APPENDICES

APPENDIX A

HOW TO LOAD BASIC

When the ALTAIR is first turned on, there is random garbage in its memory. BASIC is supplied on a paper tape or audio cassette. Somehow the information on the paper tape or cassette must be transfered into the computer. Programs that perform this type of information transfer are called loaders.

Since initially there is nothing of use in memory; you must toggle in, using the switches on the front panel, a 20 instruction bootstrap loader. This loader will then load BASIC.

To load BASIC follow these steps:

1) Turn the ALTAIR on.

2) Raise the STOP switch and RESET switch simultaneously.

3) Turn your terminal (such as a Teletype) to LINE.

Because the instructions must be toggled in via the switches on the front panel, it is rather inconvenient to specify the positions of each switch as "up" or "down". Therefore, the switches are arranged in groups of 3 as indicated by the broken lines below switches 0 through 15. To specify the positions of each switch, we use the numbers 0 through 7 as shown below:

3 SWITCH GROUP

LEFTMOST	MIDDLE	RIGHTMOST	OCTAL NUMBER
Down	Down	Down	0
Down	Down	Up	1
Down	Up	Down	2
Down	Up	Up	3
Up	Down	Down	4
Up	Down	Up	5
Up	Up	Down	6
Up	Up	Up	7

So, to put the octal number 315 in switches 0 through 7, the switches would have the following positions:

7	6	5	4	3	2	1	0	← SWITCH
UP	UP	DOWN	DOWN	UP	UP	DOWN	UP	← POSITION
3			1			5		← OCTAL NO.

Note that switches 8 through 15 were not used. Switches 0 through 7 correspond to the switches labeled DATA on the front panel. A memory address would use all 16 switches.

The following program is the bootstrap loader for users loading from paper tape, and not using a REV 0 Serial I/O Board.

OCTAL ADDRESS	OCTAL DATA	
000	041	
001	175	
002	037	(for 8K; for 4K use 017)
003	061	
004	022	
005	000	
006	333	
007	000	
010	017	
011	330	
012	333	
013	001	
014	275	
015	310	
016	055	
017	167	
020	300	
021	351	
022	003	
023	000	

The following 21 byte bootstrap loader is for users loading from a paper tape and using a REV 0 Serial I/O Board on which the update changing the flag bits has not been made. If the update has been made, use the above bootstrap loader.

OCTAL ADDRESS	OCTAL DATA	
000	041	
001	175	
002	037	(for 8K; for 4K use 017)
003	061	
004	023	
005	000	
006	333	
007	000	
010	346	
011	040	
012	310	
013	333	
014	001	
015	275	
016	310	
017	055	
020	167	

OCTAL ADDRESS (cont.)	OCTAL DATA
021	300
022	351
023	003
024	000

The following bootstrap loader is for users with BASIC supplied on an audio cassette.

OCTAL ADDRESS	OCTAL DATA	
000	041	
001	175	
002	037	(for 8K; for 4K use 017)
003	061	
004	022	
005	000	
006	333	
007	006	
010	017	
011	330	
012	333	
013	007	
014	275	
015	310	
016	055	
017	167	
020	300	
021	351	
022	003	
023	000	

To load a bootstrap loader:

1) Put switches 0 through 15 in the down position.

2) Raise EXAMINE.

3) Put 041 (data for address 000) in switches 0 through 7.

4) Raise DEPOSIT.

5) Put the data for the next address in switches 0 through 7.

6) Depress DEPOSIT NEXT.

7) Repeat steps 5 & 6 until the entire loader is toggled in.

8) Put switches 0 through 15 in the down position.

9) Raise EXAMINE.

10) Check that lights D0 through D7 correspond with the data that should

be in address 000. A light on means the switch was up, a light off means the switch was down. So for address 000, lights D1 through D4 and lights D6 & D7 should be off, and lights D0 and D5 should be on.

If the correct value is there, go to step 13. If the value is wrong, continue with step 11.

11) Put the correct value in switches 0 through 7.

12) Raise DEPOSIT.

13) Depress EXAMINE NEXT.

14) Repeat steps 10 through 13, checking to see that the correct data is in each corresponding address for the entire loader.

15) If you encountered any mistakes while checking the loader, go back now and re-check the whole program to be sure it is corrected.

16) Put the tape of BASIC into the tape reader. Be sure the tape is positioned at the beginning of the leader. The leader is the section of tape at the beginning with 6 out of the 8 holes punched.

If you are loading from audio cassette, put the cassette in the recorder. Be sure the tape is fully rewound.

