Working With Files

BASIC allows you to create, write and read files on disk, any source that can be mapped to a drive letter, or any network shared file to which you have access rights. It also supports reading from and writing to external devices via a serial or USB port (see the section on **SERIAL** files, below) and BINARY files (see the section on **BINARY** files, below).

Specifying File Location and Name.

The file specification is given in the **OPEN** statement. The format is

> **OPEN** *file-mode, file-num,* ***file-specification****, record-size*
>
> The ***file-specification*** is a string expression which must evaluate to either be a file name or a complete or partial path.
>
> > **Note**: For SERIAL files, the file specification is the name of the COMM port (e.g., COM1). See the section on SERIAL files, below.
>
> If it is a name or partially-qualified name (*partial-path* \ *filename*), then it will be appended to the **Current Working Directory**.
>
> > In GUI mode, the current working directory is shown in the "**Working Directory**" box at the top of the screen (whether or not "Use Working Directory" is checked.) Normally (but not always) this is the path from which the BASIC program is loaded. (See description of the **LOAD** and **RUN** commands).
> >
> > In CONSOLE mode, the current working directory is that which was in effect when QUICKCALC was started. You may override this with the **CHDIR** statement.
>
> The file specification must be a **string expression**. You can specify this in several ways, just like specifying file paths\names in DOS or Windows, *e.g.,*
>
> > "*filename.ext*"
> > "**..***filename.ext*"
> > "*partial-path******filename.ext*"
> > "c:*folder******sub-folder******filename.ext*"
> > *savedpath*$ + "\\" + *filename*$
> > "*network-computer******shared-folder******sub-folders******filename.ext*"
>
> **Note:** String constants must be contained in quotes. The backslash (\ ) character is specified as a double-backslash in quoted string constants. It is converted back into a single backslash when the expression is evaluated.

If a full path is not specified, the file name or partial path is appended to the **current working directory** in order to create the complete file specification.

If you wish to enter the file name at run time, you may issue an **INPUT** statement and type it in at run time. Assign the input string to a string variable and use that in the input statement, *e.g.,*

INPUT *filename$*
OPEN INPUT, 2, *filename$*

In GUI mode, you may use the function **BROWSEINPUTFILE$** or **BROWSEOUTPUTFILE$** to open a dialog to select the file from anywhere on your computer or network. These functions return a string containing a full path which you can use in an **OPEN** statement, for example:

OPEN (**INPUT**, 2, **BROWSEINPUTFILE$** ("File to Read")
OPEN (**OUTPUT**, 3, **BROWSEOUTPUTFILE$** ("File to Write")

Note: The browse functions start browsing in the current working directory. If you want the browse to start in another directory, you can use the **CHDIR** statement to set it. You may want to save the current working directory first (from **WORKINGDIR$**) so that you can restore it later.

Note: Once a file has been opened, you can determine its length (size in bytes) with the **FILESIZE** (*file-num*) function.

SEQUENTIAL and RANDOM ACCESS Files.

Sequential Files.

A Sequential file is a stream of text characters. You create a sequential file with the **OPEN OUTPUT, ...** or the **OPEN APPEND, ...** command. **OPEN APPEND, ...** re-opens an existing file so that you can add records to the end of it.

Sequential files are always read or written sequentially. Records (or lines) follow each other in the file. At the end of each line, the system puts in a carriage-return (hex '**0D**') and a line-feed (hex '**0A**'). This is done automatically when you complete writing a line.

Writing to a Sequential File.

You write to a sequential file with the **PRINT #** *file-num*, **PRINT #** *file-num* **; USING**, or **WRITE #** *file-num* statements.

With the **PRINT** statements, you can format the data however you like, but it will not necessarily be in a format that can be **INPUT** and decoded into variables. **PRINT** is useful when the file is going to be read by a person or printed later.

