NIBBLING AT APPLESOFT

Errors Without Messages

by Leslie R. Schmeltz
3224 Magnolia Ct.
Bettendorf, IA 52722

Well, Mr. Doom and Gloom is back! If you recall our last two discussions, error messages and their causes were described in some detail. This time we're going to shift our emphasis to a much more interesting process — sort of a graduate program in software sleuthing! As anyone who's ever written a program will tell you, there are times when things aren't happening as they were intended. No error messages appear in spite of the fact that processes or file manipulations may not be doing what you thought they would. Under other circumstances, error messages can be made to work for, instead of against, you. Eliminating or controlling errors will be our major topic of discussion in this installment of "Nibbling At Applesoft".

We will introduce the concept of error trapping and outline some of the procedures for effective use of error trapping. Next, error handling will be discussed. We will also be looking at some of the commands and procedures necessary to check your programs for proper operation. Finally, we'll outline a few of the more common programming tricks that will help you avoid process errors. Sounds pretty busy, doesn't it? Well, let's get right to work.

ERROR TRAPPING

The most obvious way to solve problems with errors is simply not to let them occur. While that may be easier said than done, there are some things you can include in your programs to prevent errors from sneaking in. Let's say, for instance, that you are using an **INPUT** statement for the purpose of soliciting a number between one and ten from the keyboard. How can you prevent someone from trying eleven? Easy — watch?

```
10 PRINT "PRESS A NUMBER BETWEEN
   1 AND 10"
20 INPUT A
30 IF A > 10 THEN PRINT "THAT'S MORE
   THAN 10, DUMMY!": PRINT "TRY IT
   AGAIN.": GOTO 10
50 PRINT "THANKS, I NEEDED THAT!"
60 END
```

So now we have a program that doesn't do anything (examples really don't have to, you know) except thank us for a proper value and chide us if the number inputted is **greater than 10**. You just know some wise guy will try typing zero or a minus number, right? As it stands, the program will thank him kindly and go on about its business. Let's expand the error trapping a little and try to catch the zeroes and minus numbers:

```
10 PRINT "PRESS A NUMBER BETWEEN
   1 AND 10"
20 INPUT A
30 IF A > 10 THEN PRINT "THAT'S
   MORE THAN 10, DUMMY!": PRINT
   "TRY IT AGAIN.": GOTO 10
40 IF A < 1 THEN PRINT "THAT'S LESS
   THAN 1": PRINT "TRY IT AGAIN.":
   GOTO 10
50 PRINT "THANKS, I NEEDED THAT!"
60 END
```

Try typing and RUNning these short examples. Do they indeed trap the potential errors? Can you think of any other possibilities for input that the error trapping doesn't cover? Can we streamline the error trapping? Let's try:

```
10 PRINT "PRESS A NUMBER BETWEEN
   1 AND 10"
20 INPUT A
30 IF A < 1 OR A > 10 THEN PRINT
   "THAT'S NOT BETWEEN 1 AND 10.":
   PRINT "TRY IT AGAIN.": GOTO 10
40 PRINT "THANKS, I NEEDED THAT!"
50 END
```

The preceding routine does the same job but eliminates one line by checking for values of **A** that are either too low or too high in a single line. Notice that **line 30** will execute only if the value of A is outside the acceptable limits — otherwise the program proceeds directly to **line 40**. What if the same wise guy tried to input a letter or control character instead of a number? Applesoft will come to your rescue here since A must be numeric, so there's no need to trap for those errors. Granted, this is a simple example, but the principle holds true no matter how complicated your specific application. Let's take a slightly more involved example. We will be using each of **three possible responses** for a specific action. We will, however, still want to trap out any erroneous responses to our prompt:

```
10 PRINT "PRESS RTN TO EXIT, ESC
   TO CONTINUE": PRINT "OR 'P' TO
   PRINT"
20 GET Z$
30 IF Z$ = CHR$(13) THEN END
40 IF Z$ = "P" THEN GOTO (PRINT
   ROUTINE)
50 IF Z$ < > CHR$(27) THEN GOTO 20
60 REM BALANCE OF PROGRAM
```

In the preceding example, notice we have examined the response **(Z$)** to see whether or not it is the **RETURN key (CHR$(13))** or P. If it is neither, then it must be either an error or **ESC (CHR$(27))**. Error trapping in this fashion allows you to check multiple responses and initiate desired program actions simultaneously. There are many situations where you want to check the accuracy of a keyboard response. **By properly structuring the error trapping routine, one can check for almost any possible combination of responses.** Simple error trapping in the INPUT or GET routine will often save hours of grief later in the program's operation (or lack of operation, as the case may be!).

