Outpost

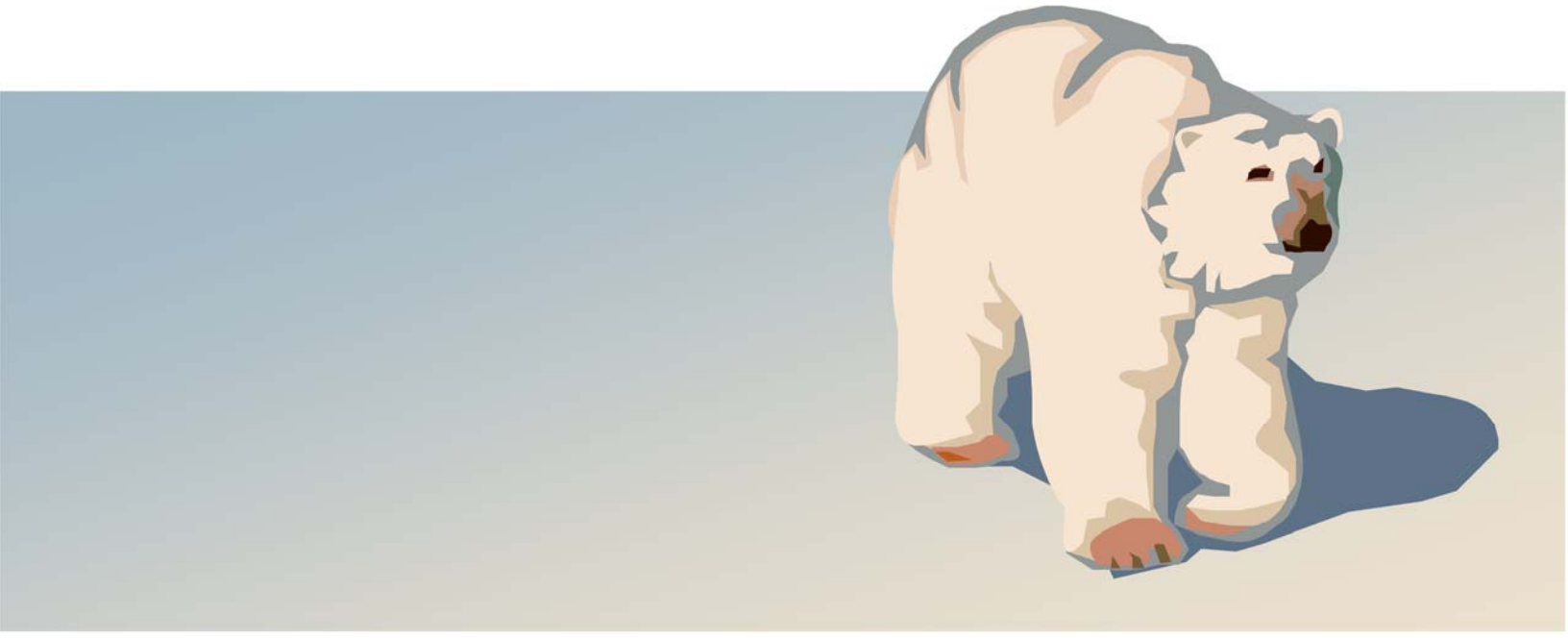
Packet Message Manager

Scripting Language and Processor

Users Guide

June 2023

Version 3.7



Contents

1 About Outpost Scripting

1.1 Introduction

This guide will introduce you to Opscripts and the Outpost Scripting Language, and show you how to create scripts that will control the flow of the **Outpost Packet Message Manager** program.

1.2 What is Outpost Scripts?

If you are reading this, I assume you already have installed and used Outpost. See the Outpost Users Guide and website for specifics on that program.

Over the years, users have requested additional capabilities with Outpost that just didn't make sense to build into the base application due to the extent of the flexibility that was requested and the complexity of the implementation to achieve it. It seemed that a different approach was needed that would allow the user to take advantage of the Outpost operational capabilities without constraining the creative ideas that Outpost users have. This was the start at looking at scripting as a means of extending Outpost's capability.