The **WRITE** statement is used when you want to be able to re-read the file later using the **INPUT #***file-num* statement. The data values written are separated by commas and string values are enclosed in quotes. **LONGMATH** numbers are written in a special format which will allow them to be re-input (see "Working with Long Numbers"). Records created with the **WRITE #** statement are not formatted for easy reading, but are in the correct format for the **INPUT #***file-num* statement.

With both the **PRINT** and **WRITE** statements, if the statement ends in a semicolon, the line is not terminated, but may be continued with additional **PRINT** or **WRITE** statements. The line is terminated and output when a **PRINT** or **WRITE** statement appears that does not end in a semicolon, or when the maximum line length is reached.

> **Note:** **PRINT** and **WRITE** statements may be intermixed in the same output line.

When the line is complete, a c/r and l-f are appended to the line and it is written to the file. The line-feed (or "new-line") character marks the end of the line. If no data is written, *e.g,* **PRINT #***file-num* with no variables, there will still be a c/r l-f pair, indicating a zero-length record.

The null character (hex '00') is not used in sequential files, and should be avoided, since some programs use it to mark the end of the file. Immediately following the line-feed is the first character of the next record (or the end of the file.) There is no null character at the end of the file.

The output file must be closed before it can be re-opened for input.
Use **CLOSE** *file-num* or simply **CLOSE** to close all open files.
All files are closed automatically when a program terminates, except when **CHAIN**-ing to another program .

Reading from a Sequential File.

Once the file has been opened for INPUT, reading begins at the beginning of the file. There are two ways to read data: **INPUT #** and **LINE INPUT #.**

INPUT #*file-num; list-of-variables* reads data from the file and assigns the data to variables.

> The data read must be compatible with the variable type to which it will be assigned. May contain numbers or strings (quoted or unquoted), separated by commas. Unquoted strings will have blanks and tab characters stripped off

front and back when they are read, and may not contain commas, quotes, or start with a digit, sign, or decimal point (*i.e.*, must not be confused with a number).

If the end of a line is reached before the list-of-variables is exhausted, the next record is read and variables continue to be assigned from the new record. Blanks and c/r and l-f characters are ignored. The file is scanned until the next data is encountered.

If the end of the file is reached before all the variables are assigned, an end-of-data error will occur. If you want to avoid these errors, you can use the **EOF** (*file-num*) to detect end-of-file before reading, or the **INPUT** (*file-num*) function, which will tell you the type of variable coming up so that you can avoid type mismatch errors.

LINE INPUT # *file-num, string-variable* reads the entire line (if possible) into a string variable.

If the line has been partially read (by an **INPUT #** or **LINE INPUT #** statement), the **LINE INPUT #** will read the remainder of the line (if possible).

All characters are read, including spaces and non-printable characters, up to the c/r l-f sequence, which are not included in the resulting string.

Note: If a line-feed or null character is embedded in a record, it will terminate the input line prematurely. You should avoid placing these characters in your data files.

If the remainder of the line is longer than 255 characters, the next 255 characters will be returned. The next **LINE INPUT #** (or **INPUT #**) will continue reading from that point.

LINE INPUT # will not read beyond the end of the record. If the record contains no characters, a zero-length string (null string) is returned.

If a **LINE INPUT #** is executed at the end of the file, an end-of-file error will occur. To avoid this kind of error, you can use the **EOF** (*file-num*, "**LINE**") function. This function does not skip over blanks or zero-length records.

Random-Access Files.

Basically, a RANDOM file is a file containing a number of records, all of which are the same length. When a file is opened for RANDOM access, you can both read and write to it, thus allowing you to access records in any order (not necessarily sequential).

The record length must be specified when the file is opened, and if the file already exists, that length must be the same as the length specified when the file was originally created.

Each record in the file has a number (starting with 1). The position in the file for reading or writing that record is calculated by multiplying the record number minus one by the record length.

A record must be written before it can be read. Records do not need to be written in sequence. If you write a record beyond the current end of the file, the file is extended to include the new record and the space between is filled with binary zeros. The file is, therefore, as long as the highest-numbered record that was written.

