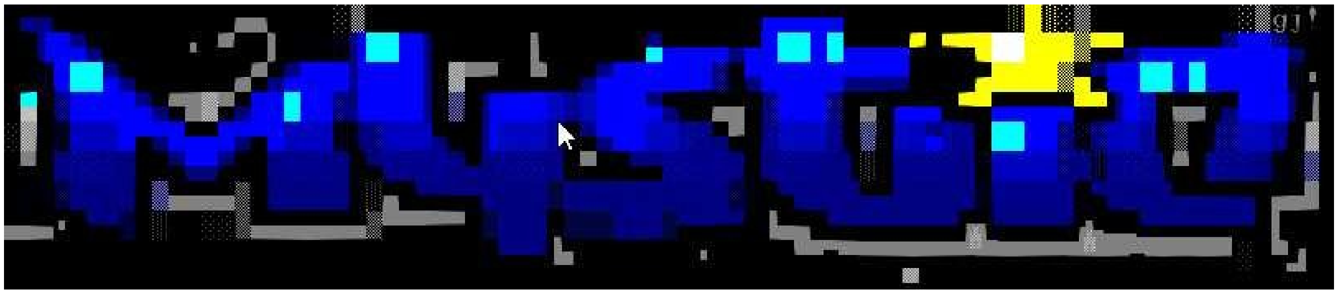

MPL Training
Lesson 10
Black Panther

First, you will notice the format has changed a bit with these lessons. I've decided to create them as PDF files, so that it's easier for me to add screen shots, and copy code segments into the text. The lessons will probably be a little bit longer, just because of the additions, but I'll try not to dump too much information into one lesson.

In the last few lessons, we've looked at how to access file. Before that, I had talked a little bit about using records and using loops. Well, in this lesson, let's combine those lessons, and move forward with them.

We'll take a real-world example here. Let's say we are writing an MPL program that will read the last 10 callers to our BBS. I know this has been done many times, but it's a fairly easy MPL to learn with, and with us working on our current project, I felt this would be a good time to go through some of the steps in a bit more detail.

Now, remember when you are going to be reading records from an existing data file, you will need to have the record format. With Mystic, this is supplied in the records.112 file, within your /docs directory. The reason for this, is our MPL will need to know what variables are located where within the record, and what the total size of the record is. For example, if the record size you are trying to read is 260 bytes, and the actual record is only 140, you will end up with variables filled with garbage data, and you'll see errors such as the program is trying to read past the end of the file.

Here is how the record is set up with the 'callers.dat' file:

```
Type   RecLastOn = Record              // CALLERS.DAT
   MDateTime  : LongInt;
   NewUser    : Boolean;
   PeerIP     : String[15];
   PeerHost   : String[50];
   Node       : Byte;
   CallNum    : LongInt;
   Handle     : String[30];
   City       : String[25];
   Address    : String[30];
   Gender     : Char;
   EmailAddr  : String[35];
   UserInfo   : String[30];
```

```
    OptionData : Array[1..10] of String[60];
    Reserved   : Array[1..53] of Byte;
  End;
```

Remember, the first line, ‘Type RecLastOn = Record’ is telling the MPL, that you are setting up a Record type variable for use in this program, and you want to call it ‘RecLastOn’.

The other variables need to be exactly as they are in the records.112 file, as far as the variable types. So, you’ll notice the first variable in the record is called ‘MDateTime’ and is set as a LongInt. But, if you look in the records.112 file, it shows the first variable named ‘DateTime’. Why is it different? Well, while I was working on this MPL, I was getting an error stating there was duplicate variable names. So, I just changed the name of the variable within the record. This is perfectly fine to do, as long as you don’t change the variable type. The name is just used within your MPL, so you can call it whatever you’d like.

Now, in order to be able to use our record in our MPL program, we need to assign it to a variable that we will be using. In this case, we know there are 10 records within the callers.dat file, so we’ll set up a variable array so we can read all of the information in one loop.

```
Var
  Call   : Array[1..10] of RecLastOn
```

There, we now have an array of 10 empty record variables that we can use in our program. So, now is the fun part. We get to create, in this case it will be a function as we’ll return a value, that will open the data file, and read each of the 10 records into our fancy ‘Call’ records variables.

```
Function ReadCall:Boolean
Var
  Ret         : Boolean = False
  Fptr        : File
  counter     :Byte=1
  Callerdat  : String
Begin
  Callerdat:=CFGDataPath+'callers.dat'
  fAssign(Fptr,Callerdat,66)
  fReset(Fptr)
  If IOResult = 0 Then Begin
    For counter:=1 to 10 do
    Begin
      If Not fEof(Fptr) Then Begin
        fReadRec(Fptr,Call[counter])
      End
    End
    Ret:=True
    fClose(Fptr)
  End
  ReadCall:=Ret
End
```

Don’t worry, it’s not as bad as it looks. That’s actually a small procedure for some of the MPLs I’ve written. :)