Let's consider another example. Suppose your program allows for the manipulation of a particular string data field of **up to 10 characters**. Simple error trapping must be used in the input to be sure the string does not exceed that length.

```
10 PRINT "TYPE YOUR FIRST NAME."
20 PRINT "(10 CHARACTERS OR LESS)"
30 INPUT N$
40 IF LEN(N$) = > 11 THEN GOTO 20
50 BALANCE OF PROGRAM
```

This routine assures you that any response of more than 10 characters will result in another "(10 CHARACTERS OR LESS)" prompt. Notice that we are recycling the original length prompt rather than using a separate message. As an alternative approach to error trapping, setting up original program prompts for dual use is a very efficient use of program lines. You could, if desired, add a separate error message and direct the user to **TRY AGAIN** or **ABBREVIATE IF NECESSARY**.

Of course, there are other situations where error trapping is used. **Limiting the range of values** for calculations, checking for possible **errors in data fields** and **eliminating redundancy** in mass storage files are but a few of the possibilities. The goal of any error trapping operation is to catch potential problems before they are allowed to occur. This is opposed to error handling (which we will discuss a little later) and might be called the proverbial "ounce of prevention" rather than the "pound of cure". The process of error trapping is not really all that complicated — you just need to think of ALL the possible error responses and anticipate them in your programming!

While trapping every possible error in a program is a worthy goal, it is rarely achieved by most programmers. By thinking in terms of error possibilities and "smoke testing" your programs with inaccurate responses, you will have gone a long way in reducing the frustration of having to deal with the same errors time after time. **Just for the experience, try "bombing" one of your favorite programs with inappropriate responses to prompts. For each successful "bomb" of the program, see if you could add some error trapping that would eliminate the problem.** Both your programming skills and the program under consideration will benefit greatly from this process.

USING ERROR TRAPPING EFFECTIVELY

As promised earlier, we will outline a few of the principles of effective error trapping. No doubt you will be able to add a few more that have worked for you, so let's consider this list as minimal.

1. In order to error-trap a routine, you must be thoroughly familiar with the type of response expected by the program.
2. Error trapping procedures must account for the full range of possible responses that can be reasonably expected from the user.
3. Messages generated by error trapping procedures should give the user a clear picture of why a specific response was incorrect and another chance to respond appropriately.
4. The end result of error trapping procedures is the prevention of program or process errors that will result in destruction of data.

5. Error trapping routines should be invisible to the program user unless an inappropriate condition is encountered.

6. Routines used for error trapping should be as simple and efficient as possible. You cannot cover every possible contingency; be content to handle the vast majority!

ERROR HANDLING

If your error trapping procedures are up to par, there will not be many unintentional errors in your programs. Unintentional? Say what? You mean there are times when errors are intentional, even desirable? YUP! Let me explain — error conditions often indicate the lack of completion of a process or the program's inability to function because of a resolvable problem. Under normal conditions, error conditions evoke one of the standard error messages contained in Applesoft and the Disk Operating System. Once the error message is printed, all program operations cease.

If we interrupt the normal process of getting an error message just prior to program termination, some alternative method must be provided to resolve the problem **before** execution can continue. This alternative method is called **Error Handling**, which seems logical enough since that is exactly what we are doing. Error handling provides some means of resolving simple problem errors without loss of data.

Earlier, we learned one method of error handling — typing RUN and re-running the program after the error condition had been remedied. As you know, typing RUN destroys any data that may have been generated by the program, unless it has been saved prior to the error condition. Is there a better way? You bet your bippy there is!

When a program error occurs, the Apple sets the value of decimal memory location 222 equal to the number indicating which error message will be printed. The user is free to set up a command that will intercept this normal process and handle the error differently than simply stopping program execution. There are some inherent dangers in setting up your own error handling routines, as we will see a little later.

AN EXAMPLE

Before we get too specific, why don't we take a look at a real live error handling routine.

