T H E C O D O S I N T E R F A C E L I B R A R Y

C I L

R E F E R E N C E M A N U A L

JANUARY 1982 REV B

TABLE OF CONTENTS

1. INTRODUCTION

The purpose of the CODOS Interface Library is to provide a means of upgrading the standard MTU BASIC to a full Disk BASIC. The CIL command set is designed to be powerful, yet easy to use. Features include the ability to read and write data in ASCII or memory image format. It also provides a means of executing CODOS Monitor commands from a BASIC program. And most importantly, it provides the ability to set or read the position of a CODOS file. It is assumed that the reader of this preliminary Reference Manual has a working knowledge of MTU CODOS.

2. STARTUP PROCEDURE

To bring the CODOS Interface Library into operation you execute a LIB command with "CIL" or "CILB" specified as the first file name in the list. CIL or CILB must be first because the CIL programs only run at one location. This is primarily due to the fact that the memory area used for execution of CODOS utilities (which is fixed), is effectively part of CIL.

The difference between CIL and CILB is that the latter includes the buffer needed by the CODOS COPYF utility while the former does not. If you expect to use COPYF as part of a SYSTEM command (see section 6.13) then CILB must be used. If COPYF is not needed while in BASIC, then CIL can be used which will give your BASIC program 5K more memory than CILB will. If you have LIBed in the CIL or CILB and wish to switch to the other, you must execute a FRELIB command first. Then you may LIB in the other Library.

3. GETTING STARTED WITH THE CIL

The CODOS Interface Library has two main functions. The first is to make it possible to give commands to the CODOS operating system from a BASIC program. This is accomplished with the SYSTEM command. The second function is to provide a means of reading and writing data files on disk. This section will demostrate various ways data may be read or written using the CIL.

The first step in accessing data is to assign a channel to the source or destination of the data. This is done from BASIC using the SYSTEM command to pass an ASSIGN command to the CODOS Monitor. Here are a couple of examples as they might appear in a program:

```
1250 SYSTEM "ASSIGN 6 OUTPUTFILE.D"

2335 SYSTEM "ASSIGN 2 N 5 INPUTDATA.T:1":SS=ST:SYSTEM "ASSIGN 2 C"
```

The first example assigns channel 6 to a file named OUTPUTFILE.D on the default drive. The second is more complicated. It first assigns channel 2 to the Null device. This will prevent the "NEW FILE ..." or "OLD FILE ..." message from appearing on the CRT when the second assignment is made. Next, channel 5 is assigned to a file named INPUTDATA.T on drive 1. After the first SYSTEM command has finished executing, the value of the status variable ST is stored in the variable SS. This is done to save the status from the last channel assignment, i.e. INPUTDATA.T to channel 5. The status value will be changed by the SYSTEM command which follows. This next SYSTEM command reassigns channel 2 back to the Console device. The value returned in SS will indicate if INPUTDATA.T was an old (SS=-64, or SS=-32 if locked) or new (SS=64) file.

You also use the SYSTEM command to free the channel when the transfer of data to that channel is complete. For example:

```
3450 SYSTEM "FREE 5 6"
```

will free channels 5 and 6. It is especially important to free channels assigned to disk files. The last partial buffer full of data may not get written to disk if it is not freed properly. If in doubt about what channels may still be assigned to disk files, executing a CODOS CLOSE command for the disk will properly free any channels still assigned to that disk.

There are three modes in which you can transfer data to or from a channel, using the CIL. The first mode is a byte at a time using the GET@ and PUT@ commands. The second mode is to transfer the data in ASCII format, using the PRINT@ and INPUT@ commands. The third mode is to transfer the data in block format, using the BLKRD@ and BLKWRT@ commands.

To assist in accessing the files, there is a POSITION@ command to position the file assigned to a channel to a particular byte. In addition, there is the TRUNC@ command to truncate a file at its current position. And finally, there is the POSN() function which returns the current position of a file. To illustrate how these command may actually be used, the following examples are provided.

The first step is to bring MTU BASIC into operation. To do this, boot a CODOS disk containing the BASIC.C, and CIL.Z files. Then bring MTU BASIC into operation by executing the command:

```
BASIC
```

At this point you have a choice of two CODOS Interface Libraries which you may load. They are called CIL and CILB. The only difference between them is in the CODOS Utility programs which they allow to be executed from BASIC. The CILB Library allows you to execute all standard CODOS Utilities. The CIL Library allows you to execute all standard utilities which do not use the large transient buffer. This means that while using the CIL library, you may not execute the COPYF or FORMAT utility programs. Since these utilities will not be needed for our examples, execute the follwing command to load the CIL Library:

```
LIB "CIL"
```

In the first sequence of examples, we will experiment with the PUT@, GET@, and POSITION@ commands. First, we will use the PUT@ command to write a file containing 255 bytes. Enter:

```
SYSTEM "ASSIGN 6 DUMMYDATA.D"
```

to assign channel 6 to a file called DUMMYDATA.D. Then enter:

```
FOR I=1 TO 255:PUT@ 6,I:NEXT
```

to write the 255 bytes. The file named DUMMYDATA.D now contains an ascending sequence of bytes, starting with one. Enter:

```
PRINT ST
```

to examine the status value returned by the last PUT@ command executed. The value printed is -127. This value is negative because -128 has been added to the status value to indicate the file pointer is at end-of-file. If we add 128 to the status value, we are left with a value of 1. This indicates that one data item, or variable, was written by the last PUT@ command.