There are 3 components to the Outpost Scripting feature:

1. **Outpost Scripting Language (OSL)**. This language is made up of a series of commands and capabilities that allow the user develop his/her own capability based on their local needs. Minimally, the command set addresses all of the expressed needs that I've heard of. Commands and statements are entered into a script editing window, can be saved for later use, compiled, and run.
2. **Outpost Script Compiler**. The OSL compiler reads the OSL script and produces a "virtual machine language" output for subsequent processing. During the compilation process, it performs all error checking of the script to ensure that the syntax of the OSL statements are correct. Once complete, the results are reported as either a Pass or a Fail.
3. **Outpost Virtual Machine**. Once the script is compiled, it is then executed within the Outpost Scripts Runtime Monitor window.

All scripting capabilities – editing, compiling, and running – are managed by the Outpost Scripts program – Opscripts.exe. Other than knowing what you want to accomplish and how to write the script, the rest is nothing more than pushing the **Run** button to kick off your script.

1.3 Introducing the Outpost Scripting Language

Currently, there are about 55 commands, functions, and predefined system variables that allow you to control Outpost in ways that cannot be done from the main Outpost user interface.

Because OSL is a language, a basic understanding of programming techniques is helpful. However, between the tutorials and examples shown in this guide, you should be able to get a script developed and running.

The Outpost Scripting Language does enforce a structure that must be followed. After you write your script, the Compile step not only produces the virtual machine code, but also ensures that the script command structure is correct. It will produce error messages if the command syntax is incorrect. In short, if you follow the (syntax) rules, your script will compile.

OSL is made up of several components:

1. **OSL Structure.** The script is defined in a structured approach. As you read through the tutorials, you will see how a script is put together, what statements are required, where specific code goes, and the rules that must be followed to get it to compile correctly.

2. **OSL statements.** OSL statements are a mix of reserved words that guide the execution of the program. Statements include things such as **`IF… THEN… ELSE`**, or **`SENDRECEIVE`**. You will see similarities between OSL and other languages that hopefully will make the learning curve easier.

3. **OSL functions.** Functions are similar to statements except that they usually take one or more parameters that further control their execution, such as **`Play(<filename>)`** or **`NextFileName(0)`**. Some functions also return a result.

4. **OSL System Variables.** Also called reserved variables, OSL pre-defines several variables that specifically support Send/Receive sessions (i.e.: BBS, MYCALL, etc.) and message creation (FROM, SUBJECT, etc.). You choose how, when, and with what value these variables are set based on the flow of your script.

1.4 Notes, assumptions, and disclaimers

1. Before beginning with Outpost Scripts, you must be familiar with setting up Outpost, creating messages and initiating Send/Receive Sessions with a BBS. If you do not know how to do this, DO NOT START working with Outpost Scripting. This application directly builds on your understanding of Outpost.
2. As mentioned, having some type of programming background will help with your Outpost Scripting efforts. If programming is new to you, I recommend you step through the entire Tutorial Series, enter all examples, and play with them to build your comfort level with scripting in general.
3. OSL is not C++, Pascal, FORTRAN, Visual Basic, or any other language. However, you will see similarities as well as differences with other languages with which you may be familiar. If you follow the syntax rules spelled out here, you should not have any problems with Opscripts.
4. Error handling will continue to evolve over time. Most of the errors are properly trapped and reported, however, it is not 100% foolproof. If a compile error points to a line number, you may need to look “in the vicinity” of the code to find the problem.
5. Finally, OSL is an evolving language. If you have an idea that you think would further enhance the usefulness of Outpost Scripting, let me know.