If you try to read a record that is beyond the end of the file, you will get an error. If you try to read a record that is within the file, but was never written, you will get all blanks.

Writing to a RANDOM file.

Each RANDOM file has an associated buffer, which is as long as the file's record length. You cannot access the buffer directly, but can build a record in the buffer using **PRINT #***file-num*, **PRINT #***file-num* **USING,** or **WRITE #***file-num*, and you can map variables into the buffer space using the **FIELD** statement (explained below).

Once the record has been built in the buffer, you output it to the desired location in the file with the **PUT #***file-num, record-number* statement. The contents of the buffer are transferred to the file at the position for that record. Exactly the record length is transferred, regardless of what you wrote to the buffer – the remainder is padded with blanks. The buffer contents are preserved in case you want to write the same data to another record number.

If the *record-number* is not given, the next record following the last **PUT** is written.

Creating the Record in the Buffer using PRINT # and WRITE #.

Writing to a buffer is similar to writing to a sequential file, with a few exceptions:

A **PRINT #** or **WRITE #** always starts at the beginning of the buffer, unless a previous **PRINT #** or **WRITE #** left the line incomplete by ending in a semicolon.

If you try to **PRINT #** or **WRITE #** beyond the record length (end of the buffer), you will get an error, so keep track of the buffer position and size of the data written.

You may only write one line to the buffer. If you write a second line, it will over-write the first.

Following the **WRITE #** or **PRINT #**, the remainder of the buffer is padded with blanks. If you just want to fill the buffer with blanks, use a **PRINT #** statement with no parameters.

LONGMATH numbers cannot be written to the buffer with a **WRITE #** statement. **WRITE #** formats **LONGMATH** numbers in a multi-line format. You may use the **PRINT #** statement for **LONGMATH** numbers, provided the number will fit in the buffer.

Creating or Updating the Record in the Buffer using FIELD variables.

When you read a RANDOM file record (using **GET**), the record remains in the buffer and you may update "fields" (groups of bytes) within that record. You must first define the fields in the record (buffer) with a **FIELD** statement:

FIELD #1, 10 **AS** *a*$, 20 **AS** *b*$, 5 **AS** *c*$

specifies that the buffer for file #1 is divided such that the first 10 characters are referenced by *a*$, the next 20 by *b*$, and the next 5 by *c*$.

You can then set data into these fields by using the variable (field) name in an assignment statement or in an **INPUT, INPUT #, READ, LSET** or **RSET** statement. In this way, they act similar to ordinary variables.

Examples:

a$ = "Hello"
b$ = *s*$
LSET *c*$ = *s*$
RSET *c*$ = *s*$
INPUT *a*$,…
INPUT #2, *a*$, …
READ *a*$, …

In each case, the value which is read or assigned to the field variable must be a string, string variable, or something which converts to a string. If the value

you wish to assign is a **DOUBLE**, you should first convert it to a string using the **STR$** function. **LONGMATH** numbers may be assigned directly to strings, provided that they are less than 256 characters long.

Note: You can also use **FIELD** statements to redefine string variables (see "Using FIELD statements with String Variables", below)

Caution: Any string assigned to a field variable will be truncated to the length of the field. This can cause undesirable results if the string is a **LONGMATH** number, for example, since you could lose the decimal point or exponent part of the number. To be safe, you should set the floating point length (with the **FLOAT** statement) and convert the number to one whose length will fit in the target field.

Note: Strings assigned to field variables are left-justified and padded with spaces (except for **RSET**, which right-justifies the string if it is shorter than the field width, and pads with spaces on the left).

Note: **FIELD** statements are processed ***before the program is run***. They may appear anywhere in the program. They are not valid from the command line.

Note: It is possible to move data within the buffer by assigning one field string variable to another, *e.g., c*$ = *a*$, in the example above.