Let's set up a scenario and see how we can program it. Your program is designed to create data files for any of several categories of items. If there is already a file on your disk by that name, no new one should be created. We also want to know when the storage disk is full so that the user can be prompted to insert a fresh data disk in the drive. Any other error should indeed terminate program operation. How do we set this up? Something like this:

```
10 D$=CHR$(4): REM CHR$(4) IS CTRL-D
20 ONERR GOTO 100
30 INPUT "NAME OF DATA FILE";FN$
40 PRINT D$; "VERIFY"; FN$
60 END
100 Y = PEEK(222): REM READ ERROR CODE
110 IF Y = 6 THEN PRINT "THERE IS NO FILE BY THAT NAME ON": PRINT "THIS DISK. OK TO CREATE ONE?": GOTO (file creation routine)
120 IF Y = 9 THEN PRINT "THIS DISK IS FULL. PLEASE INSERT A": PRINT "FRESH DATA DISK IN THE DRIVE AND": PRINT "TRY AGAIN.": GOTO (delete file, change disk and try again routine)
130 POKE 216,0: REM RESET THE ERROR FLAG.
140 PRINT "PROGRAM OPERATION TERMINATED BY ERROR."
150 END
```

This routine, as you can see, looks considerably different than the error trapping variety outlined earlier. Lines 10 through 60 are a simple program which asks you to type a file name and checks on the disk to see whether or not the file is present. If an error is encountered anytime after line 20 is executed, the program will branch to line 100 instead of printing an error message. In the error handling routine, we have provided for only two of the more likely error possibilities — **file not found** or **disk full**. Error handling routines can be as simple or elaborate as you like.

If **error code 6** is detected in memory location 222, we can assume that the file name typed was not found on the disk. In that case, a routine to either **change the name** or **create a file** by that name must be provided. If, during the File Creation portion of the program, the disk is found to be full, a routine to delete the partial file from the current disk and create a complete file on another disk must be provided. In this example, any other error code will display "PROGRAM TERMINATED BY ERROR" and the program will end.

If you remember the wide range of possible error messages we have discussed in the last couple of installments of this series, it will become quite apparent that you would probably not care to try and handle all the possible errors for any specific program. Most error handling routines try to cover a few of the more likely possibilities and let the standard error messages appear for the rest.

CAVEATS FOR USING ERROR HANDLING ROUTINES

As you have probably noted, error handling can be a powerful programming tool if properly used. On the other hand, it can be a pain in the neck if problems are encountered. This process, like so many others in programming, is best learned by experimentation and experience. I will, however, give you a few caveats to keep you out of serious trouble.

1. An **ONERR GOTO** command sets the line number to which the program will branch regardless of where an error occurs. On encountering an error, program execution continues as directed by the most recently executed ONERR GOTO statement.

2. An **ONERR GOTO** command **MUST** be executed **prior to** an error being encountered if program execution is to continue rather than terminate in an error state.

3. A **code** representing the nature of the error encountered is contained in **memory location 222** (decimal). To see which of the error codes is contained there, type **PEEK (222)**. Some of the more common error codes — **(0)-NEXT WITHOUT FOR, (5)-END OF DATA, (6)-FILE NOT FOUND, (16)-SYNTAX ERROR** and **(254)-BAD RESPONSE TO INPUT STATEMENT.**

4. **Error handling routines for problems encountered in FOR — NEXT** loops must **restart the loop** after the error is resolved. If you attempt to re-enter the loop in progress, a NEXT WITHOUT FOR error will halt program execution.

5. When handling errors encountered during the execution of a subroutine, **return to the GOSUB statement** after resolving the error. Failure to do this will result in the RETURN WITHOUT GOSUB error message.

6. With a great deal of **caution**, you may use **RESUME** at the end of an error handling routine to return program execution to the beginning of the statement where an error occurred. Be careful, however, because in certain cases RESUME can place your program in an infinite loop.

7. **Don't forget to end your error handling routines with POKE 216,0** to restore normal operation of the error message mechanism (if you want to disable the currently active ONERR).

8. As we saw in the sample, error handling routines must include provisions to **identify AND resolve the problems** noted. Program execution must be resumed at an appropriate point to avoid further difficulties.

Do you get the impression that you have just been presented with the universal solvent and have been told to come up with a container to hold it? Error handling routines can be a tremendous help in keeping the user of your programs out of trouble, but they present a whole unique series of problems of their own. The primary requirement for using error handling is being able to plan for all of the contingencies that may be encountered during the execution of a program, and set up routines that will deal with them.

WHO'S ON FIRST?

Remember the classic comedy routine that asked that same question? Have you ever heard of it? Maybe your grandparents could fill you in! Although you may not be old enough to remember Abbott and Costello, the confusion created by perfectly logical answers to questions has a lot of application in computer programming. We often get correct answers to the questions we are asking the computer — it's just that we really don't want to know what we are asking. Right answer, wrong question!