1.5 Find an Error?

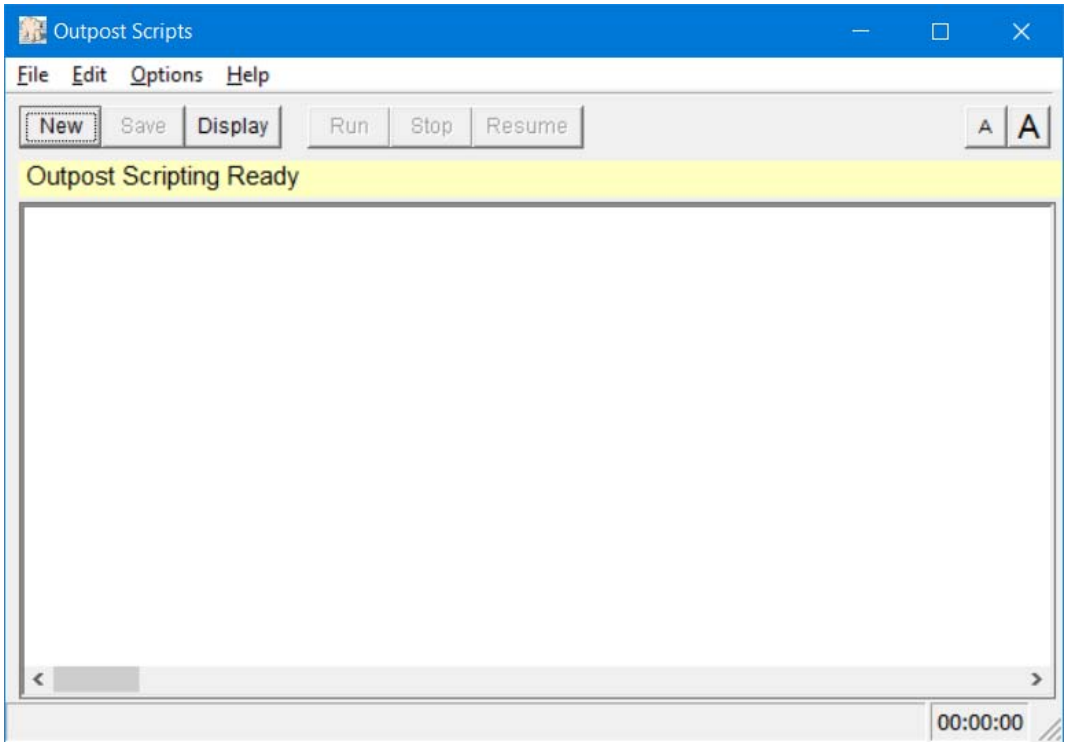
If you find an error or unsure how Outpost Scripting is supposed to run, post a message to the Outpost Users Group or send me email to kn6pe@arrl.net, include the script in question, and any information on what you were attempting to accomplish.

2 Outpost Scripting Overview

All Outpost Scripting activities are managed and controlled by the Opscript.exe program. This chapter provides an overview of the Outpost Script utility and gives you a sense of the navigation.

2.1 One form, many views

When you start the program, you will see the main window that tells you that Opscripts.exe is ready.



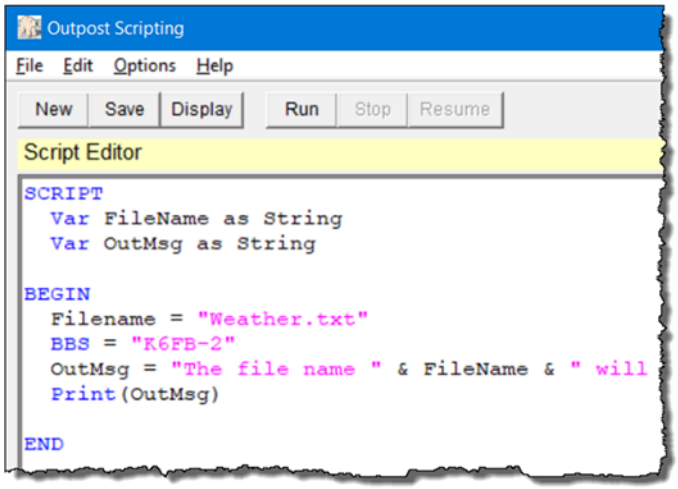
There is only one main form in which you will be operating. However, depending on what you are doing, it will look different. While we will cover the different menus and controls in a moment, the control that changes what you are looking at is the **Display** control.

There are three primary views that you will see.

