255,24,233,1,201,0,208,250,136,192,0,208,245,202,224,0,208
100 DATA240,76,0,9
110 FORQ=2304TO2406:READN:POKEQ,N:NEXTQ
116 DATA50,58,54,58,54,53
117 DATA0,1,3,4,6,7
118 FORQ=2054TO2065:READN:POKEQ,N:NEXTQ
119 PRINT:PRINT:PRINT:PRINT
120 PRINT"INPUT LENGTH OF DELAY":INPUTD
130 D1=INT(D/256):D2=D-256*D1
140 POKE2046,D1:POKE2047,D2
150 PRINT:PRINT"INPUT FINE TUNING":INPUTF:POKE2385,F
180 POKE2386,F
200 FORX=53200TO54200:POKEX,32:NEXTX
220 POKE53707,58
230 POKE11,0:POKE12,253:X=USR(X)
240 S=PEEK(531):IFS=71THEN300
250 IFS<48ORS>57THEN230
260 FORN=1TO5:T=PEEK(A(N)):POKEB(N),T:NEXTN
270 POKE53712,S:POKE53707,58:GOTO230
300 POKE11,75:POKE12,9
310 FORN=0TO5:T=PEEK(A(N)):POKE2048+N,T:NEXTN
320 X=USR(X)
OK
```

HARDWARE REVIEWS by Mike Cohen

RADIO SHACK QUICK PRINTER II $219

If you're looking for a cheap printer to use for program listings, I recommend this one. It prints 32 characters per line on aluminum coated paper. The print quality is suitable for hobby use, but not good enough for most serious uses. The low price and quiet operation are the outstanding features which outweigh the disadvantages for hobby use. The paper is sold in most Radio Shack stores at 3.95 for two rolls.

One warning-----The printer runs at 600 baud only, so you will have to make some hardware modifications on your Challenger and do a few POKEs to use this printer.

OSI DISK DRIVE $499

This is just about the most useful peripheral you can buy. It will make your Challenger seem like one of the big computers. Program transfer is almost instantaneous and you can run a program automatically when it's loaded.

You can easily transfer your programs from tape to disk on a C1p since you keep the tape interface. To run these programs, you will have to change the pokes for control-C and the USR functions, but in most cases it should be easy.

The disk operating system takes up a large amount of memory, leaving only about 8k free in a 20K system. Disk BASIC has a 9 digit accuracy, although it is slightly slower than ROM BASIC. Most DOS commands can be executed from basic. In addition to all features of ROM basic, you get integer variables and several disk commands. The operating system gives several ways to select various devices for input, output and listings. The assembler and extended monitor are easily accessible, making machine language programming easy and pleasant.

If you can afford it, run out and buy a disk drive immediately. Your Challenger will never be the same!

Fast, Easy Machine Language Screen Clear.- Danny Schwartz

```
1Ø  POKE 11,34: POKE12,2
2Ø  FORK= 546 TO 571:I=PEEK (K+6449Ø):NEXT:POKE 572,96:X=USR(X)
```

Uses the screen clear already in ROM, then adds a 'RTS' to the end of it

If you're looking for a cheap printer to use for program listings, I recommend this one. It prints 32 characters per line on aluminum coated paper. The print quality is suitable for hobby use, but not good enough for most serious uses. The low price and quiet operation are the outstanding features which outweigh the disadvantages for hobby use. The paper is sold in most Radio Shack stores at 3.95 for two rolls.

One warning-----The printer runs at 600 baud only, so you will have to make some hardware modifications on your Challenger and do a few POKEs to use this printer.

OSI DISK DRIVE $499

This is just about the most useful peripheral you can buy. It will make your Challenger seem like one of the big computers. Program transfer is almost instantaneous and you can run a program automatically when it's loaded.

You can easily transfer your programs from tape to disk on a C1p since you keep the tape interface. To run these programs, you will have to change the pokes for control-C and the USR functions, but in most cases it should be easy.

The disk operating system takes up a large amount of memory, leaving only about 8k free in a 20K system. Disk BASIC has a 9 digit accuracy, although it is slightly slower than ROM BASIC. Most DOS commands can be executed from basic. In addition to all features of ROM basic, you get integer variables and several disk commands. The operating system gives several ways to select various devices for input, output and listings. The assembler and extended monitor are easily accessible, making machine language programming easy and pleasant.

If you can afford it, run out and buy a disk drive immediately. Your Challenger will never be the same!

Fast, Easy Machine Language Screen Clear.- Danny Schwartz

```
1Ø  POKE 11,34: POKE12,2
2Ø  FORK= 546 TO 571:I=PEEK (K+6449Ø):NEXT:POKE 572,96:X=USR(X)
```

Uses the screen clear already in ROM, then adds a 'RTS' to the end of it. Exists in the free part of page 2. No need to POKE in the program or to reserve memory for it. Works on C1P and should on the C2/C4.

Probability Machine.- Salomon Lederman

This program simulates (on the screen) a number of marbles rolling down, bouncing off posts and eventually falling into a slot. Probability would favor the marbles going into those slots closest to the middle. Eventually, you will see the bell curve form.

```
 1 REM**PROBABILITY MACHINE**SALOMON LEDERMAN
 5 FORX=1TO8:READB(X):NEXTX:A=27
 10 DATA128,144,150,154,158,159,160,161
 15 FORX=53200TO54200:POKEX,32:NEXTX
 20 FORJ=0TO9:FORK=0TO2*JSTEP2
 30 POKE53411+32*J+(13-J)+K,A
 40 NEXTK:NEXTJ
 50 FORX=53422TO53701STEP31
 60 POKEX-1,176:POKEX,175:NEXTX
 80 FORX=53426TO53723STEP33:POKEX,177
 100 POKEX+1,178:NEXTX
 110 FORJ=53735TO54119STEP32
 112 FORK=0TO18STEP2
 115 POKEJ+K,149:NEXTK:NEXTJ
 120 FORX=53733TO54117STEP32:POKEX,161
 130 POKEX+22,161:NEXTX
 140 POKE53700,32:POKE53701,161
 150 POKE53723,161:POKE53724,32
 160 FORX=54117TO54139:POKEX,163:NEXTX
 170 POKE53390,161:POKE53391,156:POKE53393,157:POKE53394,161
 180 P=53392:
 190 POKEP,111:T=0
 200 R=RND(86):IFR<.5THENT=2
 210 IFPEEK(P+31+T)=32THENPOKEP,32:P=P+31+T:GOTO190
 220 IFPEEK(P+32)=32THENPOKEP,32:P=P+32:GOTO190
 222 IFPEEK(P+32)=163THENPOKEP,B(1):GOTO190
 225 FORX=1TO8:IFB(X)<>PEEK(P+32)THENNEXTX
 230 POKEP,32:IFX=8THEN250
 240 POKEP+32,B(X+1):GOTO190
 250 POKEP,B(1):GOTO190
OK
```

Multiplication.- Salomon Lederman

Multiplication is a program which will show all the steps involved in a multiplication problem. Enter a ten digit number on the rotating display, hit '*', enter another ten digit number, then press '='.

```
 1 REM**MULTIPLY**SALOMON LEDERMAN
 5 S=53499:Z=0:Z1=42:POKE11,0:POKE12,253:FORX=53200TO54200:POKEX,32
 10 NEXTX:GOSUB20:Z=32:Z1=61:GOSUB20:GOTO50
 20 X=USR(X):N=PEEK(531):IFN=Z1THENRETURN
 30 IFN<48ORN>57THEN20
 40 FORX=S-41+ZTOS-33+Z:Q=PEEK(X+1):POKEX,Q:NEXTX:POKE53467+Z,N:GOTO20
 50 POKE53488,120:FORX=53520TO53531:POKEX,131:NEXTX
 62 FORX=53458TO53467:IFPEEK(X)=32THENPOKEX,48
```

tually, you will see the bell curve form.

```
 1 REM**PROBABILITY MACHINE**SALOMON LEDERMAN
 5 FORX=1TO8:READB(X):NEXTX:A=27
 10 DATA128,144,150,154,158,159,160,161
 15 FORX=53200TO54200:POKEX,32:NEXTX
 20 FORJ=0TO9:FORK=0TO2*JSTEP2
 30 POKE53411+32*J+(13-J)+K,A
 40 NEXTK:NEXTJ
 50 FORX=53422TO53701STEP31
 60 POKEX-1,176:POKEX,175:NEXTX
 80 FORX=53426TO53723STEP33:POKEX,177
 100 POKEX+1,178:NEXTX
 110 FORJ=53735TO54119STEP32
 112 FORK=0TO18STEP2
 115 POKEJ+K,149:NEXTK:NEXTJ
 120 FORX=53733TO54117STEP32:POKEX,161
 130 POKEX+22,161:NEXTX
 140 POKE53700,32:POKE53701,161
 150 POKE53723,161:POKE53724,32
 160 FORX=54117TO54139:POKEX,163:NEXTX
 170 POKE53390,161:POKE53391,156:POKE53393,157:POKE53394,161
 180 P=53392:
 190 POKEP,111:T=0
 200 R=RND(86):IFR<.5THENT=2
 210 IFPEEK(P+31+T)=32THENPOKEP,32:P=P+31+T:GOTO190
 220 IFPEEK(P+32)=32THENPOKEP,32:P=P+32:GOTO190
 222 IFPEEK(P+32)=163THENPOKEP,B(1):GOTO180
 225 FORX=1TO8:IFB(X)<>PEEK(P+32)THENNEXTX
 230 POKEP,32:IFX=8THEN250
 240 POKEP+32,B(X+1):GOTO130
 250 POKEP,B(1):GOTO180
OK
```

Multiplication.- Salomon Lederman

Multiplication is a program which will show all the steps involved in a multiplication problem. Enter a ten digit number on the rotating display, hit '*', enter another ten digit number, then press '='.