17) Put switches 0 through 15 in the down position.

18) Raise EXAMINE.

19) If you have connected to your terminal a REV 0 Serial I/O Board on which the update changing the flag bits has not been made, raise switch 14; if you are loading from an audio cassette, raise switch 15 also.

If you have a REV 0 Serial I/O Board which has been updated, or have a REV 1 I/O Board, switch 14 should remain down and switch 15 should be raised only if you are loading from audio cassette.

20) Turn on the tape reader and then depress RUN. Be sure RUN is depressed while the reader is still on the leader. Do not depress run before turning on the reader, since this may cause the tape to be read incorrectly.

If you are loading from a cassette, turn the cassette recorder to Play. Wait 15 seconds and then depress RUN.

21) Wait for the tape to be read in. This should take about 12 minutes for 8K BASIC and 6 minutes for 4K BASIC. It takes about 4 minutes to load 8K BASIC from cassette, and about 2 minutes for 4K BASIC.

Do not move the switches while the tape is being read in.

22) If a C or an O is printed on the terminal as the tape reads in, the tape has been mis-read and you should start over at step 1 on page 46.

23) When the tape finishes reading, BASIC should start up and print MEMORY SIZE?. See Appendix B for the initialization procedure.

24) If BASIC refuses to load from the Audio Cassette, the ACR Demodulator may need alignment. The flip side of the cassette contains 90 seconds of 125's (octal) which were recorded at the same tape speed as BASIC. Use the Input Test Program described on pages 22 and 28 of the ACR manual to perform the necessary alignment.

APPENDIX B

INITIALIZATION DIALOG

STARTING BASIC

Leave the sense switches as they were set for loading BASIC (Appendix A). After the initialization dialog is complete, and BASIC types OK, you are free to use the sense switches as an input device (I/O port 255).

After you have loaded BASIC, it will respond:

MEMORY SIZE?

If you type a carriage return to MEMORY SIZE?, BASIC will use all the contiguous memory upwards from location zero that it can find. BASIC will stop searching when it finds one byte of ROM or non-existent memory.

If you wish to allocate only part of the ALTAIR's memory to BASIC, type the number of bytes of memory you wish to allocate in decimal. This might be done, for instance, if you were using part of the memory for a machine language subroutine.

There are 4096 bytes of memory in a 4K system, and 8192 bytes in an 8K system.

BASIC will then ask:

TERMINAL WIDTH?	This is to set the output line width for PRINT statements only. Type in the number of characters for the line width for the particular terminal or other output device you are using. This may be any number from 1 to 255, depending on the terminal. If no answer is given (i.e. a carriage return is typed) the line width is set to 72 characters.

Now ALTAIR BASIC will enter a dialog which will allow you to delete some of the arithmetic functions. Deleting these functions will give more memory space to store your programs and variables. However, you will not be able to call the functions you delete. Attempting to do so will result in an FC error. The only way to restore a function that has been deleted is to reload BASIC.

The following is the dialog which will occur:

4K Version

WANT SIN?	Answer " Y " to retain SIN, SQR and RND. If you answer " N ", asks next question.
WANT SQR?	Answer " Y " to retain SQR and RND. If you answer " N ", asks next question.

```
WANT RND?
```
Answer " Y " to retain RND.
Answer " N " to delete RND.

8K Version

```
WANT SIN-COS-TAN-ATN?
```
Answer " Y " to retain all four of the functions, " N " to delete all four, or " A " to delete ATN only.

Now BASIC will type out:

```
XXXX BYTES FREE

ALTAIR BASIC VERSION 3.0
[FOUR-K VERSION]
      (or)
[EIGHT-K VERSION]

OK
```

"XXXX" is the number of bytes available for program, variables, matrix storage and the stack. It does not include string space.

You will now be ready to begin using ALTAIR BASIC.

APPENDIX C

ERROR MESSAGES

After an error occurs, BASIC returns to command level and types OK. Variable values and the program text remain intact, but the program can not be continued and all GOSUB and FOR context is lost.
When an error occurs in a direct statement, no line number is printed.

Format of error messages:

Direct Statement ?XX ERROR

Indirect Statement ?XX ERROR IN YYYYY

In both of the above examples, "XX" will be the error code. The "YYYYY" will be the line number where the error occured for the indirect statement.