Programmers are often faced with finding out why certain results are being obtained from a particular program or routine. There may not be an error in the program lines, yet unsatisfactory answers are being obtained. That's about the time we begin to ask anew, "Who's On First?"

TRACE/SPEED

Applesoft incorporates several commands to help you see the operations of a program as they are taking place in order to solve these sticky types of problems. **TRACE**, which can be used in either immediate or deferred execution modes, causes the number of each line to be displayed on the screen as it is executed. Although this may mess up some of your pretty screen formats, knowing the line number where an error occurs can be a big help. To turn off the **TRACE** provision, type **NOTRACE**. Remember **SPEED**= that we discussed some time ago? Keep in mind that it is often easier to see what is happening in a program if you slow things down a bit (or a lot). To use the **SPEED**= command, just type it (while you are in immediate mode) followed by a number between **0 (slowest)** and **255 (normal speed)**. The combination of **TRACE** and **SPEED**= can really drag things out to the point where you can follow the program step by step through its operations.

MON/NOMON

Disk operations can be monitored through the use of the **MON** (short for monitor, you know) and **NOMON** commands. A little more versatile than TRACE, MON lets you select any combination of three parameters for watching — **C (Commands to the disk)**, **I (Input from disk files)** and **O (Output to the disk)**. The format for the commands is **MON C, I, O** or **NOMON C, I, O** (if all three parameters are desired). The **C, I** or **O** may be in any order in the command. At least one of the parameters is required or else the command is ignored. If MON is in effect and you were reading five items from DATA FILE, you would see the commands and data on the screen as they are executed:

```
OPEN DATA FILE
READ DATA FILE
7.5
3.25
77.6
57.9
42.1
CLOSE DATA FILE
```

As you can see, the value of having both the commands and data displayed on the screen is tremendously valuable for debugging programs. Used in conjunction with TRACE, the MON command allows you to see exactly what your program is doing during each step. Try these commands with your own programs. What you see may surprise you!

AVOIDING PROCESS ERRORS

Earlier in this article, we referred to the prospect of avoiding process errors in the execution of your programs. What are process errors? Well, I like to think of them as "little boo-boos" instead of major programming problems. They are usually the result of some carelessness on the part of the programmer and are almost always difficult to find. Most process errors do not show up with error messages because they are probably correct in terms of syntax and punctuation. Unlike most errors, these little jewels just surface when some bizarre results are being obtained from the program.

The best way to avoid process errors is, of course, not to make any mistakes in writing your programs. Simple? Yup! Possible? Doubtful. I really have never met a programmer who, at least once or twice, has not made some errors. There are some areas where mistakes are more likely to creep in than others. I would like, as a final dying gasp in this whole unpleasant discussion, to present a few of the areas where I have found process errors in some of my own programs. No doubt we could all expand this list, but let's use it as a starting point anyway.

1. **Parenthetical grouping of mathematical operations.** I usually have to end up counting the left and right parentheses to be sure the numbers are equal. It helps if you play computer and try evaluating some of your expressions to be sure the desired results are being obtained.
2. **Rounding numbers.** There is a quirk in Applesoft that adds a **.000001** to the rounded results of some calculations. There are even more quirks in my techniques for rounding numbers!
3. **Arrays**, particularly when used in conjunction with tape storage, can be confusing. It is often easier to confine your arrays to one or two dimensions unless you have a mind that's organized like a Rubik's Cube.
4. **Variable assignment and inadvertent reassignment** during the course of program execution. Do you use **FOR I = 0 to 10** to set up a FOR — NEXT loop? Me too! Most of the array examples in the Applesoft manual also designate elements by the same variable, i.e., A(I). Confusing? Yup. Even though I try not to use the same variable, every once in a while I find that same variable **(I)** used somewhere else in the same loop (most often in conjunction with array elements).
5. **Reserved words in variable names.** Really bizarre results can be obtained if you happen to intrude on the list of reserved words for your variable names. The best way to avoid this problem — study the list of reserved words in the Applesoft manual!
6. **Disk commands MUST be preceded by a carriage return.** This little requirement can give you fits if you try to use disk commands after a **GET** prompt. The usual response is for the program to print the disk command on the screen, just as though **MON C** was in effect. Solution? Precede each series of disk commands by **PRINT:**. For example: PRINT: PRINT CHR$(4);"BLOAD BINARY". (It took me a long time to catch on to this one).
7. **Punctuation and syntax** are particularly important in disk commands. You can get some very interesting results by combining several disk commands on a line and not using the proper combination of quotation marks!