For the next step we will move the pointer back to the beginning of the file, and read the first 10 bytes using the GET@ command. Enter:

```
SYSTEM "BEGINOF 6"
```

to move the file pointer back to the beginning of the file. Then enter:

```
FOR I=1 TO 10:GET@ 6,N:PRINT N:NEXT
```

to read and print the values of the first 10 bytes of the file. You should note that the value of the byte also identifies which byte of the file it is. In other words, the first byte of the file contains a value of 1, the second byte contains 2, etc. We will now use this fact to experiment with the POSITION@ command.

Enter:

```
POSITION@6,0
```

to set the file position to 0. Then enter:

```
GET@ 6,N:PRINT N
```

to read and print the value of the byte which follows position 0 in the file. Since 1 is printed, we were back at the beginning of the file. This means that positioning a file to zero will put the pointer at the very beginning of a file. The next byte read or written at this point will be the first byte in the file.

Now enter:

```
POSITION@6,252
```

to set the file position to 252. Then enter:

```
GET@ 6,N:PRINT N
```

to read and print the value of the next byte in the file. The value printed this time is 253, which is one more than the value of the position we set. To state it in a general way, setting the file position pointer to I will set the file position to just after the Ith byte in the file. The next byte to be read or written is byte number I+1.

Next we will observe what happens when end of file is reached while reading a file. Enter:

```
GET@6, N,N1,N2
```

to attempt to read three more bytes from the file. Then enter:

```
PRINT ST,N,N1,N2
```

to print the status variable and the contents of the three variables we used. The value printed for the status variable is -126. This value indicates that the file pointer reached the end-of-file sometime during the last GET@ command. It also indicates that only two (-126 + 128 = 2) of the three variables received values from the file. This is verified by the values printed for N, N1, and N2. N1 received a value of 255, which was the last byte in the file. N2 received a value of zero since there were no more bytes in the file to be read.

Next, we will experiment with the PRINT@ and INPUT@ commands. Both are similar to their counterparts in the standard command set, the PRINT and INPUT statements. However, the PRINT@ and INPUT@ are better designed to work with disk files. The following examples will illustrate the resulting difference between the PRINT@ and INPUT@, and the standard PRINT and INPUT.

There is only one main difference for the PRINT@ command. Enter:

```
PRINT 100;"A":PRINT@2,100;"A"
```

to observe this difference. The resulting output is:

```
100 A
100A
```

You will note there is a space between the first 100 and the "A", which is absent between the second line. The PRINT@ command will not output a space after a number, where the standard PRINT statement does. The only other difference is that the PRINT@ command also sets the status variable, ST.

There are a number of differences between the INPUT and INPUT@ commands. Two of these differences involve which characters indicate the end of what should be read for each variable to receive data. Both INPUT@ and INPUT will accept a comma or a carriage return character as a terminator, indicating the end of data to be read for the next variable. However, the INPUT@ command will also interpret a space as a terminator when inputting into a numeric variable. Also, it is possible to NOT interpret a comma as a terminator, when inputting into a string variable.

To illustrate the difference when inputting into a numeric variable, enter and RUN the following 1 line program:

```
10 INPUT N:PRINT N
```

In response to the "?" prompt, enter:

```
123 456
```

You will note that the space between the "3" and the "4" was ignored. Now enter and RUN the one line program:

```
10 PRINT"? ";:INPUT@1,N:PRINT N
```

to observe the effect of the space with the INPUT@ command. In response to the "?" prompt, enter:

```
123 456
```

You may have been surprised that the input terminated when the space was entered. This is due to the space being interpreted as a terminator. Once the proper data was read, the data was converted to a number and stored into N. Since the INPUT@ command had no additional variables needing data, the INPUT@ command stopped at that point. The PRINT command then printed the value 123 for N.

This actually shows another difference as well. The fact that the inputting stopped as soon as the space was entered, indicates that the INPUT@ command reads the data a byte at a time. This is so the INPUT@ command will read the exact number of characters it should. This means that editing control functions are not operational when using the INPUT@ command. The standard INPUT command reads data a line at a time. Data for more than one variable can be obtained from the line, but it will discard any extra data the line may contain. This discarding of data will make the standard INPUT statement unsuitable for reading disk files in most cases.

As stated before, it is possible to not interpret a comma as a terminator when inputting into a string variable. Obviously, this allows you to input strings that contain commas. Selecting how commas should be interpreted is done by POKEing zero or 128 into location 1920. POKEing zero will select interpreting a comma as a

terminator. POKEing 128 will select not interpreting a comma as a terminator. When you first LIB in the "CIL", the location will be set to zero.

To illustrate this, enter and RUN the one line program:

```
10 POKE1920,128:PRINT"? ";:INPUT@1,A$:PRINT A$
```

In response to the "?" prompt, enter:

```
FIRST, SECOND
```

Note that the whole string was input into A$. Now enter and RUN the one line program:

```
10 POKE1920,0:PRINT"? ";:INPUT@1,A$:PRINT A$
```

In response to the "?" prompt, again enter:

```
FIRST, SECOND
```

As with the space, the inputting stopped when the comma was entered. This time only "FIRST" is input for A$.

There is a third difference in the terminators used by the INPUT and INPUT@ command. The standard INPUT command will interpret a colon as a terminator, where the INPUT@ command does not.

There a several other differences as well. First of all, interpretation of commas and colons as terminators can be suppressed in the standard input command by enclosing the characters within quotes. Quotes have no effect in the INPUT@ command. Also, if a non-numeric character is read while inputting data for a numeric variable, the standard command will issue a REDO FROM START message, and prompt for the data again. The INPUT@ command will interpret the first non-numeric character as the end of the number. Any leftover characters will be ignored. Another minor difference is that the INPUT@ command does not ignore leading spaces when inputting into a string variable, where the standard INPUT command does.