The following are the possible error codes and their meanings:

ERROR CODE	MEANING
4K VERSION	
BS	Bad Subscript. An attempt was made to reference a matrix element which is outside the dimensions of the matrix. In the 8K version, this error can occur if the wrong number of dimensions are used in a matrix reference; for instance, LET A(1,1,1)=Z when A has been dimensioned DIM A(2,2).
DD	Double Dimension. After a matrix was dimensioned, another dimension statement for the same matrix was encountered. This error often occurs if a matrix has been given the default dimension 10 because a statement like A(I)=3 is encountered and then later in the program a DIM A(100) is found.
FC	Function Call error. The parameter passed to a math or string function was out of range. FC errors can occur due to: a) a negative matrix subscript (LET A(-1)=0) b) an unreasonably large matrix subscript (>32767) c) LOG-negative or zero argument d) SQR-negative argument

e) A↑B with A negative and B not an integer

f) a call to USR before the address of the machine language subroutine has been patched in

g) calls to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC or ON...GOTO with an improper argument.

ID Illegal Direct. You cannot use an INPUT or *(in 8K Version)* DEFFN statement as a direct command.

NF NEXT without FOR. The variable in a NEXT statement corresponds to no previously executed FOR statement.

OD Out of Data. A READ statement was executed but all of the DATA statements in the program have already been read. The program tried to read too much data or insufficient data was included in the program.

OM Out of Memory. Program too large, too many variables, too many FOR loops, too many GOSUB's, too complicated an expression or any combination of the above. (see Appendix D)

OV Overflow. The result of a calculation was too large to be represented in BASIC's number format. If an underflow occurs, zero is given as the result and execution continues without any error message being printed.

SN Syntax error. Missing parenthesis in an expression, illegal character in a line, incorrect punctuation, etc.

RG RETURN without GOSUB. A RETURN statement was encountered without a previous GOSUB statement being executed.

US Undefined Statement. An attempt was made to GOTO, GOSUB or THEN to a statement which does not exist.

/0 Division by Zero.

8K VERSION (Includes all of the previous codes in addition to the following.)

CN Continue error. Attempt to continue a program when none exists, an error occured, or after a new line was typed into the program.

LS	Long String. Attempt was made by use of the concatenation operator to create a string more than 255 characters long.
OS	Out of String Space. Save your program on paper tape or cassette, reload BASIC and allocate more string space or use smaller strings or less string variables.
ST	String Temporaries. A string expression was too complex. Break it into two or more shorter ones.
TM	Type Mismatch. The left hand side of an assignment statement was a numeric variable and the right hand side was a string, or vice versa; or, a function which expected a string argument was given a numeric one or vice versa.
UF	Undefined Function. Reference was made to a user defined function which had never been defined.

APPENDIX D

SPACE HINTS

In order to make your program smaller and save space, the following hints may be useful.

1) Use multiple statements per line. There is a small amount of overhead (5bytes) associated with each line in the program. Two of these five bytes contain the line number of the line in binary. This means that no matter how many digits you have in your line number (minimum line number is 0, maximum is 65529), it takes the same number of bytes. Putting as many statements as possible on a line will cut down on the number of bytes used by your program.

2) Delete all unnecessary spaces from your program. For instance:

```
10 PRINT X, Y, Z
```

uses three more bytes than

```
10 PRINTX,Y,Z
```

Note: All spaces between the line number and the first non-blank character are ignored.

3) Delete all REM statements. Each REM statement uses at least one byte plus the number of bytes in the comment text. For instance, the statement 130 REM THIS IS A COMMENT uses up 24 bytes of memory.

In the statement 140 X=X+Y: REM UPDATE SUM, the REM uses 14 bytes of memory including the colon before the REM.

4) Use variables instead of constants. Suppose you use the constant 3.14159 ten times in your program. If you insert a statement

```
10 P=3.14159
```

in the program, and use P instead of 3.14159 each time it is needed, you will save 40 bytes. This will also result in a speed improvement.

5) A program need not end with an END; so, an END statement at the end of a program may be deleted.

6) Reuse the same variables. If you have a variable T which is used to hold a temporary result in one part of the program and you need a temporary variable later in your program, use it again. Or, if you are asking the terminal user to give a YES or NO answer to two different questions at two different times during the execution of the program, use the same temporary variable A$ to store the reply.

7) Use GOSUB's to execute sections of program statements that perform identical actions.

8) If you are using the 8K version and don't need the features of the 8K version to run your program, consider using the 4K version instead. This will give you approximately 4.7K to work with in an 8K machine, as opposed to the 1.6K you have available in an 8K machine running the 8K version of BASIC.

9) Use the zero elements of matrices; for instance, A(0), B(0,X).