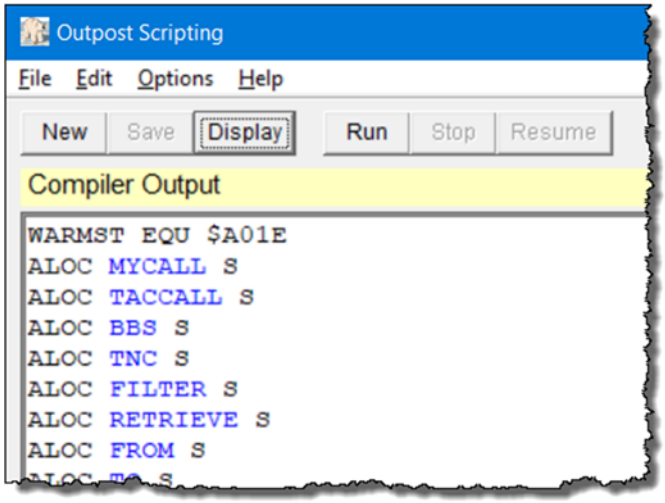
1. Script Editor. When in this view, you have editing access to the text window for creating or updating your script.
2. Compiler Output. When in this view, you can see the compiled output of your script prior to running it.
3. Runtime Monitor. When in this view, you see the execution of your script, the output of any Print statements, and status and update messages displayed by various commands.

By repeatedly pressing the **Display** Control, you cycle through each of the three views in this order. You will always know which view you are in by the changing highlighted label above the text window. Here's what each of the views will look like.

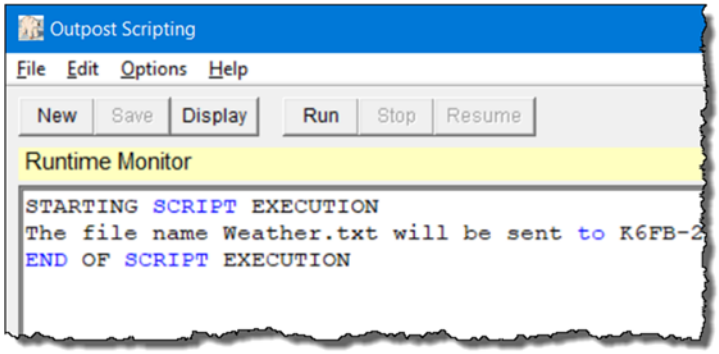
After starting a new script or loading an existing script, the form is set for editing the script.



By selecting the **File > Compile only** menu option, the script is compiled. Press **Display** to rotate to the **Compiler Output** form. You may find this interesting, but not particularly useful.



Pressing the **Run** control causes the script window to shift to the **Runtime Monitor** and begin executing the script. All of these views can be seen sequentially by pressing **Display**.



2.2 Menus and Toolbars

The *Program Controls* portion of the User Interface controls the operation and execution of all program tasks.

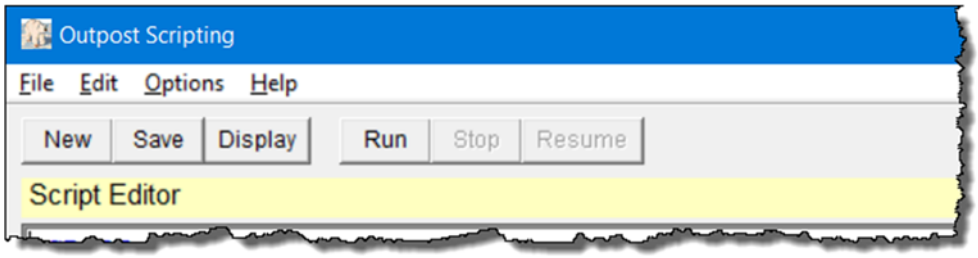


Figure 1: Main Screen Menu and Tool Bar

The Outpost menus provide different options for setting up and controlling the application.

NOTE: Some of the more common menu items are also implemented as Tool Bar buttons. See the associated menu item below for their description.

Menus	Description

File	**New:** Sets up for a new script to be entered. Automatically adds the 3 lines that that every script needs to have…
	`SCRIPT`
	
	`BEGIN`
	
	`END`
	
	Open: Opens a form that allows the user to select a previously existing script for editing or running.
	
	Save: Saves the script back to the Scripts directory. This option is only enabled after a **New** script has been started or a script has been retrieved via an Open command.
	