As a last difference, the INPUT@ has an extra feature which could prove very useful in certain situations. The INPUT@ has the ability to properly read strings longer than BASIC's input buffer, which is 192 characters long. Such a string may be read into two or more variables without dropping any characters.

Before this feature can be demonstrated, we must first write a string to disk which is longer than 192 characters. Enter:

```
SYSTEM "BEGINOF 6"
```

to position the DUMMYDATA.D file back to the beginning. Then enter:

```
A$="1234567890":FOR I=1 TO 20:PRINT@6,A$;:NEXT
```

to write a 200 character string, and enter:

```
TRUNC@6
```

to truncate the file at this point. To read back this long string, enter:

```
SYSTEM "BEGINOF 6"
```

to position the file back to the beginning again. Then enter and RUN the one line program:

```
10 INPUT@6,A$:IF ST>64 THEN INPUT@6,A1$
```

to read the long string into A$ and A1$. Now enter:

```
PRINT LEN(A$),A1$,ST
```

to see the results. You should note that the first 192 characters of the string were read into A$. The INPUT@ command will add 64 to the status variable value whenever a buffer overflow occurs during the command. By testing for ST greater than 64, we can determine whether there is more of the string to be read. The remainder of the string is then read into A1$. This final value of ST is -127, indicating that end-of-file was reached and that data for 1 variable was read by the last INPUT@ command.

You may be aware that the standard INPUT command can't be executed as a direct command, where the INPUT@ can. The problem with executing them as direct commands is that the input data overwrites the command while it is executing from BASIC's input buffer. The standard INPUT command deals with this by not allowing you to execute it as a direct command. The INPUT@ is a little more intelligent. It will input the first variable, then abort the rest of the direct command.

In the last sequence of examples, we will experiment with the BLKRD@ and BLKWRT@ commands. An important characteristic of the BLKRD@ and BLKWRT@ commands is that they transfer their data directly to and from memory. The location in memory to transfer the data is given by specifying the variable name whose contents are to be read or written. A restriction imposed by this is that the variable specified must exit prior to use in a BLKRD@ or BLKWRT@ command. This means that the variable must have been used, and thus created, prior to use in the BLKRD@ or BLKWRT@ command.

For the first example we will write some data. Enter and RUN the following program:

```
10 SYSTEM "BEGINOF 6"
20 A=200.67: B=3.5E12: N%=50: D$="ABC"
30 DIM AR(1,1): AR(0,0)=1: AR(1,0)=2: AR(0,1)=3: AR(1,1)=4
40 BLKWRT@6,A,B,N%,D$,AR(0,0),AR(1,0),AR(0,1),AR(1,1)
```

The BLKWRT@ command in line 40 above wrote the contents of each of the variables specified in the command. Also, it writes them in the order of appearance in the command. This is important because to read the data back properly, the data must be read into a variable of the same type from which it was written. The type refers to whether the variable is a floating point, integer, or string variable. Data written from a floating point variable must be read into a floating point variable, etc. In our case, we wrote two floating point numbers, followed by an integer, then a string, then four more floating point numbers.

To make the BLKRD@ and BLKWRT@ easier to use, you are permitted to specify a range of variables to be read or written. This is done by specifying a starting and ending variable. You may specify a range (or block) of non-array variables, or a range or array variables. A range is not permitted to cross from a non-array to an array variable, nor accross array variables of different arrays. As an example of specifying a range, line 40 above could have been written:

```
40 BLKWRT@6, A TO D$, AR(0,0) TO AR(1,1)
```

To illustrate the BLKRD@ command, we can read back the data just written. Enter and RUN the following program:

```
10 SYSTEM "BEGINOF 6"
20 X=0:Y=0:I%=0:A$="":DIM B(1,1)
30 BLKRD@ 6, X TO A$, B(0,0) TO B(1,1)
40 PRINT X,Y,I%,A$,B(0,0),B(1,0),B(0,1),B(1,1)
```

You should note that even though different variable names are used, they match in type to the variables from which the data is written. The groups of variables read or written by the BLKRD@ and BLKWRT@ commands are collectively called "block variables". For additional details on block variables, refer to section 6.

The BLKRD@ and BLKWRT@ are more powerful and faster than the other data transfer commands. However, they require a little more thought and effort to use. You must be very aware of how the data was written in order to read it back properly.

This concludes the getting started section. You should now enter:

```
SYSTEM "FREE 6"
```

to free the channel we have been using, and:

```
SYSTEM "DELETE DUMMYDATA.D"
```

if you wish to delete the data file we were using.

As one final note, it is fairly easy to forget that while in BASIC, the default extension is "B" instead of "C". This is done to better support the loading and saving of BASIC programs. There is one place where forgeting this will cause unexpected errors. This is when you use the SYSTEM command to execute utility programs. For example, the command

```
SYSTEM "DIR *.B"
```

results in a COMMAND NOT FOUND error because there is no DIR.B file on disk. To get the directory, the command must be:

```
SYSTEM "DIR.C *.B"
```

Also, it is recommended that you use an extension other than "B" for programs which are in ASCII format (i.e. LISTed or entered using the EDITOR). We recommend an extension of "E" for these programs.

4. FILE NAMES AND COMMAND STRINGS

In the syntax diagrams given for the SYSTEM and APPEND commands, the symbols <file name> and <command string> are used to indicate that a file name or command string is to be specified. This file name or command string may be specified as a single string or as a parameter list with the following syntax.