```
1 REM**MULTIPLY**SALOMON LEDERMAN
5 S=53499:Z=0:Z1=42:POKE11,0:POKE12,253:FORX=53200TO54200:POKEX,32
10 NEXTX:GOSUB20:Z=32:Z1=61:GOSUB20:GOTO50
20 X=USR(X):N=PEEK(531):IFN=Z1THENRETURN
30 IFN<48ORN>57THEN20
40 FORX=S-41+ZTOS-33+Z:Q=PEEK(X+1):POKEX,Q:NEXTX:POKE53467+Z,N:GOTO20
50 POKE53488,120:FORX=53520TO53531:POKEX,131:NEXTX
62 FORX=53458TO53467:IFPEEK(X)=32THENPOKEX,48
64 IFPEEK(X+32)=32THENPOKEX+32,48
66 NEXTX:FORK=0TO9:M=0:FORJ=0TO9:L=PEEK(S-K)-48:L1=PEEK(S-32-J)-48
100 L1=PEEK(S-32-J)-48
110 L=L1*L+M:M1=10*(L/10-INT(L/10)):M=INT(.5+(L-M1)/10)
120 POKE(53499-K)+(2+K)*32-J,48+INT(M1+.5)
130 NEXTJ:POKE(53499-K)+(2+K)*32-10,48+M
140 NEXTK:FORX=53864TO53883:POKEX,131:NEXTX:A=0:W=0
160 FORJ=53563TO53544STEP-1:FORK=0TO9
170 IFPEEK(J+32*K)<>32THENA=A+PEEK(J+32*K)-48
```

Extended Cassette Manipulation for the C1P.- Mike Bassman.

The OSI C1P has a very useful, if little mentioned feature. This is the ability to control the cassette port. Ohio Scientific includes, in the users manual, a very short section on cassette data files. The first thing to do is to ignore it. The information is at best, inaccurate.

The location of the cassette port is 61440 and 61441 (decimal). Location 61440 is a status flag for system readiness. When the C1P is ready to accept or send out a character the condition 'PEEK (61440)AND 1 ' is true. Location 61441 serves two functions;input and output. Thus, when you have detected that the system is ready, 61441 will contain the ASCII value of the character being read in, or if you are outputting, you then POKE the ASCII value into 61441. Program A shows these principles. What the program does is to constantly check the cassette port for characters. When a character comes in, it is printed. This routine is actually useful because you can use it to view a program as it comes in without actually putting it into memory.

PROGRAM A

```
10 IFPEEK(61440)AND1 THEN 30:REM SYSTEM READY?
20 GOTO10:REM IF NOT,GO BACK
30 PRINTCHR$(PEEK(61441));:GOTO10:REM PRINT INCOMING CHARACTER
OK
```

Cassette data files are done by a totally different method. The way to use a cassette file is to simulate a SAVE or a LOAD, but rather than saving or loading a program you save or load strings, numbers, data or whatever you wish to put in the file. There is one problem with cassette data files. Not being as reliable as disk, they tend to accumulate garbage characters on the tape. The way to get around this is to save several 'garbage' strings before the actual data file. When you are loading the data file, input a numeric variable from the tape. When you are trying to input a number, the C1P will not accept the garbage strings on the tape. Simulating a SAVE or a load within the program can be done as follows:

SAVE on (output characters to recorder and screen)	--	POKE 517,1
SAVE off(transfer output to screen only)	--	POKE 517,Ø
LOAD on (do inputs from the recorder only)	--	POKE 515,255
LOAD off(transfer inputs back to the keyboard)	--	POKE 515,0

Program B is an example of datafile techniques. What it does is to either collect strings and save them on tape, or to load strings

in the users manual, a very short section on cassette data files. The first thing to do is to ignore it. The information is at best, inaccurate.

The location of the cassette port is 61440 and 61441 (decimal). Location 61440 is a status flag for system readiness. When the ClP is ready to accept or send out a character the condition 'PEEK (61440)AND 1 ' is true. Location 61441 serves two functions:input and output. Thus, when you have detected that the system is ready, 61441 will contain the ASCII value of the character being read in, or if you are outputting, you then POKE the ASCII value into 61441. Program A shows these principles. What the program does is to constantly check the cassette port for characters. When a character comes in, it is printed. This routine is actually useful because you can use it to view a program as it comes in without actually putting it into memory.

PROGRAM A

```
10 IFPEEK(61440)AND1 THEN 30:REM SYSTEM READY?
20 GOTO10:REM IF NOT,GO BACK
30 PRINTCHR$(PEEK(61441));:GOTO10:REM PRINT INCOMING CHARACTER
OK
```

Cassette data files are done by a totally different method. The way to use a cassette file is to simulate a SAVE or a LOAD, but rather than saving or loading a program you save or load strings, numbers, data or whatever you wish to put in the file. There is one problem with cassette data files. Not being as reliable as disk, they tend to accumulate garbage characters on the tape. The way to get around this is to save several 'garbage' strings before the actual data file. When you are loading the data file, input a numeric variable from the tape. When you are trying to input a number, the ClP will not accept the garbage strings on the tape. Simulating a SAVE or a load within the program can be done as follows:

SAVE on (output characters to recorder and screen) -- POKE 517,1
SAVE off(transfer output to screen only) -- POKE 517,Ø
LOAD on (do inputs from the recorder only) -- POKE 515,255
LOAD off(transfer inputs back to the keyboard) -- POKE 515,0

Program B is an example of datafile techniques. What it does is to either collect strings and save them on tape, or to load strings from tape.

Program B on next page. ------)

continued from page 7.

PROGRAM B

```
 10 FORK=53248TO54272:POKEK,32:NEXT
 20 PRINT"1 Get and save strings":PRINT"2 load strings from tape"
 30 INPUTC:IFC<1ORC>2THEN20
 40 IFC=2THEN150
 50 INPUT"How many strings do you wish to input";NS
 60 FORK=1TONS:PRINT"String";K;INPUTA$(K):NEXT
 70 PRINT"Push record then hit    space & return"
 80 INPUTB$:POKE517,1
 90 FORK=1TO3:PRINT"Garbage string":NEXT
 100 PRINTNS:FORK=1TONS:PRINTA$(NS):NEXT
 110 POKE517,0:PRINT"Push stop on recorder":RUN
 150 PRINT"hit play on recorder    then type space & return"
 160 INPUTB$:POKE515,255
 170 INPUTNS:FORK=1TONS:INPUTA$(K):NEXT
 180 POKE515,0:PRINT"Hit stop on recorder"
 190 PRINT"These were the strings":PRINT
 200 FORK=1TONS:PRINTA$(K):NEXT:PRINT:END
OK
```

Explanation of Program B

Line 10 clears the screen.

Lines 20,30 input your selection of features.

Lines 50,60 input how many strings you wish to enter and enters them.

Lines 70,80 initialize beginning of save procedure.

Line 90 outputs the garbage strings.

Line 100 outputs to the recorder the strings you have entered.

Line 110 ends the save procedure& ends the program.

Lines 150,160 initializes beginning of load procedure.

Line 170 gets (from tape) number of strings, then the actual strings.

Line 180 ends the load procedure.

Line 190 prints out strings loaded from tape & ends the program.

Using these ideas, OSI C1P owners can now do more useful programs involving data files, some possibilities being a computerized address book and maybe even a mailing list program.

****

The 'Silicon Gulch Gazette' is now available, free of charge, on request from:

```
10 FORK=53248TO54272:POKEK,32:NEXT
20 PRINT"1 Get and save strings":PRINT"2 load strings from tape"
30 INPUTC:IFC<1ORC>2THEN20
40 IFC=2THEN150
50 INPUT"How many strings do you wish to input";NS
60 FORK=1TONS:PRINT"String";K;INPUTA$(K):NEXT
70 PRINT"Push record then hit    space & return"
80 INPUTB$:POKE517,1
90 FORK=1TO3:PRINT"Garbage string":NEXT
100 PRINTNS:FORK=1TONS:PRINTA$(NS):NEXT
110 POKE517,0:PRINT"Push stop on recorder":RUN
150 PRINT"hit play on recorder    then type space & return"
160 INPUTB$:POKE515,255
170 INPUTNS:FORK=1TONS:INPUTA$(K):NEXT
180 POKE515,0:PRINT"Hit stop on recorder"
190 PRINT"These were the strings":PRINT
200 FORK=1TONS:PRINTA$(K):NEXT:PRINT:END
OK
```

Explanation of Program B

Line 10 clears the screen.

Lines 20,30 input your selection of features.

Lines 50,60 input how many strings you wish to enter and enters them.

Lines 70,80 initialize beginning of save procedure.

Line 90 outputs the garbage strings.

Line 100 outputs to the recorder the strings you have entered.

Line 110 ends the save procedure& ends the program.

Lines 150,160 initializes beginning of load procedure.

Line 170 gets (from tape) number of strings, then the actual strings.

Line 180 ends the load procedure.

Line 190 prints out strings loaded from tape & ends the program.

Using these ideas, OSI C1P owners can now do more useful programs involving data files, some possibilities being a computerized address book and maybe even a mailing list program.

****

The 'Silicon Gulch Gazette' is now available, free of charge, on request from:

Computer Faire
333 Swett Road
Woodside, CA 94062

9 - DIGIT BASIC VARIABLES

This article shall attempt to explain how BASIC stores and accesses variables. There are three types of variables in BASIC. They are floating point, integer, and string variables. The three types of variables may be non-subscripted, i.e. simple variables or subscripted variables.

VARIABLE STORAGE

Diagram one shows where in memory variables are stored. The BASIC program starts at the address pointed to by TXTTAB. Simple variables start just after the BASIC program. Immediately following simple variables in memory are array variables. Following the array variables in memory is the free memory space. After the free memory the actual string data is stored.

VARIABLE DESCRIPTORS

BASIC must have a way of "keeping track" of all variables. In order to do this, BASIC retains a descriptor for each variable in the program. A descriptor contains, in the case of numeric variables, the current value of that variable. String descriptors contain information on the length of the string and it's location in memory.

Upon encountering a variable for the first time, BASIC constructs a descriptor for that variable. If the variable is a subscripted variable, BASIC constructs an array descriptor. New simple variables are tacked on to the end of the other simple variables. Insertion of a simple variable requires that the array variables first be moved to make room. Subscripted variables are "tacked" on to the end of existing subscripted variables. Adding a subscripted variable decreases the amount of free memory space. Because array variables must be shifted upward to accommodate new simple variables, simple variables should be defined early in the program. DIM statements, when encountered, force BASIC to set up an array descriptor large enough to accommodate the size given in the DIM statement.

SIMPLE FLOATING POINT DESCRIPTORS

Simple floating point numeric variables have a descriptor containing the variable name and it's value. Diagram 2A shows the descriptor. Two bytes are always reserved for the variable name regardless of it's length in the program.

Floating point numbers are always stored in memory in a "NORMALIZED" form. "NORMALIZED" means that the binary number is shifted to the left until the most significant bit (MSB) is a one. The fractional value's sign bit is then placed into the MSB of the fractional value.

The MSB of the fractional value is implied to be a one unless the number is zero. A variable whose value is zero has it's exponent set to zero. In all cases, the binary point is implied to be to the left of the MSB in the fractional value.

SIMPLE INTEGER VARIABLE DESCRIPTORS

Diagram 2B shows the format of a simple integer

VARIABLE STORAGE

Diagram one shows where in memory variables are stored. The BASIC program starts at the address pointed to by TXTTAB. Simple variables start just after the BASIC program. Immediately following simple variables in memory are array variables. Following the array variables in memory is the free memory space. After the free memory the actual string data is stored.

VARIABLE DESCRIPTORS

BASIC must have a way of "keeping track" of all variables. In order to do this, BASIC retains a descriptor for each variable in the program. A descriptor contains, in the case of numeric variables, the current value of that variable. String descriptors contain information on the length of the string and it's location in memory.

Upon encountering a variable for the first time, BASIC constructs a descriptor for that variable. If the variable is a subscripted variable, BASIC constructs an array descriptor. New simple variables are tacked on to the end of the other simple variables. Insertion of a simple variable requires that the array variables first be moved to make room. Subscripted variables are "tacked" on to the end of existing subscripted variables. Adding a subscripted variable decreases the amount of free memory space. Because array variables must be shifted upward to accommodate new simple variables, simple variables should be defined early in the program. DIM statements, when encountered, force BASIC to set up an array descriptor large enough to accommodate the size given in the DIM statement.

SIMPLE FLOATING POINT DESCRIPTORS

Simple floating point numeric variables have a descriptor containing the variable name and it's value. Diagram 2A shows the descriptor. Two bytes are always reserved for the variable name regardless of it's length in the program.

Floating point numbers are always stored in memory in a "NORMALIZED" form. "NORMALIZED" means that the binary number is shifted to the left until the most significant bit (MSB) is a one. The fractional value's sign bit is then placed into the MSB of the fractional value.

The MSB of the fractional value is implied to be a one unless the number is zero. A variable whose value is zero has it's exponent set to zero. In all cases, the binary point is implied to be to the left of the MSB in the fractional value.

SIMPLE INTEGER VARIABLE DESCRIPTORS

Diagram 2B shows the format of a simple integer descriptor. As with floating point variables, two bytes are always allocated for the name. BASIC requires a method by which it may differentiate between floating point, integer and string variables. The method used is a simple one involving the variable name. Floating point names are not changed, however string and integer names are affected. Integer names always have $8Ø added to the two bytes reserved for it's name. String variables have $8Ø added to the second byte reserved for the name. Integers are stored in two's complement form with bit 7 of the most significant byte reflecting the sign. Integers are stored in two bytes with bit

Diagram one shows where in memory variables are stored. The BASIC program starts at the address pointed to by TXTTAB. Simple variables start just after the BASIC program. Immediately following simple variables in memory are array variables. Following the array variables in memory is the free memory space. After the free memory the actual string data is stored.

VARIABLE DESCRIPTORS

BASIC must have a way of "keeping track" of all variables. In order to do this, BASIC retains a descriptor for each variable in the program. A descriptor contains, in the case of numeric variables, the current value of that variable. String descriptors contain information on the length of the string and it's location in memory.

Upon encountering a variable for the first time, BASIC constructs a descriptor for that variable. If the variable is a subscripted variable, BASIC constructs an array descriptor. New simple variables are tacked on to the end of the other simple variables. Insertion of a simple variable requires that the array variables first be moved to make room. Subscripted variables are "tacked" on to the end of existing subscripted variables. Adding a subscripted variable decreases the amount of free memory space. Because array variables must be shifted upward to accommodate new simple variables, simple variables should be defined early in the program. DIM statements, when encountered, force BASIC to set up an array descriptor large enough to accommodate the size given in the DIM statement.

SIMPLE FLOATING POINT DESCRIPTORS

Simple floating point numeric variables have a descriptor containing the variable name and it's value. Diagram 2A shows the descriptor. Two bytes are always reserved for the variable name regardless of it's length in the program.

Floating point numbers are always stored in memory in a "NORMALIZED" form. "NORMALIZED" means that the binary number is shifted to the left until the most significant bit (MSB) is a one. The fractional value's sign bit is then placed into the MSB of the fractional value.

The MSB of the fractional value is implied to be a one unless the number is zero. A variable whose value is zero has it's exponent set to zero. In all cases, the binary point is implied to be to the left of the MSB in the fractional value.

SIMPLE INTEGER VARIABLE DESCRIPTORS

Diagram 2B shows the format of a simple integer descriptor. As with floating point variables, two bytes are always allocated for the name. BASIC requires a method by which it may differentiate between floating point, integer and string variables. The method used is a simple one involving the variable name. Floating point names are not changed, however string and integer names are affected. Integer names always have $8Ø added to the two bytes reserved for it's name. String variables have $8Ø added to the second byte reserved for the name. Integers are stored in two's complement form with bit 7 of the most significant byte reflecting the sign. Integers are stored in two bytes with 15 bits reserved for the value. This limits the range of an integer variable to be between -32767 and +32767.

9 - DIGIT BASIC VARIABLES

This article shall attempt to explain how BASIC stores and accesses variables. There are three types of variables in BASIC. They are floating point, integer, and string variables. The three types of variables may be non-subscripted, i.e. simple variables or subscripted variables.

VARIABLE STORAGE

Diagram one shows where in memory variables are stored. The BASIC program starts at the address pointed to by TXTTAB. Simple variables start just after the BASIC program. Immediately following simple variables in memory are array variables. Following the array variables in memory is the free memory space. After the free memory the actual string data is stored.

VARIABLE DESCRIPTORS

BASIC must have a way of "keeping track" of all variables. In order to do this, BASIC retains a descriptor for each variable in the program. A descriptor contains, in the case of numeric variables, the current value of that variable. String descriptors contain information on the length of the string and it's location in memory.

Upon encountering a variable for the first time, BASIC constructs a descriptor for that variable. If the variable is a subscripted variable, BASIC constructs an array descriptor. New simple variables are tacked on to the end of the other simple variables. Insertion of a simple variable requires that the array variables first be moved to make room. Subscripted variables are "tacked" on to the end of existing subscripted variables. Adding a subscripted variable decreases the amount of free memory space. Because array variables must be shifted upward to accommodate new simple variables, simple variables should be defined early in the program. DIM statements, when encountered, force BASIC to set up an array descriptor large enough to accommodate the size given in the DIM statement.

SIMPLE FLOATING POINT DESCRIPTORS

Simple floating point numeric variables have a descriptor containing the variable name and it's value. Diagram 2A shows the descriptor. Two bytes are always reserved for the variable name regardless of it's length in the program.

Floating point numbers are always stored in memory in a "NORMALIZED" form. "NORMALIZED" means that the binary number is shifted to the left until the most significant bit (MSB) is a one. The fractional value's sign bit is then placed into the MSB of the fractional value.

The MSB of the fractional value is implied to be a one unless the number is zero. A variable whose value is zero has it's exponent set to zero. In all cases, the binary point is implied to be to the left of the MSB in the fractional value.

Diagram one shows where in memory variables are stored. The BASIC program starts at the address pointed to by TXTTAB. Simple variables start just after the BASIC program. Immediately following simple variables in memory are array variables. Following the array variables in memory is the free memory space. After the free memory the actual string data is stored.

VARIABLE DESCRIPTORS

BASIC must have a way of "keeping track" of all variables. In order to do this, BASIC retains a descriptor for each variable in the program. A descriptor contains, in the case of numeric variables, the current value of that variable. String descriptors contain information on the length of the string and it's location in memory.