Caution: Referencing (using or assigning to) a field variable which is beyond the length of the record will result in an error.

Redefining Records.

A **FIELD** statement defines how a record is laid out, or broken up into fields. If you want to redefine the fields in a given record, you can use multiple **FIELD** statements, each breaking the same record up in different ways.

Each redefining **FIELD** statement must reference the same file number. The string variables defined in the **FIELD** statement may not duplicate each other or any other variables used in **FIELD** statements or ordinary string variables in the program.

If you simply want to skip a group of characters, you can use **SPACE$** as the string variable name. This name (which is also the name of a function) may be used as many times as you like, and doesn't count toward the maximum number of variables (10) you can define in a single **FIELD** statement.

You can have up to 40 **FIELD** statements in any program.

Examples:

FIELD #2, 10 **AS** *a*$, 30 **AS** *name*$, 30 **AS** *address*$
FIELD #2, 10 **AS SPACE$**, 15 **AS** *first*$, 15 **AS** *last*$
FIELD #2, 70 **AS SPACE$**, 40 **AS** *comments*$

The first statement defines 3 fields. The second redefines *name*$ as *first*$ and *last*$. The third skips over the first 70 characters and defines the next 40 as *comment*$.

Reading From a RANDOM file.

You can read any existing record from the RANDOM file into the buffer with the statement:

GET #*file-num, record-number*

.

The contents of the file at the desired position are transferred to the buffer. Exactly the record length is transferred, regardless of what is in the file record – if the record terminates prematurely, the remainder is padded with blanks. The existing buffer contents are over-written.

Note: If you try to **GET** a record that is beyond the end of the file, an error will occur.

Once the record has been read into the buffer, you may access its contents by:

an **INPUT #** statement, to read selected variables,
a **LINE INPUT #** statement, to get the entire record (if possible),
using **FIELD** variables (see above).

Field [string] variables may be assigned to other variables, printed, used in functions, or anywhere strings are used (exceptions are the **MID$** and **SWAP** statements. When a field variable is referenced, the entire length (including padding blanks) is used.

Reading the buffer using **INPUT #** works similarly to **INPUT #** for sequential files, except that reading does not continue beyond the end of the buffer and attempting to do so will cause an error. You can use the **INPUT #** function to test if any data remains before attempting to read it. You can have more than one **INPUT #** statement, which will continue reading where the first one left off, provided there is data remaining in the buffer. After a **GET**, the pointer is reset and an **INPUT #** statement will again start at the beginning of the buffer.

You can intermix **INPUT #** statements with the use of field variables.

Random File Example:

```
1000 // Sample Check Register Using Random File
1010 data 1000 // start balance
1020 data -25.40, 100, -16.65, -35.82, -99.95, -10, -35.17
1030 data 195.00, -30.95, -25.50, -123.40, -89.99, 50.00
1040 open random,2,"register.data",60
1050 field #2, 30 as descr$, 10 as deposit$, 10 as check$, 10 as
   balance$
1060 recnum = 1
1065 descr$ = "Item:": rset check$ = "Check:":
1070 rset deposit$ = "Deposit:": rset balance$ = "Balance:"
1080 put #2, recnum
1090 recnum = recnum + 1
1100 read bal
1110 print #2 : // blank the buffer
1120 b$=str$(bal, "####.##")
1130 rset balance$ = b$
1140 descr$="Starting balance"
1150 put #2, recnum
1160 recnum = recnum + 1
1165 chknum = 1
1170 while read(2) > 0
1180   read chk
1190   write #2
1200   if chk > 0 goto 1240
1210   descr$ = "Check #" + str$(chknum, "###")
1220   rset check$ = str$(-chk,"####.##")
1225   chknum = chknum + 1
1230   goto 1260
1240   descr$ = "Deposit"
1250   rset deposit$ = str$(chk,"####.##")
1260   put #2, recnum
1270   recnum = recnum + 1
1280 wend
1290 recs = recnum
1300 // calculate running balance
1310 recnum = 3
1320 while recnum < recs
1330   get #2, recnum
1340   bal = bal - val(check$) + val(deposit$)
1350   rset balance$ = str$ (bal, "####.##")
1360   put #2, recnum
1370   recnum = recnum + 1
1380 wend
1390 // print
1400 recnum = 1
1410 while recnum < recs
1420   get #2, recnum
1430   line input #2, line$
1440   print line$
1450   recnum = recnum + 1
1460 wend
```