STORAGE ALLOCATION INFORMATION

Simple (non-matrix) numeric variables like V use 6 bytes; 2 for the variable name, and 4 for the value. Simple non-matrix string variables also-use 6 bytes; 2 for the variable name, 2 for the length, and 2 for a pointer.

Matrix variables use a minimum of 12 bytes. Two bytes are used for the variable name, two for the size of the matrix, two for the number of dimensions and two for each dimension along with four bytes for each of the matrix elements.

String variables also use one byte of string space for each character in the string. This is true whether the string variable is a simple string variable like A$, or an element of a string matrix such as Q1$(5,2).

When a new function is defined by a DEF statement, 6 bytes are used to store the definition.

Reserved words such as FOR, GOTO or NOT, and the names or the intrinsic functions such as COS, INT and STR$ take up only one byte of program storage. All other characters in programs use one byte of program storage each.

When a program is being executed, space is dynamically allocated on the stack as follows:

1) Each active FOR...NEXT loop uses 16 bytes.

2) Each active GOSUB (one that has not returned yet) uses 6 bytes.

3) Each parenthesis encountered in an expression uses 4 bytes and each temporary result calculated in an expression uses 12 bytes.

APPENDIX E

SPEED HINTS

The hints below should improve the execution time of your BASIC program. Note that some of these hints are the same as those used to decrease the space used by your programs. This means that in many cases you can increase the efficiency of both the speed and size of your programs at the same time.

1) Delete all unnecessary spaces and REM's from the program. This may cause a small decrease in execution time because BASIC would otherwise have to ignore or skip over spaces and REM statements.

2) *THIS IS PROBABLY THE MOST IMPORTANT SPEED HINT BY A FACTOR OF 10.* Use variables instead of constants. It takes more time to convert a constant to its floating point representation than it does to fetch the value of a simple or matrix variable. This is especially important within FOR...NEXT loops or other code that is executed repeatedly.

3) Variables which are encountered first during the execution of a BASIC program are allocated at the start of the variable table. This means that a statement such as 5 A=0:B=A:C=A, will place A first, B second, and C third in the symbol table (assuming line 5 is the first statement executed in the program). Later in the program, when BASIC finds a reference to the variable A, it will search only one entry in the symbol table to find A, two entries to find B and three entries to find C, etc.

4) *(8K Version)* NEXT statements without the index variable. NEXT is somewhat faster than NEXT I because no check is made to see if the variable specified in the NEXT is the same as the variable in the most recent FOR statement.

5) Use the 8K version instead of the 4K version. The 8K version is about 40% faster than the 4K due to improvements in the floating point arithmetic routines.

6) The math functions in the 8K version are much faster than their counterparts simulated in the 4K version. (see Appendix G)

APPENDIX F

DERIVED FUNCTIONS

The following functions, while not intrinsic to ALTAIR BASIC, can be calculated using the existing BASIC functions.

FUNCTION	FUNCTION EXPRESSED IN TERMS OF BASIC FUNCTIONS
SECANT	SEC(X) = 1/COS(X)
COSECANT	CSC(X) = 1/SIN(X)
COTANGENT	COT(X) = 1/TAN(X)
INVERSE SINE	ARCSIN(X) = ATN(X/SQR(-X*X+1))
INVERSE COSINE	ARCCOS(X) = -ATN(X/SQR(-X*X+1))+1.5708
INVERSE SECANT	ARCSEC(X) = ATN(SQR(X*X-1))+(SGN(X)-1)*1.5708
INVERSE COSECANT	ARCCSC(X) = ATN(1/SQR(X*X-1))+(SGN(X)-1)*1.5708
INVERSE COTANGENT	ARCCOT(X) = -ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X) = (EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X) = (EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X) = -EXP(-X)/(EXP(X)+EXP(-X))*2+1
HYPERBOLIC SECANT	SECH(X) = 2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	CSCH(X) = 2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X) = EXP(-X)/(EXP(X)-EXP(-X))*2+1
INVERSE HYPERBOLIC SINE	ARGSINH(X) = LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARGCOSH(X) = LOG(X+SQR(X*X-1))
INVERSE HYPERBOLIC TANGENT	ARGTANH(X) = LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARGSECH(X) = LOG((SQR(-X*X+1)+1)/X)
INVERSE HYPERBOLIC COSECANT	ARGCSCH(X) = LOG((SGN(X)*SQR(X*X+1)+1)/X)
INVERSE HYPERBOLIC COTANGENT	ARGCOTH(X) = LOG((X+1)/(X-1))/2

APPENDIX G

SIMULATED MATH FUNCTIONS

The following subroutines are intended for 4K BASIC users who want to use the transcendental functions not built into 4K BASIC. The corresponding routines for these functions in the 8K version are much faster and more accurate. The REM statements in these subroutines are given for documentation purposes only, and should not be typed in because they take up a large amount of memory.