Let’s go through it line-by-line.

```
Function ReadCall:Boolean
```

Function is telling the compiler, this procedure will be returning a value. In this case, it will be returning a boolean, or True/False (0/1), which indicates if it was successful or not. It doesn’t take any parameters, or they would be within ( )’s after the function name.

```
Var
  Ret        : Boolean = False
  Fptr       : File
  counter    :Byte=1
  Callerdat : String
```

In this function, we’ll be using three different local variables. The ‘Ret’ will be storing our boolean, which at this point is set to false. ‘Fptr’ is going to be our file pointer name, and ‘counter’ will be used as our counting variable.

```
Begin
```

I hope by this point you have a basic understanding of what the Begin and End do. :)

```
Callerdat:=CFGDataPath+’callers.dat’
```

This line, is setting the Callerdat string to hold the value of the location of our file. The CFGDataPath, is a Mystic function that expands to the path of the /data directory within Mystic. What we’re doing, is just adding the name ‘callers.dat’ to the end of that path. So, on a Linux system, that would actually hold ‘/home/bbs/mystic/data/callers.dat’, and on Windows ‘c:\bbs\mystic\data\callers.dat’, or whatever your actual paths are. This nice thing with doing it this way, is you don’t have to worry about how the Sysop of the other system has their directories set up. As long as Mystic knows the directories, you can use the CFGDataPath, and you’ll have the right directory.

```
fAssign(Fptr,Callerdat,66)
```

As we learned in the previous lessons, the fAssign is actually opening the file, which is listed in Callerdat, and assigning it the name of ‘Fptr’. The 66 at the end is telling the compiler to look for any file that matches the filename we gave it.

```
fReset(Fptr)
```

The fReset is just setting the internal file pointer to the beginning of the file.

```
If IoResult = 0 Then Begin
```

This line is checking to see if the opening of the file, and setting the internal file pointer were both successful. If they are, they give the errorlevel of 0, which is reported by using the IoResult function. If it is anything other than 0, that means something went wrong, and there's a problem.

```
For counter:=1 to 10 do
```

In this line, we are setting up a For/Do loop. Remember those? We will use the variable 'counter' to keep track of how many times we run through this loop. As we know there are 10 records contained within the 'callers.dat' file, we will need to repeat it 10 times.

```
If Not fEof(Fptr) Then
```

This is a simple check to make sure we're not at the end of the file. The fEof function will return 'True' if we are at the end of the file, and 'False' if we are not. So, to translate that line, If we are Not at the End-of-file, then complete the next section.

```
fReadRec(Fptr,Call[counter])
```

I know I didn't cover this command in the previous lesson, as it's not documented very well, anywhere. It is similar to the fRead function that we did talk about, but is only used for reading records. If you remember, the fRead function was formatted as: fRead(Fptr,Call[counter],SizeOf(RecLastOn)). While using this function, you don't need to tell it what the size of the record is, as it will know by the size of the record you are writing it to.

The Fptr is the name we gave our file, and we want it to read the record, and store it into Call[counter]. Call was the name of our array of records, and the [counter] will increase each time the loop is performed. The first record will be read into Call[1], the second into Call[2], etc. (Handy, isn't it?)

Then, after our MPL program has completed the For/Do loop 10 times, we will have all of the information from the file, saved in our array of records. At this time, we close the loop, and move on to the next line.

```
Ret:=True
```

Now that the loop is finished, and our program didn't crash, we will set the 'Ret' variable to true. This line is contained within the If/Then statement, so if the IoResult was false, it will never get to this line of the code.

```
fClose(Fptr)
```

Make sure we close our file. :)

```
ReadCall:=Ret
```

Now that our function has completed everything successfully, we can set the contents of 'Ret' to the name of our function, which is 'ReadCall'. That way, the procedure that is calling this function, can use the True/False to know if this was completed successfully.

That is the end of that function. I hope that helped to explain things a little bit better. There were some questions on using loops to read information from data files, and I thought this would be a good time to cover this information.

Also, in case you’re wondering how I called that function, I used a line like this:

```
If (ReadCall=False) Then progexit
```

This will tell the compiler to run the ‘ReadCall’ function. If it returns ‘True’, continue on with the program. If it returns ‘False’, then run the ‘progexit’ procedure, which will exit the program.

I’ll also show you quickly how I also used a loop to write this information to the screen. It’s pretty much the same setup, but using WriteLn instead of fReadRec.

```
for count:=1 to 10 do
  WriteLn('|09  '+PadRt(Call[count].Handle,22,' ')+Int2Str(Call[count].Node)+'
'+PadRt(Call[count].City,25,' ')+' '+Int2Str  (Call[count].CallNum)+'
'+DateSTR(Call[count].MDateTime,1));
```

(damn word wrap…) That WriteLn statement is all on one line of code.

```
PadRt(Call[count].Handle,22,’ ‘)
```

The PadRt function tells the compiler to write the contents of that variable (Call[count].Handle) into a space 22 spaces long, and fill in the area after the text with spaces (‘ ‘).

I think that’s enough info for this lesson. I don’t want to hear about anyone’s head exploding! ;)