Upon encountering a variable for the first time, BASIC constructs a descriptor for that variable. If the variable is a subscripted variable, BASIC constructs an array descriptor. New simple variables are tacked on to the end of the other simple variables. Insertion of a simple variable requires that the array variables first be moved to make room. Subscripted variables are "tacked" on to the end of existing subscripted variables. Adding a subscripted variable decreases the amount of free memory space. Because array variables must be shifted upward to accommodate new simple variables, simple variables should be defined early in the program. DIM statements, when encountered, force BASIC to set up an array descriptor large enough to accommodate the size given in the DIM statement.

SIMPLE FLOATING POINT DESCRIPTORS

Simple floating point numeric variables have a descriptor containing the variable name and it's value. Diagram 2A shows the descriptor. Two bytes are always reserved for the variable name regardless of it's length in the program.

Floating point numbers are always stored in memory in a "NORMALIZED" form. "NORMALIZED" means that the binary number is shifted to the left until the most significant bit (MSB) is a one. The fractional value's sign bit is then placed into the MSB of the fractional value.

The MSB of the fractional value is implied to be a one unless the number is zero. A variable whose value is zero has it's exponent set to zero. In all cases, the binary point is implied to be to the left of the MSB in the fractional value.

SIMPLE INTEGER VARIABLE DESCRIPTORS

Diagram 2B shows the format of a simple integer descriptor. As with floating point variables, two bytes are always allocated for the name. BASIC requires a method by which it may differentiate between floating point, integer and string variables. The method used is a simple one involving the variable name. Floating point names are not changed, however string and integer names are affected. Integer names always have $8Ø added to the two bytes reserved for it's name. String variables have $8Ø added to the second byte reserved for the name. Integers are stored in two's complement form with bit 7 of the most significant byte reflecting the sign. Integers are stored in two bytes with 15 bits reserved for the value. This limits the range of an integer variable to be between -32767 and +32767.

SIMPLE STRING VARIABLE DESCRIPTORS

Diagram 2C illustrates the format used for simple string descriptors. A string variable always has $8Ø added to the second byte reserved for the name. String descriptors, unlike numeric variable descriptors, do not contain the value of the variable. A string descriptor contains the length of the string and a pointer to the actual string data. Strings maybe located within the actual program text or within the string space stored at the top of memory. The location of the string is dependent on how the string was defined. If a string variable is equated to string data within quotes, then that string's descriptor would point into the program text. A string variable equated to a CHR$, LEFT$, RIGHT$, or MID$ function will create a string in string space. INPUT statements will also cause a string to be formed in string space. If one string variable is equated to another, that string's pointer may or may not point into string space. Where the string descriptor points to is dependent on where the string variable to the right of the equal sign is located. If it is located within the program, then both descriptors will point into the work space. If the string that is being equated to resides in string space, then that string will be copied to the bottom of free space and the second string's descriptor will point to the copy.

ARRAY VARIABLE DESCRIPTORS

Diagram 3 illustrates the format of array variable descriptors. The array descriptors are basically the same as simple variable descriptors. The differences are information that defines the length of the array, the number of subscripts and the maximum value for each subscript, i.e. DIM A$ (2Ø,1Ø) would set the maximum subscripts to be 2Ø and 1Ø respectively. If an array variable is encountered that has not been DIMensioned, BASIC will default to a DIMension of 1Ø. In other words the array will be set up as if a DIMension of 1Ø had been executed. It is important to note that DIM statements must be executed to be effective. In memory, arrays are stored as sequential lists. The formula for accessing a array entry is:

For a List Array A$ (N):

Address of descriptor = N * Length of descriptor + Starting Array address

e.g., N=5Ø: Array data starts at $6ØØA

Address = (5Ø*3) + $6ØØA = $96 + $6ØØA = $6ØAØ

Multiple subscripted variables are slightly more involved. Diagram 4 shows a string array in memory. The array has been DIMensioned to 2 by 3. The formula for calculating the offset into an array to access a specific variable is shown below.

Offset = Length of descriptor * (S1+((S1max+1)*S2))

Working through an example should clarify the procedure. The example will use the array A$ (X,Y). The array has been DIMensioned as DIM A$ (2,3). The descriptor length will equal 3, one byte for the string length plus two bytes for the pointer to the actual string data. Since BASIC permits use of a subscript of zero, there are actually 12 variables in the array. The legal subscripts range

point into the program text. A string variable equated to a CHR$, LEFT$, RIGHT$, or MID$ function will create a string in string space. INPUT statements will also cause a string to be formed in string space. If one string variable is equated to another, that string's pointer may or may not point into string space. Where the string descriptor points to is dependent on where the string variable to the right of the equal sign is located. If it is located within the program, then both descriptors will point into the work space. If the string that is being equated to resides in string space, then that string will be copied to the bottom of free space and the second string's descriptor will point to the copy.

ARRAY VARIABLE DESCRIPTORS

Diagram 3 illustrates the format of array variable descriptors. The array descriptors are basically the same as simple variable descriptors. The differences are information that defines the length of the array, the number of subscripts and the maximum value for each subscript, i.e. DIM A$ (2Ø,1Ø) would set the maximum subscripts to be 2Ø and 1Ø respectively. If an array variable is encountered that has not been DIMensioned, BASIC will default to a DIMension of 1Ø. In other words the array will be set up as if a DIMension of 1Ø had been executed. It is important to note that DIM statements must be executed to be effective. In memory, arrays are stored as sequential lists. The formula for accessing a array entry is:

For a List Array A$ (N):

Address of descriptor = N * Length of descriptor + Starting Array address

e.g., N=5Ø: Array data starts at $6ØØA

Address = (5Ø*3) + $6ØØA = $96 + $6ØØA = $6ØAØ

Multiple subscripted variables are slightly more involved. Diagram 4 shows a string array in memory. The array has been DIMensioned to 2 by 3. The formula for calculating the offset into an array to access a specific variable is shown below.

Offset = Length of descriptor * (S1+((S1max+1)*S2))

Working through an example should clarify the procedure. The example will use the array A$ (X,Y). The array has been DIMensioned as DIM A$ (2,3). The descriptor length will equal 3, one byte for the string length plus two bytes for the pointer to the actual string data. Since BASIC permits use of a subscript of zero, there are actually 12 variables in the array. The legal subscripts range from A$ (Ø,Ø) to A$ (2,3). We shall refer to the first subscript as S1 and to the second subscript as S2. The maximum subscript for S1 and S2, as defined in the DIMension statement is 2 and 3 respectively. The array is organized with A$ (Ø,Ø) at the front and A$ (2,3) as the last entry. If subscript S1 increases by one the offset increases by the length of one descriptor i.e. by 3 bytes. If S2 increases by one the offset would increase by (S1max+1)*3. This may seem a little strange, so let's examine the array format a little closer. The first variable in the array is A$(Ø,Ø). The next variable is A$ (1,Ø). In other words the array is in the order A$ (Ø,Ø), A$(1,Ø), A$(2,Ø), A$(Ø,1), A$ (1,1), etc.

This then indicates that when S1 increases by one, we are moving forward by one descriptor. However, if S2 increases by one, we must move past the descriptors containing the values for A$ (Ø,S2) through A$ (S1max,S2).

Using the values in our example, the relative offset to A$ (2,3) would be:

Relative offset = descriptor length *(S1+((S1max+1)*S2))

Relative offset = 3*(2+((2+1)*3))

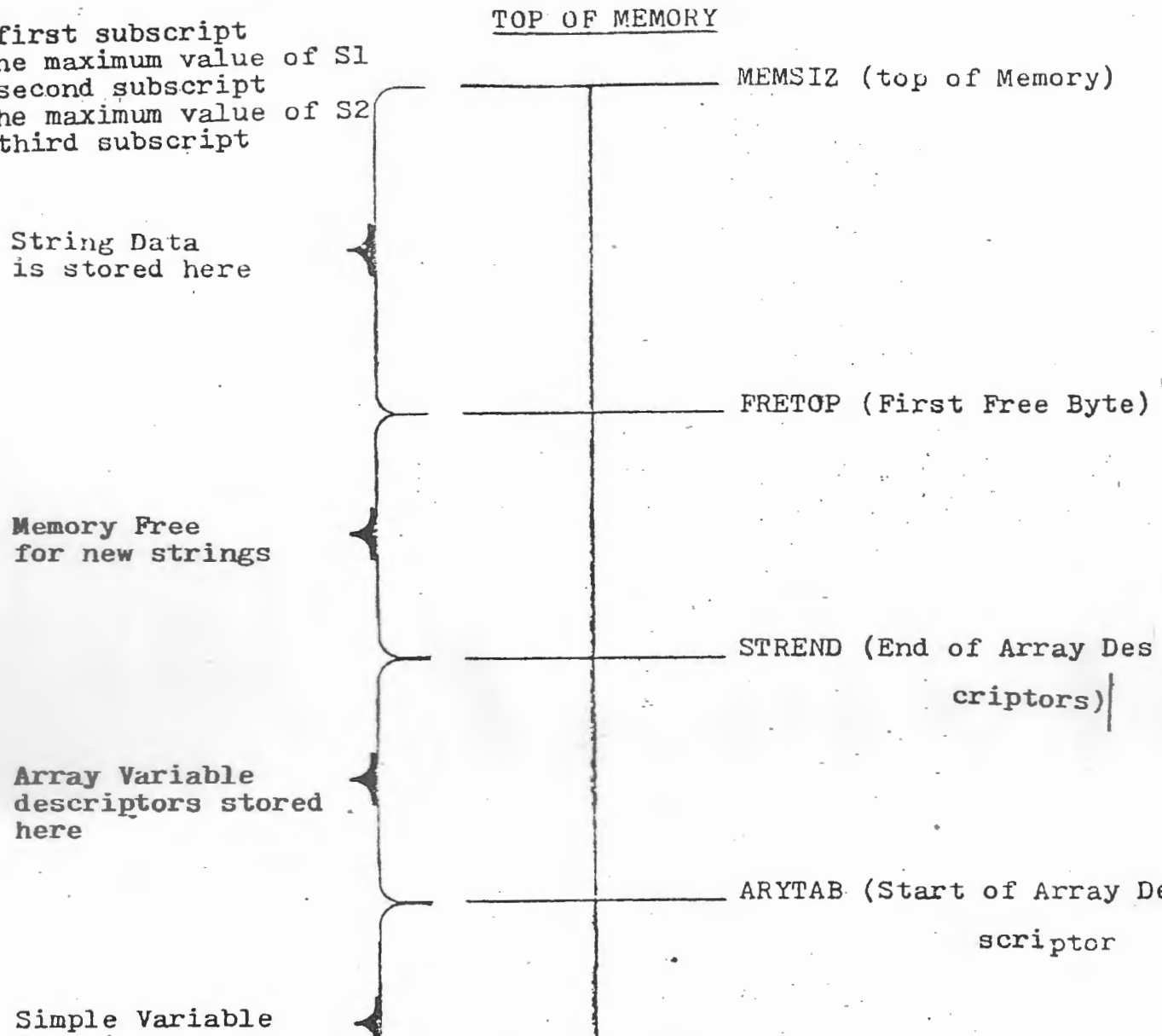
Relative offset = 3*(2+9) = 3*11 = 33

Therefore, if one listed to access A$ (2,3) one would set up a pointer to the start of the array descriptors. Then the relative offset would be added to the array pointer. Next one would pick up the length of the string and the two byte pointer to the actual string data. At this point we have all the information required to access the string data of A$ (2,3). Variable descriptors for array variables with more than two subscripts follow the same BASIC format. A generalized form for accessing a descriptor of a variable having N subscripts is shown below.

Relative offset = DL (S1+(S1max+1)*S2+(S2max+1)*S3+.....)

Where DL is the length of the descriptor

S1 is the first subscript
S1max is the maximum value of S1
S2 is the second subscript
S2max is the maximum value of S2
S3 is the third subscript
etc.
etc.



up the length of the string and the two byte pointer to the actual string data. At this point we have all the information required to access the string data of A$ (2,3). Variable descriptors for array variables with more than two subscripts follow the same BASIC format. A generalized form for accessing a descriptor of a variable having N subscripts is shown below.

Relative offset = DL (S1+(S1max+1)*S2+(S2max+1)*S3+.....)

Where DL is the length of the descriptor

S1 is the first subscript
S1max is the maximum value of S1
S2 is the second subscript
S2max is the maximum value of S2
S3 is the third subscript
etc.
etc.

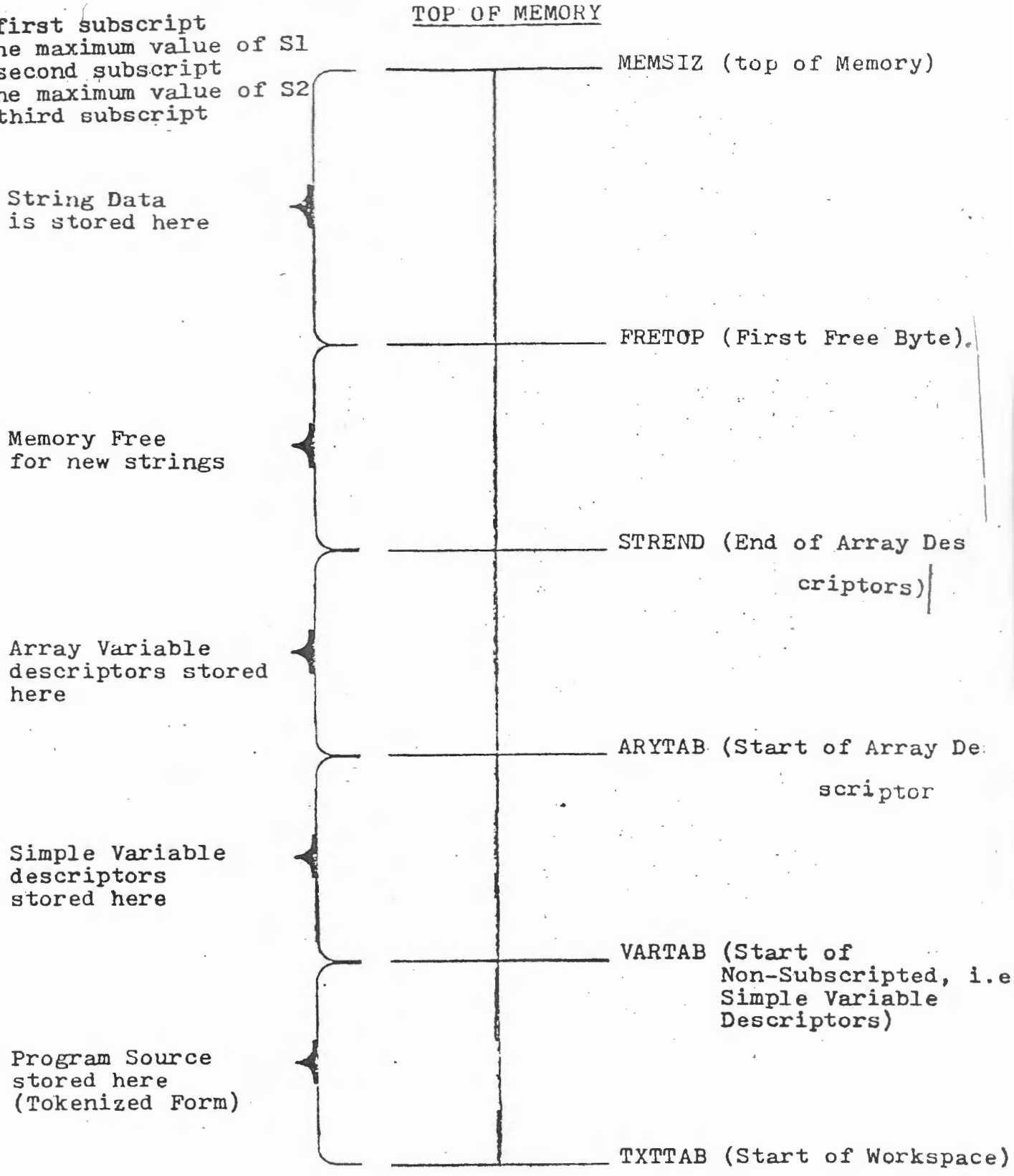
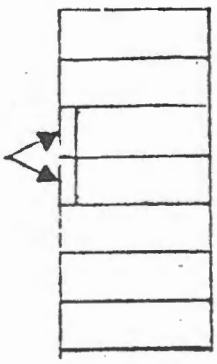


DIAGRAM 1

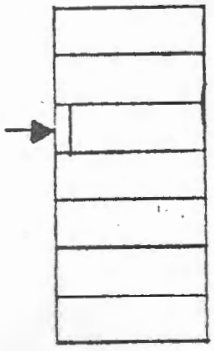
A



SIMPLE FLOATING POINT VARIABLE DESCRIPTOR

First character of name
Reserved for second character of name
Signed exponent with the sign bit complement
Fraction value sign bit, MSB fraction value
Fraction value
Fraction value
LSB of fractional value

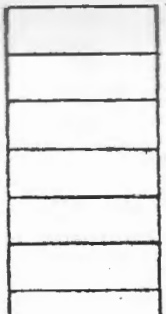
B



SIMPLE INTEGER VARIABLE DESCRIPTOR

First character of name +$8Ø
Second character of name +$8Ø
Sign bit, MSB
LSB
Ø - meaningless
Ø - meaningless
Ø - meaningless

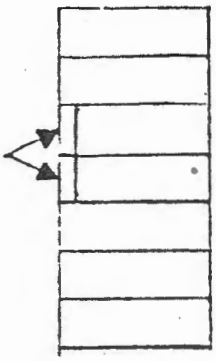
C



SIMPLE STRING VARIABLE DESCRIPTOR

First character of name
Second character of name + $8Ø
Length of string
Low address of string
High address of string
Ø - meaningless

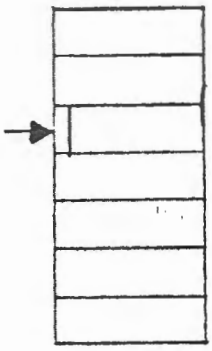
A



SIMPLE FLOATING POINT VARIABLE DESCRIPTOR

First character of name
Reserved for second character of name
Signed exponent with the sign bit complement
Fraction value sign bit, MSB fraction value
Fraction value
Fraction value
LSB of fractional value

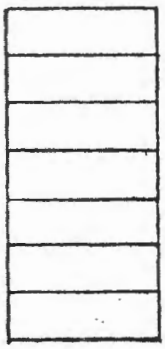
B



SIMPLE INTEGER VARIABLE DESCRIPTOR

First character of name +$8Ø
Second character of name +$8Ø
Sign bit, MSB
LSB
Ø - meaningless
Ø - meaningless
Ø - meaningless

C

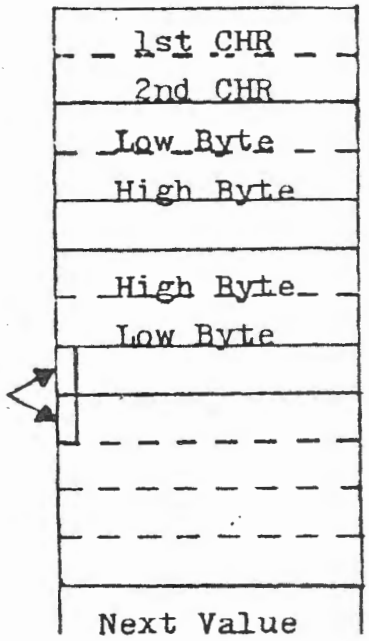


SIMPLE STRING VARIABLE DESCRIPTOR

First character of name
Second character of name + $8Ø
Length of string
Low address of string
High address of string
Ø - meaningless
Ø - meaningless

Diagram 2

FLOATING POINT ARRAY VARIABLE DESCRIPTOR

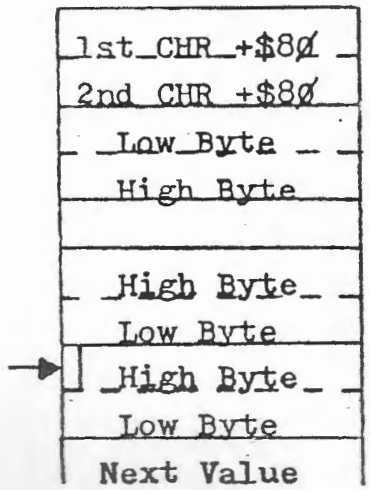


Name

Total length of array in bytes

Number of subscripts
Maximum subscript +1 (May repeat, one for each subscript)
Signed exponent with sign bit complemented
MSB of Fractional value
Fractional value continued
Fractional value continued
LSB of fractional value

INTEGER ARRAY VARIABLE DESCRIPTOR

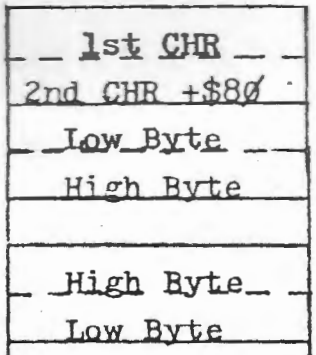


Name

Total length of array in bytes

Number of subscripts
Maximum subscript +1 (may repeat, one for each subscript)
Integer in two's complement form

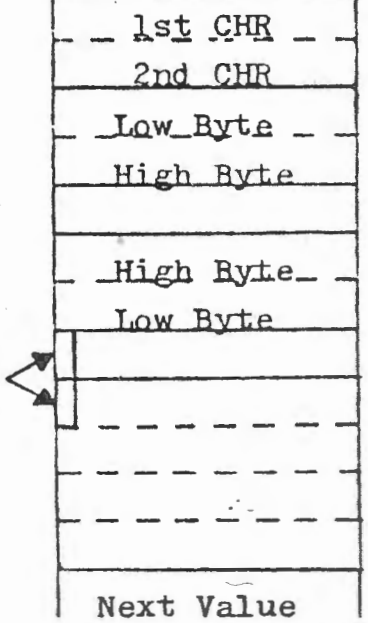
STRING ARRAY VARIABLE DESCRIPTOR



Name

Total length of array in bytes

Number of subscripts
Maximum subscript +1 (may repeat, one for each subscript)

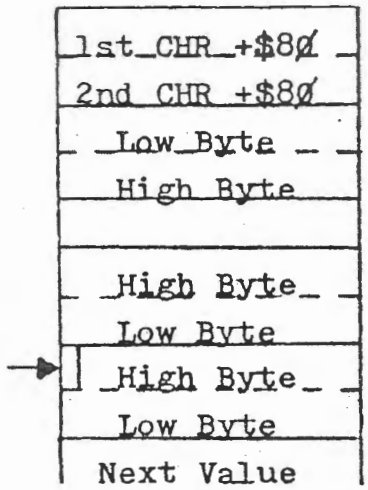


Name

Total length of array in bytes

Number of subscripts
Maximum subscript +1 (May repeat, one for each subscript)
Signed exponent with sign bit complemented
MSB of Fractional value
Fractional value continued
Fractional value continued
LSB of fractional value

INTEGER ARRAY VARIABLE DESCRIPTOR



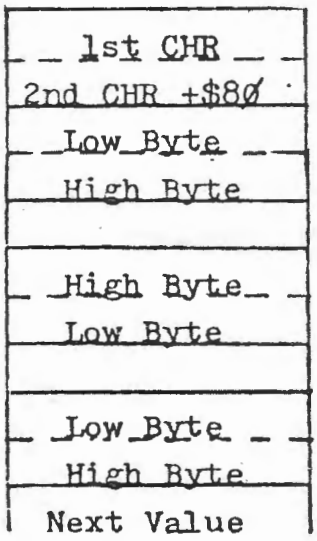
Name

Total length of array in bytes

Number of subscripts
Maximum subscript +1 (may repeat, one for each subscript)

Integer in two's complement form

STRING ARRAY VARIABLE DESCRIPTOR



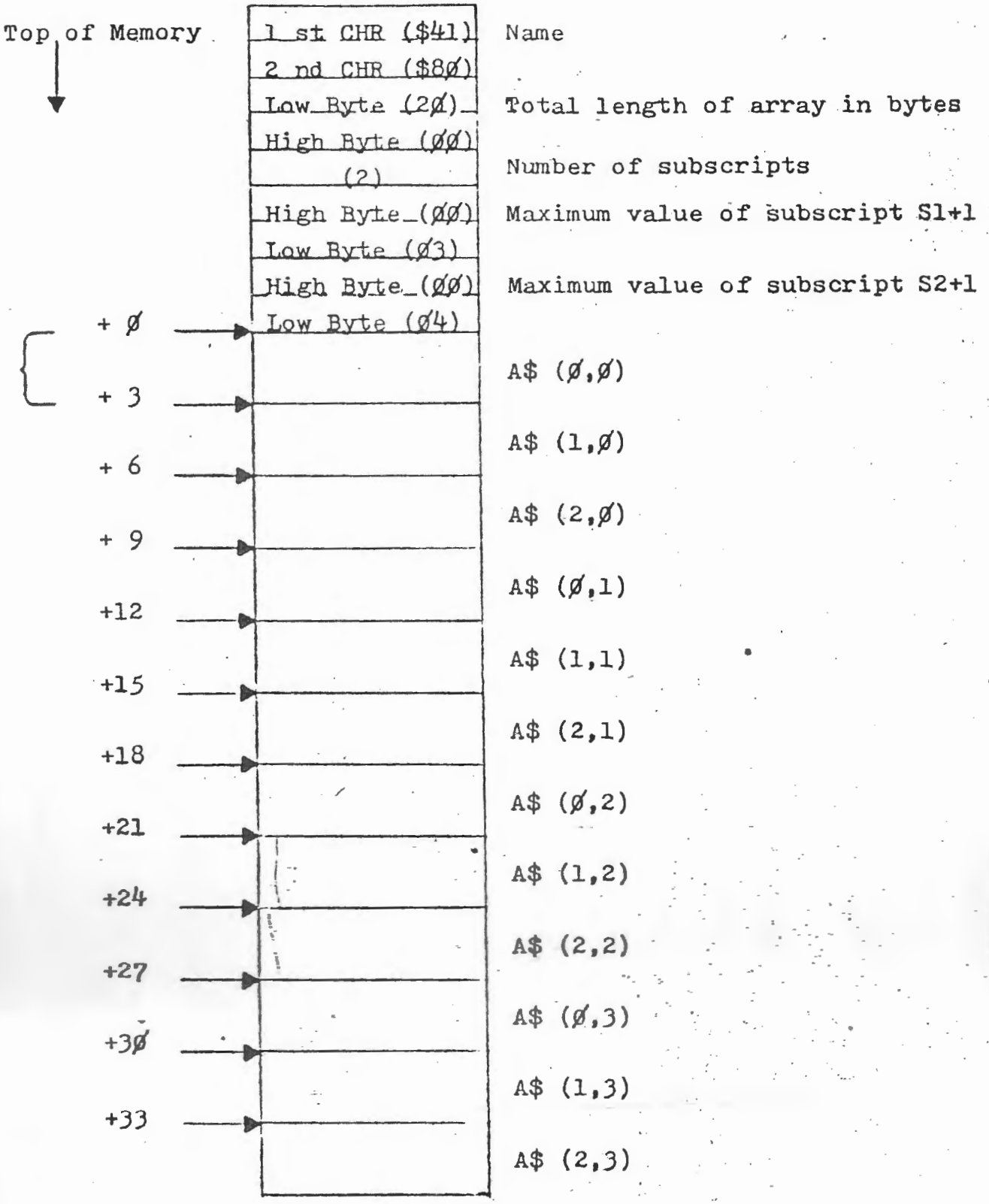
Name

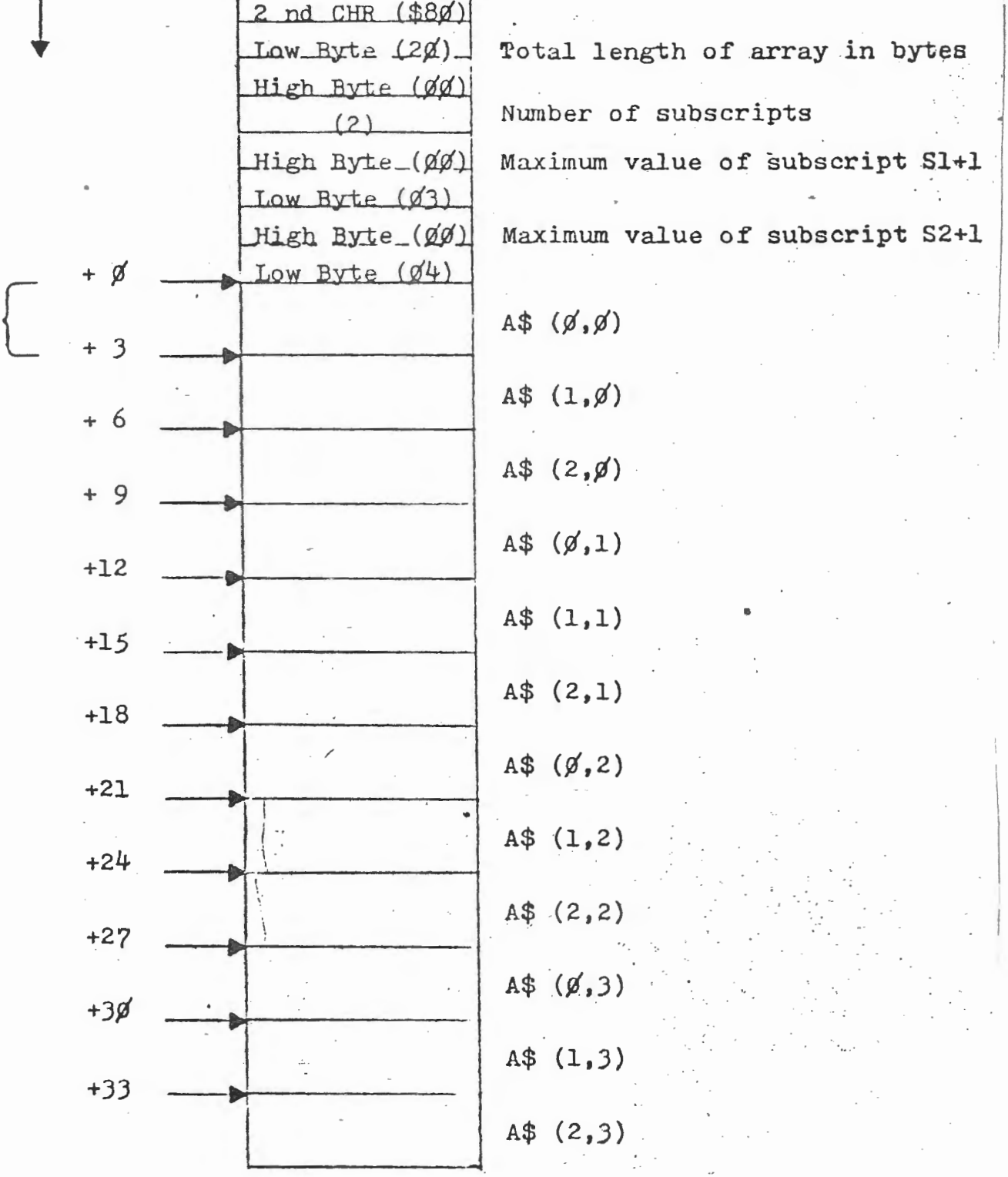
Total length of array in bytes

Number of subscripts
Maximum subscript +1 (may repeat, one for each subscript)
Length of string

Address pointer to actual string data

Diagram 3


Top of Memory
1 st CHR ($41)
2 nd CHR ($8Ø)
Low Byte (2Ø)
High Byte (ØØ)
(2)
High Byte (ØØ)
Low Byte (Ø3)
High Byte (ØØ)
Low Byte (Ø4)
Name
Total length of array in bytes
Number of subscripts
Maximum value of subscript S1+1
Maximum value of subscript S2+1
+ Ø
+ 3
+ 6
+ 9
+12
+15
+18
+21
+24
+27
+3Ø
+33
A$ (Ø,Ø)
A$ (1,Ø)
A$ (2,Ø)
A$ (Ø,1)
A$ (1,1)
A$ (2,1)
A$ (Ø,2)
A$ (1,2)
A$ (2,2)
A$ (Ø,3)
A$ (1,3)
A$ (2,3)



ARRAY A$ (2,3) IN MEMORY

Diagram 4

Hangman.- Mike Bassman

This is the word game of hangman done for the C1P.

If you wish to enlarge or change the vocabulary words (contained in DATA statements in lines 1000-1090) you must change the variable for the number of words,NW, in line 5.