The following are the subroutine calls and their 8K equivalents:

8K EQUIVALENT	SUBROUTINE CALL
P9=X9↑Y9	GOSUB 60030
L9=LOG(X9)	GOSUB 60090
E9=EXP(X9)	GOSUB 60160
C9=COS(X9)	GOSUB 60240
T9=TAN(X9)	GOSUB 60280
A9=ATN(X9)	GOSUB 60310

The unneeded subroutines should not be typed in. Please note which variables are used by each subroutine. Also note that TAN and COS require that the SIN function be retained when BASIC is loaded and initialized.

```
60000 REM EXPONENTIATION: P9=X9↑Y9
60010 REM NEED: EXP, LOG
60020 REM VARIABLES USED: A9,B9,C9,E9,L9,P9,X9,Y9
60030 P9=1 : E9=0 : IF Y9=0 THEN RETURN
60040 IF X9<0 THEN IF INT(Y9)=Y9 THEN P9=1-2*Y9+4*INT(Y9/2) : X9=-X9
60050 IF X9<>0 THEN GOSUB 60090 : X9=Y9*L9 : GOSUB 60160
60060 P9=P9*E9 : RETURN
60070 REM NATURAL LOGARITHM: L9=LOG(X9)
60080 REM VARIABLES USED: A9,B9,C9,E9,L9,X9
60090 E9=0 : IF X9<=0 THEN PRINT "LOG FC ERROR"; : STOP
60095 A9=1 : B9=2 : C9=.5 : REM THIS WILL SPEED UP THE FOLLOWING
60100 IF X9>=A9 THEN X9=C9*X9 : E9=E9+A9 : GOTO 60100
60110 IF X9<C9 THEN X9=B9*X9 : E9=E9-A9 : GOTO 60110
60120 X9=(X9-.707107)/(X9+.707107) : L9=X9*X9
60130 L9=(((.598979*L9+.961471)*L9+2.88539)*X9+E9-.5)*.693147
60135 RETURN
60140 REM EXPONENTIAL: E9=EXP(X9)
60150 REM VARIABLES USED: A9,E9,L9,X9
60160 L9=INT(1.4427*X9)+1 : IF L9<127 THEN 60180
60170 IF X9>0 THEN PRINT "EXP OV ERROR"; : STOP
60175 E9=0 : RETURN
60180 E9=.693147*L9-X9 : A9=1.32988E-3-1.41316E-4*E9
60190 A9=((A9*E9-8.30136E-3)*E9+4.16574E-2)*E9
60195 E9=(((A9-.166665)*E9+.5)*E9-1)*E9+1 : A9=2
60197 IF L9<=0 THEN A9=.5 : L9=-L9 : IF L9=0 THEN RETURN
```

```
60200 FOR X9=1 TO L9 : E9=A9*E9 : NEXT X9 : RETURN
60210 REM COSINE: C9=COS(X9)
60220 REM N.B. SIN MUST BE RETAINED AT LOAD-TIME
60230 REM VARIABLES USED: C9,X9
60240 C9=SIN(X9+1.5708) : RETURN
60250 REM TANGENT: T9=TAN(X9)
60260 REM NEEDS COS. (SIN MUST BE RETAINED AT LOAD-TIME)
60270 REM VARIABLES USED: C9,T9,X9
60280 GOSUB 60240 : T9=SIN(X9)/C9 : RETURN
60290 REM ARCTANGENT: A9=ATN(X9)
60300 REM VARIABLES USED: A9,B9,C9,T9,X9
60310 T9=SGN(X9): X9=ABS(X9): C9=0 : IF X9>1 THEN C9=1 : X9=1/X9
60320 A9=X9*X9 : B9=((2.86623E-3*A9-1.61657E-2)*A9+4.29096E-2)*A9
60330 B9=((((B9-7.5289E-2)*A9+.106563)*A9-.142089)*A9+.199936)*A9
60340 A9=((B9-.333332)*A9+1)*X9 : IF C9=1 THEN A9=1.5708-A9
60350 A9=T9*A9 : RETURN
```

APPENDIX H

CONVERTING BASIC PROGRAMS NOT WRITTEN FOR THE ALTAIR

Though implementations of BASIC on different computers are in many ways similar, there are some incompatibilites which you should watch for if you are planning to convert some BASIC programs that were not written for the ALTAIR.