Output

```
Item:                          Deposit:    Check:  Balance:
Starting balance                                    1000.00
Check #  1                                   25.40   974.60
Deposit                          100.00              1074.60
Check #  2                                   16.65  1057.95
Check #  3                                   35.82  1022.13
Check #  4                                   99.95   922.18
Check #  5                                   10.00   912.18
Check #  6                                   35.17   877.01
Deposit                          195.00              1072.01
Check #  7                                   30.95  1041.06
Check #  8                                   25.50  1015.56
Check #  9                                  123.40   892.16
Check # 10                                   89.99   802.17
Deposit                           50.00               852.17
```

Using FIELD statements with String Variables.

Just as **FIELD** statements can be used to map or define random-access file buffers, they can also be used to map string variables, which also permits their use with sequential files if you use **LINE INPUT #** to read the file.

In this way, you can lay out a report line before printing it, or pick apart an input line. Specify

> **FIELD** *string-var-name*, *width* **AS** *string-var*, …

where *string-var-name* specifies the name of the string variable you wish to re-define in the same manner as with random-access file buffers.

> **Note**: *string-var-name* may not specify an array or array element. If you dimension an array called *string-var-name* it will cause an error.

> **Note**: A field may not extend beyond 255 characters from the start of the string variable.

FIELD statements are processed before the program is run, so the variable referenced in *string-var-name* will not be defined until the program is run. You can also use the string anywhere else string variables are used.

References to the fields will reference or change the contents of the string. Retrieving data beyond the end of the string will result in blanks. Setting data into fields beyond the current end of the string will cause the string to be lengthened and padded with blanks. Unlike random-access file buffers, there is no error in these cases.

Keep in mind that the fields may have trailing blanks. The length of the field string will be the length as defined in the **FIELD** statement.

Note: Each time you lengthen a string, additional string storage is used, so it is more efficient to start with the string at its ultimate maximum length. Either set the rightmost field first or initialize the string with blanks (with the **SPACE$** (*n*) function).

SERIAL "Files"

Technically, serial communication is not a file operation, however it is implemented within QuickCalc BASIC using the same statements and functions as are used with files.

This kind of serial communication is "records" (i.e., strings) rather than bytes, and operates in one direction at a time. You can send a string, and you can receive a string, just not simultaneously. Although data can travel in both directions through the serial cable (or USB cable), the data is sent using the **PRINT #** or **WRITE #** statements and received with the **INPUT #** or **LINE INPUT #** statements, so you must complete one statement before executing another. You can't "monitor" a line for bytes as is done in a terminal emulator program (like Hyper Terminal).

Basically, this works by sending a message and then waiting for a reply. It is intended for communicating with a micro-controller (such as an Arduino) which has a similar function built into it. Using this type of communication, you can turn an Arduino with a sensor into a computer-controlled automated test device, controlling what is to be measured and returning the data for printing, analysis and/or plotting.

Opening a Serial File

You open a SERIAL file the same as a sequential or random file, using the **OPEN** statement with a few extra parameters:

OPEN *file-mode, file-num, file-specification* [*, record-size* [*, baud-rate* [*, parity* [*, stop-bits*]]]]

file-mode must be SERIAL,

file-num must be an available file number, not already open, similar to standard files.

file-specification must refer to an available and active COMM port, which may be a USB port with the appropriate device driver assigned to a COMn port ID. You may hard code it, (e.g., "COM2"), use a string variable containing the port ID, or use the function **BROWSECOMMPORTS$**.

record-size is at least the size of the longest message you will send or expect to receive.

baud-rate is the speed of the data communication. The default is 9600 baud. The selected value must match the baud-rate setting of the device with which you are communicating.

parity is used for error-detection. The default is no parity checking, which works well for short lines at moderate speed. It must match the parity setting of the device with which you are communicating.