```
1 GOSUB500
2 REM**HANGMAN**MIKE BASSMAN
5 NW=50
7 L1=53524
10 FORK=53248TO54272:POKEK,32:NEXT
15 DIMA(30)
20 FORK=53928TO53932:POKEK,161:NEXT
30 FORK=53514TO53898STEP32:POKEK,161:NEXT
40 FORK=53515TO53520:POKEK,161:NEXT
50 X=RND(X)
60 X=INT(RND(X)*NW+1)
70 FORK=1TOX:READW$:NEXTK
80 FORK=1TOLEN(W$):POKE54055+K,95:NEXTK
90 POKE11,0:POKE12,253
100 X=USR(X)
110 C=PEEK(531)
115 IFC<65THEN100
120 FORK=1TOLEN(W$)
130 IFASC(MID$(W$,K,1))=CTHEN400
140 NEXTK:GOTO600
142 NR=NR+1
145 L=L+2:IFL=12THENL=0:L1=L1+34
147 POKEL1+L-2,C
150 ONNRGOSUB160,200,240,280,310,330,350,370,380,390
155 IFNR=10THEN5000
157 GOTO100
160 POKE53550,176:POKE53551,161:POKE53552,173
170 POKE53582,177:POKE53583,161:POKE53584,175
180 RETURN
200 POKE53614,210:POKE53616,207:POKE53710,209:POKE53712,203
210 POKE53615,135:POKE53711,123
220 FORK=53646TO53678STEP32:POKEK,136:POKEK+2,143:NEXTK
230 RETURN
240 POKE53742,187:POKE53774,187:RETURN
280 POKE53744,187:POKE53776,187:RETURN
310 POKE53644,150:POKE53645,150:RETURN
330 POKE53649,151:POKE53650,151:RETURN
350 POKE53773,150:RETURN
370 POKE53777,150:RETURN
380 POKE53676,168:RETURN
390 POKE53618,165:RETURN
400 FORK=1TOLEN(W$)
410 IFASC(MID$(W$,K,1))=CTHENGOSUB450:POKE54023+K,C
420 NEXTK
```