<parm> [,<parm>]...

where <parm> may be a string or number, constant, variable, or expression. If a number is used, it is automatically converted to a string (positive numbers will have a preceeding space). The actual file name or command string used will consist of a concatenation of the strings associated with each parameter in the list. For example, given that NM$="MYFILE", CN=3, and DN=1, here are some examples and their resulting file name or command string.

PARAMETERS	SEEN BY CODOS
NM$,":",DN	"MYFILE: 1"
NM$,".C:",DN	"MYFILE.C: 1"
"ASSIGN",CN+2," ",NM$	"ASSIGN 5 MYFILE"
"FREE",3	"FREE 3"

5. THE STATUS OF THE ST VARIABLE

MTU BASIC has a reserved variable called ST which is used to obtain the completion status of various I/O operations. Most of the commands in the CIL Library set this status variable upon completion of the command. For commands which read or write data, ST is set to the number data items read or written. 128 will be subtracted from this value if EOF is encoutered. In other commands, ST may be set differently or not changed. This will be indicated in the discussion for each command.

6. BLOCK VARIABLES

The BLKRD@ and BLKWRT@ commands read and write groups of data items called "block variables". Basically, a block variable is the contents of memory between a specified beginning point and a specified ending point. These beginning and ending points are identified by the variable names that are physically stored in memory at these points.

Thus a block variable will contain one or two variable names which have the effect of designating the area of memory to be read or written. An important restriction is that the variables contained in a block variable must exist prior to using the block variable. For non-array variables, this means the variable must appear in some previously executed statement since variables are created as they are encounted during execution. Sometimes they will need to appear in a previously executed statement only for the purpose of creating them. In the case of array variables, they must be dimensioned prior to use in a block variable. In a way, this has the effect of requiring block variables to be declared prior to use.

In the following discussion it is important to realize that it makes no difference how data is written to a file. For example, 10 variables that are stored sequentially in memory could be written to a file one at a time in sequence, in two groups of 5, or all at once with the same results just as long as the writing sequence is the same in each case.

6.1 SINGLE VARIABLE BLOCK

The Single Variable Block consists of a single data item specified by name of the variable. This data item may be a standard floating point variable, an integer variable, a particular array element, or a string variable. Simple variables must have been used in the program prior to being used in the BLKRD@ or BLKWRT@ commands so that they indeed exist in memory. In the case of array elements, the array must have been previously DIMensioned. The following are some examples of Single Variable Blocks:

```
A                    B(I,J)
IJ%                  N$
```

6.2 SIMPLE RANGE BLOCK

The Simple Range Block is a group of non-array data items specified by naming a starting and ending variable. The block will consist of the starting and ending data items named, plus all those stored in memory in between. The data items may be standard floating point variables, integer variables, or string variables (arrays are discussed below). Here are some examples of Simple Range Blocks:

```
A TO T
N% TO B$
X$ TO YY
```

As you can see from the examples, the types of variables (integer, floating point, or string) may be freely mixed in a Simple Range Block. It is important to notice that there is no way to tell how much data is included in the Simple Range Block by looking at the block variable. Therefore, it would be wise to explicitly create each Simple Range Block early in a program. For example, a statement similar to the following one might appear in your program:

```
10 A=0:B=0:NM$="":I%=0:T=0 :REM CREATE BLOCK A TO T
```

The order of the variable definition in the previous example statement is important because when the data is read back, it will have to be into a Simple Variable Block with the same order of data types, i.e., float, float, string, integer, float.

6.3 ARRAY RANGE BLOCK

The third type of block variable is the Array Range Block. In this type of block, a group of array data items is specified by naming a starting and ending array element. Two additional restrictions are that the group of data items must be part of the same array, and the starting data item must preceed the ending data item in memory. This second restriction simply means that the starting array item should be the one with the lower subscripts. For multi-dimensional arrays, the rightmost subscript is the most "significant". The following examples indicate valid Array Range Blocks:

```
A(0) TO A(100)
C1(0,0,0) TO C1(10,10,10)
B(2,1) TO B(1,20)
A$(1,2) TO A$(20,2)
C(10,1,1) TO C(1,1,10)
B%(10,1,1) TO B%(100,1,1)
```

6.4 THE BLOCK ARGUMENT ERROR

In the use of BLKRD@ and BLKWRT@ commands, you will eventually encounter the "BLOCK ARGUMENT ERROR". This is due a block variable violating one of the restrictions. The first restiction that could be violated is that each variable used in the command must already exist. Using the wrong variable name is the easiest way of violating this restriction.

The second restriction is that the first variable in a range block actually specify the beginning of a range. Should the beginning variable specified follow the ending variable in memory, then the range is invalid. The important fact to remember is that storage locations are assigned to variables starting with low memory and go up. The order in which the locations are assigned to variables is the order in which the variables are encoutered during execution. If the statement:

```
10 A=0:B=1000:X$="ABC"
```

was encountered as the first line of a program, storage locations would be first assigned to A, then B, and finally X$. Thus the variable A would be found in memory before X$. This means that "A TO X$" would be a valid simple range block, where "X$ TO A" would not. This problem can also occur with array variable names in an array range block. The order of occurence in memory for an example array named A dimensioned by DIM A(10,10,10) is:

```
A(0,0,0), A(1,0,0), A(2,0,0), ... , A(0,1,0), A(1,1,0), ... , A(10,10,10)
```

Thus "A(10,0,0) TO A(6,1,1) is a valid array range block, where "A(5,10,10) TO A(6,1,1,)" is not.

A final restriction is that an array range block must not cross from one array to another. Thus "A(10,1) TO B(1,1)" is invalid.