We could go on and on with this discussion, but I think you get the point. Every programmer, whether beginner or expert, needs to **compile a mental list of areas where problems are likely to surface** in his programs.

Each and every item on that list should receive extra checking during the process of debugging a new program. If you are aware of potential problem areas, debugging and editing becomes a much simpler process.

SUMMARY

In this installment, we have discussed some of the positive errors which may be eliminated or rectified without the loss of program function and data that may have been generated. Although avoiding errors altogether is certainly a worthwhile goal, it is rarely achieved in everyday programming efforts. Knowing how to trap errors, handle the more common ones within a program and outline areas where you are likely to make errors will save countless hours of debugging in the future.

MAJOR POINTS:

Error Trapping.

1. Most often used to detect inappropriate responses to INPUT or GET statements within a program.
2. May be used to check multiple responses and initiate program operations simultaneously.
3. Other areas where error trapping may be used include limiting the range of values for calculations, checking for possible errors in data fields and eliminating redundancy in mass storage files.
4. In order to successfully error trap your programs, you need to think of all the possible error responses and provide for them in your program.
5. One of the best ways to sharpen your error trapping skills is to intentionally try to "bomb" one of your own programs, and then eliminate the error with some simple error trapping procedures.

Using Error Trapping Effectively.

1. You must be familiar with the type of response expected by the program.
2. Error trapping procedures should account for the full range of possible responses that can be reasonably expected from the user.
3. Messages generated by error trapping procedures should be clear and self-explanatory to the user.
4. Error traps should remain invisible during program execution unless an error condition is encountered.
5. Keep your routines for error trapping as simple and efficient as possible.

Error Handling.

1. Provides a means of identifying and rectifying errors that occur during the execution of a program.
2. Interrupts the normal process used by your Apple to handle errors.
3. Should cover the most likely error possibilities and provide for the rest as a group.
4. Can be a powerful programming tool if properly used, a very large pain in the neck if not!

Using Error Handling Effectively.

1. An **ONERR GOTO** statement must be executed prior to an error being encountered if error handling, rather than program termination, is to take place.
2. Program execution branches to the line number specified by the most recently executed **ONERR GOTO** statement when an error surfaces.

3. A code representing the nature of the error is contained in decimal **memory location 222**. This code may be examined through the use of a **PEEK (222)** command.
4. Error handling that occurs in the process of a **FOR — NEXT** loop must restart the loop.
5. Errors encountered and handled during the operation of a subroutine must return program execution to the **GOSUB** statement.
6. If used with appropriate caution, **RESUME** will return program execution to the beginning of the statement where an error occurred.
7. In order to reset the normal error handling mechanisms in your Apple, use **POKE 216,0**.
8. Error handling must be able to **identify AND resolve** any problems noted, and then return program execution to the proper point to avoid further errors.

Watching Your Program Operate.

1. Line numbers of Applesoft program lines may be displayed during program operation through the use of the **TRACE** command.
2. **TRACE** is disabled by typing **NOTRACE**. Either command may be used in deferred or immediate execution modes.
3. **SPEED** = may be used to slow down the display processes associated with a program. This will, in effect, slow down program operations enough to help debug some of the problems.
4. Disk operations may be monitored with the **MON** command. The parameters **C (Disk commands), I (Input from the disk)** and **O (Output to the disk)** may be specified.
5. The **MON** capabilities selected may be disabled individually or as a group by using the **NOMON** command with the same parameters.

Process Errors.

1. Are minor errors that do not surface during the debugging of a program because they have correct syntax and punctuation.
2. Often the first indication of a process error is some bizarre result coming from the program.
3. **Every programmer needs to compile a mental list of areas where he is likely to make process errors and check those areas carefully in his programs.**
4. Some of the process error problem areas mentioned by the author include **parenthetical grouping for mathematical operations, rounding numbers, arrays, variable assignments, reserved words in variable names, disk commands, punctuation and syntax.**

WHAT'S NEXT?

Have you had enough of this gloomy discussion about errors and problems? I certainly have! Let's get back on a happier track next time and discuss some positive programming for a change. I'm going to keep you in suspense until then, but rest assured you will enjoy the topic a heck of a lot more than what we have been talking about for the last couple of installments. See you then!