```
1 GOSUB500
2 REM**HANGMAN**MIKE BASSMAN
5 NW=50
7 L1=53524
10 FORK=53248TO54272:POKEK,32:NEXT
15 DIMA(30)
20 FORK=53928TO53932:POKEK,161:NEXT
30 FORK=53514TO53898STEP32:POKEK,161:NEXT
40 FORK=53515TO53520:POKEK,161:NEXT
50 X=RND(X)
60 X=INT(RND(X)*NW+1)
70 FORK=1TOX:READW$:NEXTK
80 FORK=1TOLEN(W$):POKE54055+K,95:NEXTK
90 POKE11,0:POKE12,253
100 X=USR(X)
110 C=PEEK(531)
115 IFC<65THEN100
120 FORK=1TOLEN(W$)
130 IFASC(MID$(W$,K,1))=CTHEN400
140 NEXTK:GOTO600
142 NR=NR+1
145 L=L+2:IFL=12THENL=0:L1=L1+34
147 POKEL1+L-2,C
150 ONNRGOSUB160,200,240,280,310,330,350,370,380,390
155 IFNR=10THEN5000
157 GOTO100
160 POKE53550,176:POKE53551,161:POKE53552,173
170 POKE53582,177:POKE53583,161:POKE53584,175
180 RETURN
200 POKE53614,210:POKE53616,207:POKE53710,203:POKE53712,203
210 POKE53615,135:POKE53711,123
220 FORK=53646TO53678STEP32:POKEK,136:POKEK+2,143:NEXTK
230 RETURN
240 POKE53742,187:POKE53774,187:RETURN
280 POKE53744,187:POKE53776,187:RETURN
310 POKE53644,150:POKE53645,150:RETURN
330 POKE53649,151:POKE53650,151:RETURN
350 POKE53773,150:RETURN
370 POKE53777,150:RETURN
380 POKE53676,168:RETURN
390 POKE53618,165:RETURN
400 FORK=1TOLEN(W$)
410 IFASC(MID$(W$,K,1))=CTHENGOSUB450:POKE54023+K,C
420 NEXTK
430 IFNC=LEN(W$)THEN5200
440 GOTO100
450 IFPEEK(54023+K)=32THENNC=NC+1
460 RETURN
```

Listing continued on next page-----)

Hangman listing continued from page 15