7. COMMAND DESCRIPTIONS

The following are descriptions for each of the commands in the CODOS Interface Library. You will note that many of them end with the "@" character. This indicates a Channel Number argument is expected after the the command name.

7.1 APPEND

PURPOSE: To append a BASIC program on disk to the program in memory.

SYNTAX: APPEND <file name>

ARGUMENTS:

<file name> = a file name string as described in section 3.

EXAMPLES:

```
APPEND "GRAPHPLOT"
```

will append the BASIC program "GRAPHPLOT.B" on the default drive to the program already in memory.

NOTES:

1. The APPEND command can only add a program to the end of the one in memory.

2. The BASIC program to be appended must have been written with the SAVE command.

3. Before the program is appended, a check is made to insure that the starting line number of the specified program is greater than the last line number of the program in memory. If not, the APPEND command is aborted with an error message. This check is done to insure that the line numbers of combined programs will be sequential.

4. Execution of the APPEND command has no effect on the status variable ST.

7.2 BLKWRT@

PURPOSE: To write data from one of more block variables to disk.

SYNTAX: BLKWRT@ <channel> , <block variable> [, <block variable>] ...

ARGUMENTS:

<channel> = an expression which evaluates to the desired channel.

<block variable> = a block variable as defined in section 4.

DISCUSSION:

The BLKWRT@ command is intended to help simplify the writing of groups of variables, thereby providing a form of record I/O. Also, the transfer rate with the BLKWRT@ command will tend to be faster than the other data output commands. Advice on maximizing the throughput of BLKWRT@ is given in Appendix A.

EXAMPLES:

Given that the statement

```
A=0:B=0:NM$="":I%=0:T=0 :REM CREATE BLOCK A TO T
```

creates the variables it contains, the following examples will operate as described:

```
BLKWRT@ 3,A,NM$
```

will write the floating point number in A, followed by the string in NM$ to channel 3.

```
BLKWRT@ 3,A TO I%
```

will write the floating point number in A, the floating point number in B, the string in NM$, and the integer in I% to channel 3.

```
BLKWRT@ CN,X(1)TO X(20)
```

will write 20 floating point numbers from X(1) through X(20) to the channel specified by CN.

NOTES:

1. Each floating point variable causes 5 bytes to be written. Each integer causes 2 bytes to be written. Each string variable causes a one byte length to be written followed by the indicated number of characters. Note that if a block variable doesn't contain any string variables, the number of bytes written by the block variable is fixed.

2. The status variable ST is set to the number data items written. 128 is subtracted from ST is End-of-File is reached or being extended.

7.3 BLKRD@

PURPOSE: To read data from disk into one or more block variables.

SYNTAX: BLKRD@ <channel> , <block variable> [, <block variable>] ...

ARGUMENTS:

<channel> = an expression which evaluates to the desired channel.

<block variable> = a block variable as defined in section 4.

DISCUSSION:

The BLKRD@ command is intended to help simplify the reading of groups of variables, thereby providing a form of record I/O. Also, the transfer rate with the BLKRD@ command will tend to be faster than the other data input commands. Advice on maximizing the throughput of the BLKRD@ command is given in Appendix A.

EXAMPLES:

Given that the statement

```
A=0: B=0: NM$="": I%=0: T=0: REM CREATE BLOCK A TO T
```

creates the variables it contains, the following examples will operate as described:

```
BLKRD@ 3,A,NM$
```

will read a floating point number into variable A, and a string into NM$ from channel 3.

```
BLKRD@ 3,A TO I%
```

will read a floating point number into A, a floating point number into B, a string into NM$, and an integer into I% from channel 3.

```
BLKRD@ CN,X(1)TO X(20)
```

will read 20 floating point numbers into X(1) through X(20) from the channel specified by CN.

NOTES:

1. In order for this command to transfer data properly, you MUST READ THE DATA EXACTLY AS IT WAS WRITTEN IN REFERENCE TO TYPE!!! Here type refers to whether it is floating point, string, or integer. In other words, floating point numbers must be read into floating point variables, integers into integer variables and strings into strings. This is necessary because the type of the variable being read into indicates to CIL how many bytes to read from the file.

2. Each floating point variable causes 5 bytes to be read. Each integer causes 2 bytes to be read. Each string variable causes a one byte length to be read followed by the indicated number of characters.

3. If End-of-file is encountered during execution of the BLKRD command, the remaining variables will be left unchanged.

4. The status variable ST is set to the number data items read. 128 is subtracted from ST if End-of-File is encoutered.

7.4 ENTER@

PURPOSE: To enter ASCII program lines or data from a channel.

SYNTAX: ENTER@ <channel>

AGRUMENTS:

<channel> = an expression which evaluates to the desired channel.

DISCUSSION:

This command simply redefines the input channel for MTU BASIC to the channel specified in the command. The ENTER command found in MTU BASIC allows you to declare the file or device to use as the input source, but not which channel to use; it always uses channel zero. The CIL ENTER@ command allows you to define the channel, instead of the file or device. You should refer to the MTU BASIC Manual for information about the ENTER command.

EXAMPLES:

```
ENTER@ 3
```

will cause BASIC's input channel to be set to channel 3.

NOTES:

1. BASIC's input channel will remain set to the selected channel until the End-of-File is encountered. At this point, the input channel is reset to 1.

2. Executing this command has no effect on the status variable, ST.

7.5 GET@

PURPOSE: To read a byte from a channel (often assigned to a disk file).

SYNTAX: GET@ <channel> , <variable> [, <variable>] ...

ARGUMENTS:

<channel> = an expression which evaluates to the desired channel.

<variable> = a BASIC variable which is to receive the byte.