1) Matrix subscripts. Some BASICs use " [" and "] " to denote matrix subscripts. ALTAIR BASIC uses " (" and ") ".

2) Strings. A number of BASICs force you to dimension (declare) the length of strings before you use them. You should remove all dimension statements of this type from the program. In some of these BASICs, a declaration of the form DIM A$(I,J) declares a string matrix of J elements each of which has a length I. Convert DIM statements of this type to equivalent ones in ALTAIR BASIC: DIM A$(J).

ALTAIR BASIC uses " + " for string concatenation, not " , " or " & ".

ALTAIR BASIC uses LEFT$, RIGHT$ and MID$ to take substrings of strings. Other BASICs use A$(I) to access the Ith character of the string A$, and A$(I,J) to take a substring of A$ from character position I to character position J. Convert as follows:

OLD	NEW
A$(I)	MID$(A$,I,1)
A$(I,J)	MID$(A$,I,J-I+1)

This assumes that the reference to a substring of A$ is in an expression or is on the right side of an assignment. If the reference to A$ is on the left hand side of an assignment, and X$ is the string expression used to replace characters in A$, convert as follows:

OLD	NEW
A$(I)=X$	A$=LEFT$(A$,I-1)+X$+MID$(A$,I+1)
A$(I,J)=X$	A$=LEFT$(A$,I-1)+X$+MID$(A$,J+1)

3) Multiple assignments. Some BASICs allow statements of the form: 500 LET B=C=0. This statement would set the variables B & C to zero.

In 8K ALTAIR BASIC this has an entirely different effect. All the " ='s " to the right of the first one would be interpreted as logical comparison operators. This would set the variable B to -1 if C equaled 0. If C did not equal 0, B would be set to 0. The easiest way to convert statements like this one is to rewrite them as follows:

```
500 C=0:B=C.
```

4) Some BASICs use " \ " instead of " : " to delimit multiple statements per line. Change the " \'s " to " :'s " in the program.

5) Paper tapes punched by other BASICs may have no nulls at the end of each line, instead of the three per line recommended for use with ALTAIR BASIC.

To get around this, try to use the tape feed control on the Teletype to stop the tape from reading as soon as ALTAIR BASIC types a carriage return at the end of the line. Wait a second, and then continue feeding in the tape.

When you have finished reading in the paper tape of the program, be sure to punch a new tape in ALTAIR BASIC's format. This will save you from having to repeat this process a second time.

6) Programs which use the MAT functions available in some BASICs will have to be re-written using FOR...NEXT loops to perform the appropriate operations.

APPENDIX I

USING THE ACR INTERFACE

NOTE: The cassette features, CLOAD and CSAVE, are only present in 8K BASICs which are distributed on cassette. 8K BASIC on paper tape will give the user about 130 more bytes of free memory, but it will not recognize the CLOAD or CSAVE commands.

The CSAVE command saves a program on cassette tape. CSAVE takes one argument which can be any printing character. CSAVE can be given directly or in a program. Before giving the CSAVE command start your audio recorder on Record, noting the position of the tape.

CSAVE writes data on channel 7 and expects the device status from channel 6. Patches can easily be made to change these channel numbers.

When CSAVE is finished, execution will continue with the next statement. What is written onto the tape is BASIC's internal representation of the program in memory. The amount of data written onto the tape will be equal to the size of the program in memory plus seven.

Variable values are not saved on the tape, nor are they affected by the CSAVE command. The number of nulls being printed on your terminal at the start of each line has no affect on the CSAVE or CLOAD commands.

CLOAD takes its one character argument just like the CSAVE command. For example, CLOAD E.

The CLOAD command first executes a "NEW" command, erasing the current program and all variable values. The CLOAD command should be given before you put your cassette recorder on Play.

BASIC will read a byte from channel 7 whenever the character ready flag comes up on channel 6. When BASIC finds the program on the tape, it will read all characters received from the tape into memory until it finds three consecutive zeros which mark the end of the program. Then BASIC will return to command level and type "OK".

Statements given on the same line as a CLOAD command are ignored. The program on the cassette is not in a checksummed format, so the program must be checked to make sure it read in properly.