This value must be a **string** constant (in quotes) or string variable. Allowable values are: "**NO**", "**ODD**", "**EVEN**", "**MARK**", or "**SPACE**".

stop-bits is the number of stop bits used following each character. This is a numeric value. Allowable values are: 1, 1.5, or 2. The default is 1. It must match the stop bits setting of the device with which you are communicating.

OPEN may also be used as a function, returning an error code, similar to ordinary files.

Note: An **OPEN** may be successful and still the communication may not work, if the settings do not match the external device.

Sending Serial Messages

Messages are sent using the **PRINT #** and/or **WRITE #** statements. This is similar to the way it is done with regular files, with a few exceptions.

Each **PRINT #** or **WRITE #** statement sends one line of data through the serial port.

If a **PRINT #** or **WRITE #** parameter list ends in a semi-colon, you must issue at least one more to complete the line. Nothing is sent until the line is complete.

If you exceed the buffer size, **it will cause an error** and terminate the program. Data will not "wrap around" to the next message.

Do not include C/R or l-f (new-line). A new-line character will be appended to the message when it is sent out and [hopefully] stripped off when it is received by the external device. An extra new-line character at the end may cause a second "empty" message to be sent. [The new-line character which terminates a string will not be sent.]

If you wish, you can format a string using any of the available statements, including **FIELD**, **LSET**, **RSET**, **MID$,** *etc*., *etc*. (see above) and then send it using a single **PRINT #** statement (e.g., **PRINT #**2, *string-var*$). Just don't exceed the buffer size.

Remember, a string may be up to 255 characters. A data record sent could be longer if you print more than one string, however you would not be able to read it back with **LINE INPUT #.**

Receiving Serial Messages

Serial messages are read using the **INPUT #** and/or **LINE INPUT #** statements. Again, this is similar to regular file operations, with a few exceptions:

Serial operations can **time out**. If the [complete] message is not received after 5 seconds, you will get an error which will terminate the program.

To avoid time-outs, you should not issue the **INPUT #** or **LINE INPUT #** until you know a message is waiting. You can use the **INPUT function** to determine this:

```
100 ready = INPUT (file-number)
if ready = 0 then goto 100  // loop until message is ready
INPUT # file-number, list-of-variables
```

You must get all the data from the input message in a single **INPUT #** or **LINE INPUT #** statement. Another **INPUT #** or **LINE INPUT #** statement will initiate another serial read, which will throw away the buffer contents , and more importantly, cause a new read which could cause a time out error.

If you want to pick apart the message or interpret it in more than one way, use the **LINE INPUT #** and then operate on the returned string.

Extra data items that are not assigned to variables will be ignored. Extra variables in the **INPUT #** statement (for which no data values appear) will not be changed. No error messages are generated in either case.

A good way to handle this is to read the entire line into a string with **LINE INPUT #**. Then parse the string any way you like to extract the data.

Serial Protocol

Of course, you may define your own set of rules for sending/receiving messages, but here are some of the things which I have found to be important.

There should be a **one-to-one correspondence** between incoming and outgoing messages. It should be a command-response protocol.

If the computer requests data from the external device and the data is not ready yet, the device should return a “not ready” message. Otherwise, the computer may have to wait and can get “hung up” waiting. The computer can keep requesting data, and eventually get a positive response. Also, the computer should be able to send a

"cancel request" message, to which the device should respond "OK" after cancelling the operation.

The important thing is that the two machines must not get "out of sync", which is why it is best to send one message and get one response.