DISCUSSION:

The GET@ command can read bytes from any input device or disk file if assigned to the selected channel. The GET@ command differs substantially from the standard GET command when the variable is numeric. The GET@ command will place the actual byte value in the numeric variable. For string variables, the byte becomes a one character string.

EXAMPLES:

Given that the value of the next byte to be read on channel 4 is 65 (for the Console this is equivalent to hitting the "A" key), then the following examples will operate as described:

```
GET@4,B
```

will cause the value of B to become 65.

```
GET@ 4,A$
```

will cause A$ to become "A".

```
GET@2 R(1),R(2),R(3),R(4)
```

will cause R(1) to become 65 and R(2), R(3), R(4) to become whatever the next 3 bytes read from channel 2 are equal to.

NOTES:

1. If the channel is assigned to a file which is at End-of-File, the GET@ command will return the value 0 for numeric variables and a null string for string variables.

2. The status variable ST is set to the number data items read. 128 is subtracted from ST if End-of-File is encountered.

7.6 INPUT@

PURPOSE: To input ASCII data from a channel (often assigned to a file on disk).

SYNTAX: INPUT@ <channel> , <variable> [, <variable>] ...

ARGUMENTS:

<channel> = an expression which evaluates to the desired channel.

<variable> = a BASIC variable which is to receive the data.

DISCUSSION:

The INPUT@ command can perform all of the same functions as the standard INPUT command, plus a few more. When inputting numbers, the INPUT@ command will skip any leading spaces, then treat a trailing space as the end of the number. When inputting strings, leading blanks are read into the string. Also, a flag is available to specify if a comma is to be interpreted as part of the string, or the end of the string.

EXAMPLES:

Given that "155 200, TEXT" are the next characters to come from channel 3, and CN=3, then the following examples will operate as described:

```
INPUT@ 3,N,I,S$
```

will result in setting N to 155, I to 200, and S$ to " TEXT".

```
INPUT@ CN,A$,B$
```

will result in setting A$ to "155 200" and B$ to " TEXT".

```
POKE 1920,128 :INPUT@ CN,A$
```

will result in setting A$ to "155 200, TEXT". See NOTE 3 for details about the POKE.

NOTES:

1. The INPUT@ command will read ASCII data from the specified channel until a termination character is encountered. That data is then converted as necessary and stored in the specified variable.

2. Termination characters are defined as follows:

A. A carriage return is always interpreted as a termination character.

B. A comma is always a termination character when inputting into a numeric variable.

C. A comma may or may not be a terminator when inputting into a string variable, as determined by a flag (see NOTE 3).

D. A space will be interpreted as a terminator when inputting into a numeric variable if it follows a non-blank character.

3. When inputting into a numeric variable, the characters read are not checked. If none of the characters read are numeric, the value returned is zero.

3. The flag found at location 1920 decimal (780 hex) controls how commas are treated when inputting into a string variable. If set to zero, a comma will be interpreted as a termination character. If set to 128, the comma will be placed in the string and the input continued.

4. Reaching the End-of-File will always cause input to terminate.

5. The INPUT command uses the BASIC Input Buffer as its buffer while reading the data. This buffer is 192 characters long. Should the buffer become full while inputting a string, the INPUT command will read one additional character to see if it is a terminator. If it isn't, this character is saved and 64 is added to the status variable ST. The saved character will be read as the first character for the next variable which is input. This may occur in the current INPUT command, or a later. If you want to discard this saved character, assign a channel to the null device and do an input from that channel.

6. Another feature of the INPUT@ command is that it may be executed as a direct command provided only one input variable is specified. The reason only one is allowed is that the direct command is located in BASIC's input buffer while it is being executed. Extra input variables could be overwritten by the input data for the first.

7. The INPUT@ command always returns a zero or null string in response to a null input. A null input is when the first character read is a terminating character.

8. The status variable ST is set to the number data items read or written. 128 will be subtracted from this value if End-of-File is encoutered. 64 will be added if data for one of the input variables overflows the input buffer.

7.7 LIST@

PURPOSE:

SYNTAX: LIST@ <channel> [, [<first line>] [- [<last line>]]]

AGRUMENTS:

<channel> = any expression which evaluates to the desired channel.

<first line>= the number of the first line to list. Defaults to the first line of the BASIC program.

<last line>= the number of the last line to list. Defaults to the last line of the BASIC program.

EXAMPLES:

```
LIST@ 3
```

will list all of the BASIC program in memory to channel 3.

```
LIST@ 3,100-
```

will list to channel 3 starting with line 100 and continuing to the end of the BASIC program.

```
LIST@ 4,-300
```

will list from the beginning of the BASIC program up to and including line 300 to channel 4.

NOTES:

1. Executing the LIST@ command has no effect on the status variable ST.

7.8 ONERR

PURPOSE: To allow a BASIC program to do its own CODOS error handling.

SYNTAX: ONERR { <statements> / OFF }

ARGUMENTS:

<statements> = one or more BASIC statements separated by colons.
OFF = if specified as the parameter, then the ONERR handling is turned off and normal CODOS error handling is restored.

DISCUSSION:

The purpose of the ONERR command is to allow a BASIC program to do its own handling of CODOS errors if this is desired. Normally a CODOS error will print an error message and halt the program. If the user of a program was not the author, he may not know how to restart the program without loss of data should a CODOS error occur. In cases where this must be avoided, the ONERR command can be used to let the BASIC program do the error handling.