If BASIC does not return to command level and type "OK", it means that BASIC either never found a file with the right filename character, or that BASIC found the file but the file never ended with three consecutive zeros. By carefully watching the front panel lights, you can tell if BASIC ever finds a file with the right name.

Stopping the ALTAIR and restarting it at location 0 will prevent BASIC from searching forever. However, it is likely that there will either be no program in the machine, or a partial program that has errors. Typing NEW will always clear out whatever program is in the machine.

Reading and writing data from the cassette is done with the INP, OUT and WAIT statements. Any block of data written on the tape should have its beginning marked with a character. The main thing to be careful of is allowing your program to fall behind while data passes by unread.

Data read from the cassette should be stored in a matrix, since

there isn't time to process data as it is being read in. You will probably want to detect the end of data on the tape with a special character.

At location 4050=7722 Base 8 put:

```
7722/333        IN      255     ;(255 Base 10=377 Base 8) Get
7723/377                        ;the value of the switches in A
7724/107        MOV     B,A     ;B gets low part of answer
7725/257        XRA     A       ;A gets high part of answer
7726/052        LHLD    6       ;get address of routine
7727/006
7730/000                        ;that floats [A,B]
7731/351        PCHL            ;go to that routine which will
                                ;return to BASIC
                                ;with the answer
```

MORE ON PEEK AND POKE *(8K VERSION ONLY)*

As mentioned before, POKE can be used to set up your machine language routine in high memory. BASIC does not restrict which addresses you can POKE. Modifying USRLOC can be accomplished using two successive calls to POKE. Patches which a user wishes to include in his BASIC can also be made using POKE.

Using the PEEK function and OUT statement of 8K BASIC, the user can write a binary dump program in BASIC. Using INP and POKE it is possible to write a binary loader.

PEEK and POKE can be used to store byte oriented information. When you initialize BASIC, answer the MEMORY SIZE? question with the amount of memory in your ALTAIR minus the amount of memory you wish to use as storage for byte formatted data.

You are now free to use the memory in the top of memory in your ALTAIR as byte storage. See PEEK and POKE in the Reference Material for a further description of their parameters.

APPENDIX K

ASCII CHARACTER CODES

DECIMAL	CHAR.	DECIMAL	CHAR.	DECIMAL	CHAR.
000	NUL	043	+	086	V
001	SOH	044	,	087	W
002	STX	045	-	088	X
003	ETX	046	.	089	Y
004	EOT	047	/	090	Z
005	ENQ	048	0	091	[
006	ACK	049	1	092	\
007	BEL	050	2	093	]
008	BS	051	3	094	↑
009	HT	052	4	095	←
010	LF	053	5	096	`
011	VT	054	6	097	a
012	FF	055	7	098	b
013	CR	056	8	099	c
014	SO	057	9	100	d
015	SI	058	:	101	e
016	DLE	059	;	102	f
017	DC1	060	<	103	g
018	DC2	061	=	104	h
019	DC3	062	>	105	i
020	DC4	063	?	106	j
021	NAK	064	@	107	k
022	SYN	065	A	108	l
023	ETB	066	B	109	m
024	CAN	067	C	110	n
025	EM	068	D	111	o
026	SUB	069	E	112	p
027	ESCAPE	070	F	113	q
028	FS	071	G	114	r
029	GS	072	H	115	s
030	RS	073	I	116	t
031	US	074	J	117	u
032	SPACE	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	$	079	O	122	z
037	%	080	P	123	{
038	&	081	Q	124	|
039	'	082	R	125	}
040	(	083	S	126	~
041	)	084	T	127	DEL
042	*	085	U		

LF=Line Feed FF=Form Feed CR=Carriage Return DEL=Rubout

CHR$ is a string function which returns a one character string which contains the ASCII equivalent of the argument, according to the conversion table on the preceeding page. ASC takes the first character of a string and converts it to its ASCII decimal value.

One of the most common uses of CHR$ is to send a special character to the user's terminal. The most often used of these characters is the BEL (ASCII 7). Printing this character will cause a bell to ring on some terminals and a "beep" on many CRT's. This may be used as a preface to an error message, as a novelty, or just to wake up the user if he has fallen asleep. (Example: PRINT CHR$(7);)

A major use of special characters is on those CRT's that have cursor positioning and other special functions (such as turning on a hard copy printer).

As an example, try sending a form feed (CHR$(12)) to your CRT. On most CRT's this will usually cause the screen to erase and the cursor to "home" or move to the upper left corner.

Some CRT's give the user the capability of drawing graphs and curves in a special point-plotter mode. This feature may easily be taken advantage of through use of ALTAIR BASIC's CHR$ function.