> It is possible to send one message and get 2 responses, but you must expect and attempt to read that second response and not send another message until you have received it.
>
> One technique for receiving multiple messages is to watch for messages inside a loop, and keep receiving messages until the device sends an "end" message. Of course, if the "end" message never arrives, you will loop forever.

It is best to keep the data format fixed and simple. Always send and expect the same number of data items as a response to each type of request.

New-Line characters (hex 0A or '\n') are a "no-no'. Don't include them in any message in either direction. The Serial communications routines in the PC and the external device append them to messages and strip them off on receiving them. An extra new-line character could prematurely end the message, or cause the system to think that more than one message had been received. This would cause the protocol to get out of sync.

In the remote device, don't send a partial message if there may be some time elapse before the rest of it is sent. For example, "The answer is: " [*pause while we compute it*] "3.15159". The delay could cause a time-out to occur. Wait until the data is all available before sending the first part of the message.

Communicating with an Arduino micro-controller.

I have tested these routines with the Arduino Mega2560 and Nano boards. It should also be compatible with the Uno and similar boards from other manufacturers.

In the Arduino Sketch

Receive the incoming message.

Use something like:

```
int get_serial_line ()
   {
   int   count;
   while (!Serial.available ());
   count = Serial.readBytesUntil ('\n', line, 50);
   line [count] = '\0';
   return count;
   }
```

It is important to read the entire line before sending an outgoing message.

Sending messages:

Use Serial.print and Serial.println.

Serial.print prints part of the line. You must end it with a Serial.println

In the PC (Basic Program)

Don't include C/R or l-f characters in the sent messages.

Send one message, then read a response.

Example:

PC:	Get Sensor Data
Arduino:	OK
PC:	Sensor Data Ready?
Arduino:	NO

(brief wait and repeat until data is ready)

PC:	Sensor Data Ready?
Arduino:	YES
PC:	Transmit Sensor Data
Arduino:	1.23456

The idea is that one message = one response. No long waits.

It is also a good idea to have some kind of "cancel" message which the Arduino can listen for. This would clear any ongoing process and wait for the next command.

Example:

PC:	Get Sensor Data
Arduino	OK
PC	Sensor Data Ready?
Arduino	NO

(after retrying for a reasonable time)

PC	Cancel
Arduino	OK

Receiving Data:

Use the **INPUT ( )** function to see if a message is ready, then use **INPUT #** or **LINE INPUT #** to read it.

Example:

```
OPEN SERIAL, 2, "COM5", 80, 9600, "no", 1
PRINT #2, "outgoing-message"    // send request for data
100 msg_avail = INPUT (2)
IF (NOT msg_avail)  GOTO 100   // loop until message avail
LINE INPUT # 2, input_line$     // read the returned message
```

Sent data:

Use **PRINT #** or **WRITE #** statements. If **PRINT #** or **WRITE #** parameter list ends in a semi-colon, you must issue at least one more to complete the line.

If you exceed the buffer size, **it will cause an error.** Data will not wrap around to the next message.

If you are waiting for an input message, don't send another output message until after you have received it. If you think your message got lost, first issue an **INPUT ( )** function, and then send your output message. In this case, the Arduino will need to have a protocol established to deal with unexpected messages (like "cancel").

Do not include C/R or l-f (new-line). A new-line character will be appended to the message when it is send out and [hopefully] stripped off when it is received by the Arduino.

The Arduino File Transfer Program.

As an example of Serial file usage, plus other QuickCalc BASIC features, I have written a program to transfer files to and from an Arduino with an SD card, using the same serial port that is used to upload the programs to the Arduino. This program, and the associated Arduino sketch, are included with the Sample Programs. Compile and upload File_Transfer.ino to the Arduino, make sure the serial cable is connected, and run File Transfer.txt in QuickCalc.