```
 500 FORK=53248TO54272:POKEK,32:NEXT
 510 PRINT"        HANGMAN"
 520 PRINT"        -------"
 530 FORK=1TO10:PRINT:NEXT
 540 FORK=1TO2000:NEXT:RETURN
 600 FORG=53524TO53524+34STEP34
 610 FORR=2TO10STEP2
 620 IFPEEK(G+R-2)=CTHENF1=1
 630 NEXTR:NEXTG
 640 IFF1=1THENF1=0:GOTO100
 650 GOTO142
 1000 DATA"HELLO"
 1010 DATAGYPSY,HIEROGLYPHIC,CRUCIBLE,OFTEN,COMPUTER
 1020 DATAINCREDIBLE,CONVERTER,PROGRAM,BICYCLE,LIGHTENING
 1030 DATARADIO,DIFFICULT,CASSETTE,CREATIVE,BIAS,OPTIMIST
 1040 DATAPESSIMIST,INSTRUMENT,WALLET,SHUTTLE,MONEY,FRAME
 1050 DATAGARBAGE,PAPER,BAGGAGE,AIRPLANE,ENGINE,RECORD
 1060 DATACUBICLE,PSYCHOPATH,PORTRAIT,DECIMAL,DICTIONARY
 1070 DATAPHOTOGRAPH,FLUORESCENT,MAGAZINE,PORTABLE,CARTON
 1080 DATADELIGHT,AUTOMOBILE,CLOSET,VOLCANO,INEDIBLE
 1090 DATASLEEVE,CABINET,DUST,XYLOPHONE,CELLO,PURPLE
 5000 D$="YOU LOSE THIS TIME!":GOSUB5010
 5005 GOTO5020
 5010 FORK=1TOLEN(D$):POKE53444+K,ASC(MID$(D$,K,1)):NEXTK:RETURN
 5020 FORK=1TO3000:NEXT
 5030 D$="THE WORD WAS              ":GOSUB5010
 5040 FORK=1TOLEN(W$)
 5050 POKE54023+K,ASC(MID$(W$,K,1)):NEXTK
 5060 FORK=1TO3000:NEXT:GOTO5210
 5200 D$="YOU WIN THIS TIME!":GOSUB5010
 5205 FORK=1TO3000:NEXTK
 5210 INPUT"TRY AGAIN";Y$:IFLEFT$(Y$,1)="Y"THENCLEAR:GOTO5
 5220 FORK=53248TO54272:POKEK,32:NEXT:PRINT"BYE"
OK
```

Lissajous.-Mike Cohen