The ONERR command operates as follows when the parameter consists of one or more BASIC statements. When executed with no CODOS error pending, the ONERR command causes the current line number to be saved as the Error Line Number. Next, the BASIC statements following the ONERR command are skipped over and execution of the BASIC program continues as usual. If a CODOS error occurs after this point, all GOSUBs and FOR/NEXT loops are exited. Execution then jumps the program line identified by the Error Line Number. This line necessarily contains an ONERR command. However, because there is a pending CODOS error, the BASIC statements after the ONERR command are executed. The ST variable will contain the standard CODOS error number which the program can use to identify the error.

EXAMPLES:

```
1000 ONERR GOTO 9000
```

will establish line 1000 as the Error Line Number. On the occurrence of a CODOS error, the GOTO 9000 will be executed.

```
2900 ONERR GOSUB 4000:IF FL = ER THEN PRINT "CODOS ERROR ";ST :STOP
```

will establish line 2900 as the Error Line Number. On the occurrence of a CODOS error, the GOSUB 4000 will be executed. If FL is not equal to ER (presumably set by the subroutine at 4000) then the program will contine on the next line after 2900. If FL is equal to ER, then then "CODOS ERROR ", followed by the CODOS error number is printed and the program halts.

```
1776 ONERR OFF
```

will restore normal CODOS error handling.

NOTE:

1. Because all GOSUBs and FOR/NEXTs are exited when a CODOS error occurs, the ONERR command should not be executed within subroutines or FOR/NEXT loops.
2. See the CODOS manual appendix A for a list of error numbers and their corresponding error conditions.
3. ONERR handling is automatically turned off if you execute a LOAD, RUN, or FRELIB command.

7.9 POSITION@

PURPOSE: To position the file assigned to a channel.

SYNTAX: POSITION@ <channel> , <position>

AGRUMENTS:

<channel> = an expression which evaluates to the desired channel.

<position> = an expression which evaluates to the desired file position.

DISCUSSION:

CODOS files are organized as abitrarily large one-dimensional arrays of bytes. A position pointer is maintained for each assigned file which identifies the next byte or bytes to be read or written. The POSITION@ command sets the position pointer for the file assigned to the specified channel.

EXAMPLES:

Given that the file assigned to channel 3 begins with the sequence of characters "ABCDEF", then the following examples will operate as described:

```
POSITION@ 3,0
```

will position the file to the very beginning (character number 0). The next character read will be "A".

```
POSITION@ 3,4
```

will position the file to point to character number 4. The next character read will be "E".

```
POSITION@ CN,RN*RL
```

will postion the file assigned to channel CN to the beginning of record RN, assuming each record in the file is RL bytes long. Note that any record or field structure to the file is entirely up to the BASIC programmer; CODOS considers all files to merely be a long string of bytes.

NOTES:

1. If the file position specified is beyond the end-of-file, then the file is positioned at end-of-file.

2. The status variable ST is set to -128 if the file is positioned to end-of-file, Otherwise it is set to 0.

3. Positioning a file at random takes at most one disk access per position command.

4. File lengths may easily become much longer than 64K bytes thus necessitating the use of a floating point number for the position argument. Therefore you must be careful to avoid round-off error in your position calculations. Round-off can usually be avoided by never allowing fractional factors to enter into your file position calculations.

7.10 POSN() FUNCTION

PURPOSE: To determine the current position of a file assigned to a channel.

SYNTAX: POSN(<channel>)

AGRUMENTS:

<channel> = an expression which evaluates to the desired channel number.

DISCUSSION:

CODOS files are organized as abitrarily large one dimesional arrays of bytes. A position pointer is maintained for each assigned file which controls the next byte or bytes to be read or written. The POSN() function returns the value of the position pointer for the file assigned to the specified channel. It is important to remember that this is a function instead of a command. The POSN() function must be used as part of an expression.

EXAMPLES:

If the file assigned to channel 5 is positioned at the beginning of the file, then

```
FP=POSN(5)
```

will set the variable FP to 0.

If the file assigned to channel 6 starts with the sequence of characters "ABCDEF", and the "C" had just been read by another command, then:

```
FP=POSN(6)
```

will set FP to 3 indicating that character number 3 (the "C") had just been read.

```
RN=POSN(CN)/RL
```

will set RN to the next record number of the file assigned to channel CN, assuming all of the records in the file are RL bytes long.

NOTES:

1. The value returned by the POSN() function can exceed the 32767 maximum allowed for integers. Therefore floating point variables should generally be used to receive the returned position pointer value, though integer variables are permitted for small files.

2. The status variable ST is set to -128 if the file position returned represents End-of-File, otherwise, ST is set to 0.

7.11 PRINT@

PURPOSE: To output ASCII data to a channel (often assigned to a file on disk).

SYNTAX: PRINT@ <channel> [, <expression> [{,} [<expression>]] ...]
                                          {;}

AGRUMENTS:

<channel> = an expression which evaluates to the desired channel.

<expression> = an expression which evaluates to the data you wish to output.

DISCUSSION:

The PRINT@ command operates almost identically to the standard MTU BASIC PRINT command. The comma, semicolon, TAB() function, and SPC() function have the same effects as they do in the standard PRINT command. The only two differences are that PRINT@ command does not add a space to the end of numeric data, and does set the ST variable.

EXAMPLES:

Given that N = 253 and N$ = "ABC", the following examples will operate as described:

```
PRINT@ 3,100,N$
```

will output " 100",6 spaces for the comma, "ABC", and a carriage return to channel 3.

```
PRINT@ 3,N;N$;
```

will output " 253ABC" to channel 3.

```
PRINT@2,N;TAB(20);N$;TAB(40);
```

will output " 253",16 spaces to tab to column 21, "ABC", and finally 17 spaces to tab to column 41. This example will always output 40 bytes of characters.