APPENDIX L

EXTENDED BASIC

When EXTENDED BASIC is sent out, the BASIC manual will be updated to contain an extensive section about EXTENDED BASIC. Also, at this time the part of the manual relating to the 4K and 8K versions will be revised to correct any errors and explain more carefully the areas users are having trouble with. This section is here mainly to explain what EXTENDED BASIC will contain.

INTEGER VARIABLES These are stored as double byte signed quantities ranging from -32768 to +32767. They take up half as much space as normal variables and are about ten times as fast for arithmetic. They are denoted by using a percent sign (%) after the variable name. The user doesn't have to worry about conversion and can mix integers with other variable types in expressions. The speed improvement caused by using integers for loop variables, matrix indices, and as arguments to functions such as AND, OR or NOT will be substantial. An integer matrix of the same dimensions as a floating point matrix will require half as much memory.

DOUBLE-PRECISION Double-Precision variables are almost the opposite of integer variables, requiring twice as much space (8bytes per value) and taking 2 to 3 times as long to do arithmetic as single-precision variables. Double-Precision variables are denoted by using a number sign (#) after the variable name. They provide over 16 digits of accuracy. Functions like SIN, ATN and EXP will convert their arguments to single-precision, so the results of these functions will only be good to 6 digits. Negation, addition, subtraction, multiplication, division, comparision, input, output and conversion are the only routines that deal with Double-Precision values. Once again, formulas may freely mix Double-Precision values with other numeric values and conversion of the other values to Double-Precision will be done automatically.

PRINT USING Much like COBOL picture clauses or FORTRAN format statements, PRINT USING provides a BASIC user with complete control over his output format. The user can control how many digits of a number are printed, whether the number is printed in scientific notation and the placement of text in output. All of this can be done in the 8K version using string functions such as STR$ and MID$, but PRINT USING makes it much easier.

DISK I/O EXTENDED BASIC will come in two versions, disk and non-disk. There will only be a copying charge to switch from one to the other. With disk features, EXTENDED BASIC will allow the user to save and recall programs and data files from the ALTAIR FLOPPY DISK. Random access as well as sequential access will be provided. Simultaneous use of multiple data files will be allowed. Utilities will format new disks, delete files and print directories. These will be BASIC programs using special BASIC functions to get access to disk information such as file length, etc. User programs can also access these disk functions, enabling the user to write his own file access method or other special purpose

disk routine. The file format can be changed to allow the use of other (non-floppy) disks. This type of modification will be done by MITS under special arrangement.

OTHER FEATURES Other nice features which will be added are:

Fancy Error Messages
An ELSE clause in IF statements
LIST, DELETE commands with line range as arguments
Deleting Matrices in a program
TRACE ON/OFF commands to monitor program flow
EXCHANGE statement to switch variable values (this will speed up string sorts by at least a factor of two).
Multi-Argument, user defined functions with string arguments and values allowed

Other features contemplated for future release are:

A multiple user BASIC
Explicit matrix manipulation
Virtual matrices
Statement modifiers
Record I/O
Paramaterized GOSUB
Compilation
Multiple USR functions
"Chaining"

EXTENDED BASIC will use about 11K of memory for its own code (10K for the non-disk version) leaving 1K free on a 12K machine. It will take almost 20 minutes to load from paper tape, 7 minutes from cassette, and less than 5 seconds to load from disk.

We welcome any suggestions concerning current features or possible additions of extra features. Just send them to the ALTAIR SOFTWARE DEPARTMENT.

APPENDIX M

BASIC TEXTS

Below are a few of the many texts that may be helpful in learning BASIC.

1) BASIC PROGRAMMING, John G. Kemeny, Thomas E Kurtz, 1967, p145

2) BASIC, Albrecht, Finkel and Brown, 1973

3) A GUIDED TOUR OF COMPUTER PROGRAMMING IN BASIC, Thomas A Dwyer and Michael S. Kaufman; Boston: Houghton Mifflin Co., 1973

Books numbered 1 & 2 may be obtained from:

People's Computer Company
P.O. Box 310
Menlo Park, California
94025

They also have other books of interest, such as:

101 BASIC GAMES, Ed. David Ahl, 1974 p250

WHAT TO DO AFTER YOU HIT RETURN or PCC's FIRST BOOK OF COMPUTER GAMES

COMPUTER LIB & DREAM MACHINES, Theodore H. Nelson, 1974, p186

2450 Alamo SE
Albuquerque, NM 87106