```
 1 REM**LISSAJOUS**MIKE COHEN
 10 DIMS(20,20),C(15)
 15 TV=53478
 20 INPUT"A<B";A,B
 30 FORM=0TO15:READC(M):NEXTM
 40 FORX=1TO20:FORY=1TO20:S(X,Y)=0:NEXTY,X
 50 FORM=53248TO54171:POKEM,32:NEXTM
 60 PI=3.1415926
 65 Z=18:T=3
```

Draws pretty lissajous patterns for the C1P. Values which produce prettiest patterns are: 1,2 and 1,1.

```
600 FORG=53524TO53524+34STEP34
610 FORR=2TO10STEP2
620 IFPEEK(G+R-2)=CTHENF1=1
630 NEXTR:NEXTG
640 IFF1=1THENF1=0:GOTO100
650 GOTO142
1000 DATA"HELLO"
1010 DATAGYPSY,HIEROGLYPHIC,CRUCIBLE,OFTEN,COMPUTER
1020 DATAINCREDIBLE,CONVERTER,PROGRAM,BICYCLE,LIGHTENING
1030 DATARADIO,DIFFICULT,CASSETTE,CREATIVE,BIAS,OPTIMIST
1040 DATAPESSIMIST,INSTRUMENT,WALLET,SHUTTLE,MONEY,FRAME
1050 DATAGARBAGE,PAPER,BAGGAGE,AIRPLANE,ENGINE,RECORD
1060 DATACUBICLE,PSYCHOPATH,PORTRAIT,DECIMAL,DICTIONARY
1070 DATAPHOTOGRAPH,FLUORESCENT,MAGAZINE,PORTABLE,CARTON
1080 DATADELIGHT,AUTOMOBILE,CLOSET,VOLCANO,INEDIBLE
1090 DATASLEEVE,CABINET,DUST,XYLOPHONE,CELLO,PURPLE
5000 D$="YOU LOSE THIS TIME!":GOSUB5010
5005 GOTO5020
5010 FORK=1TOLEN(D$):POKE53444+K,ASC(MID$(D$,K,1)):NEXTK:RETURN
5020 FORK=1TO3000:NEXT
5030 D$="THE WORD WAS              ":GOSUB5010
5040 FORK=1TOLEN(W$)
5050 POKE54023+K,ASC(MID$(W$,K,1)):NEXTK
5060 FORK=1TO3000:NEXT:GOTO5210
5200 D$="YOU WIN THIS TIME!":GOSUB5010
5205 FORK=1TO3000:NEXTK
5210 INPUT"TRY AGAIN";Y$:IFLEFT$(Y$,1)="Y"THENCLEAR:GOTO5
5220 FORK=53248TO54272:POKEK,32:NEXT:PRINT"BYE"
OK
```

Lissajous.-Mike Cohen

```
1 REM**LISSAJOUS**MIKE COHEN
10 DIMS(20,20),C(15)
15 TV=53478
20 INPUT"A<B";A,B
30 FORM=0TO15:READC(M):NEXTM
40 FORX=1TO20:FORY=1TO20:S(X,Y)=0:NEXTY,X
50 FORM=53248TO54171:POKEM,32:NEXTM
60 PI=3.1415926
65 Z=18:T=3
70 FORTH=0TO2*PISTEP2*PI/180
80 R=Z*SIN(TH*T)
90 X=R*COS(A*TH)
100 Y=R*SIN(B*TH)
105 X=INT(X+Z):Y=INT(Y+Z)
110 GOSUB5000
120 NEXT TH
130 END
1000 DATA32,168,166,155,167,156,170,175
1010 DATA165,169,157,177,154,178,176,161
5000 BT=2↑((XAND1)+2*(YAND1))
5010 XX=INT(X/2):YY=INT(Y/2)
5030 CHAR=S(XX,YY)ORBT
5040 POKETV+32*YY+XX,C(CHAR)
5050 S(XX,YY)=CHAR
5060 RETURN
```

Draws pretty lissajous patterns for the C1P. Values which produce prettiest patterns are: 1,2 and 1,1.

Sources of Software for Ohio Scientific.- Mike Bassman

Aardvark Technical Services
1690 Bolton
Walled Lake, MI 48088 (313) 624-6316

Aurora Software Associates
353 South 100 East, #6
Springville, Utah 84663

Creative Computing
P.O. Box 789-M
Morristown, NJ 07960

Mountain Software
25600 Village Circle
Golden, CO 80401 (303) 526-0692

Procom Software
8 Hampton South
Southampton, MA 01073

Kaleidoscope.- Salomon Lederman

This program produces kaleidoscope patterns for the C1P screen. A lot of program crammed into very little space. Has 8-way symmetry.

```
 1 REM**KALEIDOSCOPE**SALOMON LEDERMAN
 2 REMC=CENTER OF SCREEN:I=L IS DENSITY OF BLOCKS
 5 C=53744:P=32:L=.8:DATA1,1,1,-1,-1,1,-1,-1:FORK=53248TO54272
10 POKEK,32:NEXT:P=193-P
20 FORX=0TO15:FORY=0TOX:IFRND(1)<LTHEN50
30 FORN=1TO4:READA,B:POKEC+A*X+32*B*Y,P:POKEC+A*Y+32*B*X,P:NEXTN
50 RESTORE:NEXTY:NEXTX:P=193-P:GOTO20
OK
```

Basic/Machine Language Variable Passing.- Thomas Cheng

This article will explain how to pass a value to a machine language subroutine and back to Basic using the USR(X) function. A subroutine may be called from Basic by the statement X=USR(X). To do this, locations 11, 12

353 South 100 East, #6
Springville, Utah 84663

Creative Computing
P.O. Box 789-M
Morristown, NJ 07960

Mountain Software
25600 Village Circle
Golden, CO 80401 (303) 526-0692

Procom Software
8 Hampton South
Southampton, MA 01073

Kaleidoscope.- Salomon Lederman

This program produces kaleidoscope patterns for the C1P screen. A lot of program crammed into very little space. Has 8-way symmetry.

```
 1 REM**KALEIDOSCOPE**SALOMON LEDERMAN
 2 REMC=CENTER OF SCREEN:1-L IS DENSITY OF BLOCKS
 5 C=53744:P=32:L=.8:DATA1,1,1,-1,-1,1,-1,-1:FORK=53248TO54272
10 POKEK,32:NEXT:P=193-P
20 FORX=0TO15:FORY=0TOX:IFRND(1)<LTHEN50
30 FORN=1TO4:READA,B:POKEC+A*X+32*B*Y,P:POKEC+A*Y+32*B*X,P:NEXTN
50 RESTORE:NEXTY:NEXTX:P=193-P:GOTO20
OK
```

Basic/Machine Language Variable Passing.- Thomas Cheng

This article will explain how to pass a value to a machine language subroutine and back to Basic using the USR(X) function. A subroutine may be called from Basic by the statement X=USR(X). To do this, locations 11, 12 ($A,$B) must be set to the location of the first byte of the subroutine. For example, if your starting location is 546_{10}, location 11 must be 34_{10} and location 12 must be 02_{10}. To pass the value from Basic (the value is the variable within the X=USR(X)), you must call a subroutine from ROM, at location AE05, which places the number in locations AE,AF (lo, hi byte).

To pass a value back to Basic, the high byte must be left in the accumulator; the low byte in the Y register; and the subroutine at AFC1 must be called.

CONTINUED*****)

Transfer-continued from page 17.

There are limitations on the possible values passed. Negative values are represented by bit 7 of the most insignificant byte being set, so positive numbers are limited by $(2^7-1)*256+255$ or 32767, or -33023 for negative numbers. Below is an example of these transfer techniques. A Basic program will pass a value of 1 (in variable X) to a machine language program which will increment this value, then return to Basic.

BASIC PROGRAM.

```
1Ø POKE 11,34:POKE12,2
2Ø X=1
3Ø X=USR(X)
4Ø PRINT X;:GOTO 3Ø
```

MACHINE LANGUAGE PROGRAM.

(starts at 222_{16})

```
222 JSR AEØ5

225 LDA AE
227 LDY AF
229 INY
22A JSR AFC1
22D RTS
```