NOTES:

1. The comma will cause spaces to be output until the next tab stop is reached. The tab stops for the PRINT@ command are fixed at every ten spaces.

2. The TAB() function will cause the PRINT@ command to output spaces until the character count for that PRINT@ command equals the TAB parameter. If the character count already equals or exceeds the TAB parameter, the TAB function is ignored. The TAB function can be used to force fixed amounts of data to be written even through the numbers or strings printed may vary in length.

3. The SPC() function will output the number of spaces indicated by its argument.

4. The status variable ST is set to the number data items written. 128 will be subtracted from this value if End-of File is reached or extended while writing.

7.12 PUT@

PURPOSE: To output a byte to a channel (often assigned to a disk file).

SYNTAX: PUT@ <channel> , <expression> [, <expression>] ...

ARGUMENTS:

<channel> = an expression which evaluates to the desired channel.

<expression> = a numeric expression which evaluates to the byte to be output.

DISCUSSION:

The PUT@ command provides a means of writing files on a byte by byte basis. This command in combination with the GET@ command guarantees total control over the contents of any CODOS file.

EXAMPLES:

```
PUT@4,65
```

will output the byte 65 (41 hex, character "A") to channel 4.

```
PUT@ 4,ASC("X"),N
```

will output the byte 88 (58 hex) followed by the byte in N to channel 4.

NOTES:

1. The status variable ST is set to the number data items written. 128 is subtracted from ST if End-of-File is reached or written over.

2. An ILLEGAL QUANTITY ERROR is given if the number to be output is outside the range of 0 through 255.

7.13 SYSTEM

PURPOSE: To execute any CODOS command from MTU BASIC.

SYNTAX: SYSTEM <command string>

AGRUMENTS:

<command string> = a command string as described in section 2.0

DISCUSSION:

The SYSTEM command is used to pass commands to the CODOS operating system for execution. Any CODOS Monitor command may be executed. Any CODOS User command may also be executed provided it does not conflict with memory used by MTU BASIC or the CODOS operating system. Using the CODOS Utility area (B400 to BDFF hex) is permitted. If you are using CIL.Z, then may execute any utility program which does not use the Large Transient Buffer (A000 to B3FF hex). This excludes the COPYF and FORMAT utilities. If you are using CILB.Z, then this last restriction does not apply.

EXAMPLES:

```
SYSTEM "ASSIGN 3 MYDATA.D"
```

will assign channel 3 to the file MYDATA.D on the default drive.

```
SYSTEM "FREE",CN
```

will free the channel specified by CN.

```
SYSTEM "DIR.C *.B:1"
```

will perform a directory listing of all files on drive 1 with the extension ".B".

NOTES:

1. When executing utility programs with the SYSTEM command, you must specify the ".C" on the utiltity name. Remember, the default extension is changed to ".B" when running MTU BASIC.

2. The status variable will be set to the current value of the file status byte after each CODOS command. This is useful only when an "ASSIGN" command is passed as a CODOS Monitor command to be executed. The status variable will set according to the table below:

ST VALUE	CHANNEL STATUS
0	Assigned to a device
64	New file created and assigned
-64	Assigned to an existing file
-32	Assigned to an existing locked file

7.14 TRUNC@

PURPOSE: To truncate the file assigned to a channel at its current position.

SYNTAX: TRUNC@ <channel>

AGRUMENTS:

<channel> = an expression which evaluates to the desired channel.

DISCUSSION:

The TRUNC@ command provides an easy means of truncating a file to a shorter length such as after being re-written.

EXAMPLE:

```
TRUNC@ 3
```

will truncate the file assigned to channel 3 at its current position.

NOTES:

1. The TRUNC@ command has no effect if the channel specified is assigned to a device.

2. Since the current position will become the new End-of-File, the status variable ST is set to -128 to indicate End-of-File.

APPENDIX A.

COMMAND SUMMARY

CODOS INTERFACE LIBRARY - CIL

This enhancement library adds a complete set of CODOS disk operating system commands to MTU BASIC. The following is a list of the commands:

APPEND <filename>	Append program on disk to program in memory
BLKWRT@ <chan>,<blkvar>[,<blkvar>] ...	Write BASIC variable memory image to a channel
BLKRD@ <chan>,<blkvar>[,<blkvar>] ...	Read channel into BASIC variable memory image
ENTER@ <chan>	Enter ASCII program lines or data from channel
GET@ <chan>,<var>[,<var>] ...	Input single bytes from a channel
INPUT@ <chan>,<var>[,<var>] ...	Read ASCII data into variables
LIST@ <chan>,<firstline>-<lastline>	List BASIC program to a channel
ONERR { <statements> / OFF }	Allow or discontinue user error handling
POSITION@ <chan>,<position>	To position a file assigned to a channel
POSN (<chan>)	Determine position of file assigned to channel
PRINT@ <chan>,<expr>[,<expr>] ...	Output ASCII data to a channel
PUT@ <chan>,<expr>[,<expr>] ...	Output single bytes to a channel
SYSTEM <command>	Execute any CODOS command
TRUNC@ <chan>	Truncate a file assigned to a channel

The parameters used above are defined as follows:

<filename> File name string as described in section 3.

<chan> An expression that evaluates to a legal CODOS channel number.

<blkvar> A block variable specification as described in section 5.

<var> A BASIC variable either floating point, integer, string, or array element.

<firstline>
<lastline> The line number of a BASIC statement.

<statements> One or more BASIC statements concatenated into a single line.

<position> An expression that evaluates to a legal CODOS file position.

<expr> A numerical or string expression that evaluates to between 0 and 255.

<command> A command string as described in section 3.