The file transfer is slow (noticeable for larger files) mostly because the Arduino SD card reads and writes data one byte at a time. However, for smaller files, it is quicker than removing the SD card and connecting it to the PC, especially when the SD card is not easily accessible.

BINARY Files.

BASIC generally works with ASCII text file format. It is line-oriented. A line is a string of characters, generally terminating with a carriage-return (C/R or hex 0D) and line-feed (l-f or hex 0A). The C/R is optional.

Because the file data must be able to be read into character strings, the records must not contain "null" characters (hex 00) and may not have C/R or l-f embedded in the records. If a BASIC program attempts to read a file that violates these restrictions, errors can occur and data may be lost.

Normally, for simple BASIC programs, these restrictions do not present a problem. Sometimes, however, you may want to read a file which contains binary-formatted data such as image files, executable files, zip files, database files, files containing floating-point encoded data, *etc*. QuickCalc provides a way for you to read and write these files.

The BINARY File Types.

You identify binary files by giving them the type "**INPUT BINARY**" or "**OUTPUT BINARY**" in the **OPEN** statement or function"

> **OPEN** [**INPUT BINARY** | **OUTPUT BINARY**], *file-num*, *file-spec*, *record-length*

This is similar to the way ordinary text files are specified, except for the new file types.

Note: The keywords **INPUT BINARY** and **OUTPUT BINARY** must contain exactly one space between the words.

The *record-length* field is **NOT** optional for binary files. It specifies the maximum number of characters to be read from or written to the file, and must **not be greater than 127**.

Input and Output Translation.

Data from binary files is read or written as a block. There is no logical record delimiter. It reads exactly the number of bytes specifies, including nulls, l-f, C/R, etc. The last block read from the file may be smaller.

Since the data must be read into a string variable, the "forbidden" characters are translated into escape sequences as follows:

C/R	('0D') is replaced with	\r
l-f	('0A') is replaced with	\n

null	('00') is replaced with	\0
quote mark	is replaced with	\"

The resulting block is now a valid character string. It is terminated with a null character and assigned to a character string. You may then parse it or pick it apart using the various BASIC statements and functions.

Note: The resulting string, after translation, may be up to **twice as long** as the original. That is the reason for the length restriction of 127. The translated string must be less than 256 to fit into a BASIC string variable.

When a string variable is written to a BINARY file type, the reverse translation is applied. The terminating null character is not considered part of the data. The resulting block, after reverse translation, may have fewer bytes than the translated string. The exact number of bytes in the reverse-translated block is what will be written to the file.

The translation and reverse-translation is performed for you automatically as you read and write the data.

Note: A translated string may be sent and received using a SERIAL file (see above). This provides a way to send and receive binary data. The device at the other end of the transmission must translate the data back.

Reading and Writing BINARY Files.

Binary files are read using the **LINE INPUT #** statement. The data must be assigned to a string variable.

LINE INPUT #*file-num, string-var$*

The ordinary **INPUT** statement is invalid with BINARY file types.

The resulting string variable will contain translated character sequences. It may also contain non-printable characters, tab characters, control characters, etc. Printing this data to a printer will probably not work.

If, after reading the line, the length of the string variable is zero, it means you have reached the end of the file.

Binary files are written using the **PRINT #** statement:

PRINT #*file-num*, [*string-var$* | "*string-constant*"]

The string variable or constant must have the substitutions listed above to make it "legal". If you read the string from a binary file as described above, it will be legal. If it is a quoted string which you built from other data, you must make those substitutions yourself before "printing" it.

A Simple File Copy Program

```
OPEN INPUT BINARY, 2, "input-file", 127
OPEN OUTPUT BINARY, 3, "output-file", 127
100 LINE INPUT #2, x$
IF LEN (x$) = 0 then GOTO 200
PRINT #3, x$
GOTO 100
200 CLOSE 3
CLOSE 2
```

The above program should create an exact duplicate of the input file. Just before the **PRINT** statement, you could modify the string to change the data any way you like (as long as it remains "legal")