	Save As: Opens a form to allow the user to select a directory and enter a file name to save this script. The script is saved as an ASCII text file. Script files default to a `<script_name>.txt` file extent.
	
	Compile: Causes the script to be compiled only. The script is not run. The Output of the compiler is stored in a file named `<script_name>.ocs` .
	
	Run: Causes the script to be compiled, and then run. If the script was previously compiled, the script is only run.
	
	Most recently used: Displays the list of scripts previously opened. Up to 10 entries can be added.

Edit	**Find/Replace**: Used to search the script for a string. When the "Replace With" field is filled in, string substitutions can be made with one or all strings being updated.

Menus	Description
	Cut: Copies and deletes any highlighted text from the script editor. The text is placed in the MS-Windows clipboard. **Copy**: Copies any highlighted text in any from the script editor. The text is placed in the MS-Windows clipboard. **Paste**: Inserts text from the clipboard at the position where the cursor is located. **Clear All**: Erases the contents of the script editor. **Select All**: Selects all text in the script editor.
Options	☑ **Run Debug**: Dumps the initialization of the script to the Runtime window. ☑ **Verbose**: When checked, generates more detailed information when this option is checked. Default is not checked. ☑ **Exit Enabled**: When checked, enables the EXIT statement so that when encountered in a script, the script will end and the program is exited. Default is not checked. ***NOTE:*** Only set this option AFTER you save any changes and when you are sure your script runs per your liking. ☑ **AutoRun Hidden**: When checked and when Outpost runs a script as part of the Send/Receive settings (see Outpost > Tools > Script Settings), causes the program to run in the background. No interaction with the script can occur. Default is not checked. ☑ **Word Wrap**: When checked, automatically moves a word from the end of one line of text to the beginning of the next when there is insufficient space.

The following are other controls found on the main form.

Commands	Description
Display	Changes the text display in the following order: 1. Script Editor 2. Compiler Output 3. Runtime Monitor Continuing to press Display continues to rotate through these views.
Run	Causes the script to be compiled, and then run. If the script was previously compiled, the script is only run.

Commands	Description
Stop	Stops the script from running. Press Run to restart the script from the beginning.
Resume	Resumes execution of the script after the Pause(0) script command (pause and wait for user interaction) is executed.

2.3 Running Opscripts.exe

There are 3 ways to run the Opscripts program:

1. From Outpost. From the Outpost main menu, select Tools > Scripts… select Scripting Tool.
2. From Windows Explorer. Navigate to the Outpost program directory, and double-click on the Opscripts.exe icon.
3. From a user defined .bat file. Opscripts can be run outside of Outpost with optional command line parameters and switches that are passed to the program. Run command format is:

   ```
   Opscripts.exe [ command_line_options ]
   ```

   Available parameters and switches

-f <filename>	Starts Opscripts with the named script file. When loaded, the script automatically compiles and runs.
-d	Turns Debug on. This is the same function as selected from the Options menu described above.
-v	Turns Verbose on. This is the same function as selected from the Options menu described above.
-hide	Turns AutoRun Hidden on. This is the same function as selected from the Options menu described above.

3 Tutorials

This chapter includes 13 tutorials that will walk you through many of the OSL statements, functions, and commands. The tutorials are as follows:

3.1. **Writing your first script**. This session gives you a basic overview of the Opscripts.exe main window, describes some of the controls, and gives you a chance to create and run a very simple script.

3.2. **OSL components**. This section overviews all the components that that makes up OSL.

3.3. **Playing with strings.** This is an overview of how OSL can use strings to enhance the overall control and presentation of information.

3.4. **Arithmetic operations.** This session describes the use of mathematical equations to perform simple calculations. When dealing with packet, there really should be nothing more complicated than incrementing a counter.

3.5. **Looping forever.** This session shows how to set up unconditional looping on a block of code. Other examples of this will show up in later sections.

3.6. **Conditional looping.** This session describes the statements that let you loop on a block of code as long as a specific condition is met.

3.7. **Other conditional operations.** The IF… THEN… ELSE series of statements are described and show how checks can be made and acted on based on the outcome of the test.

3.8. **Send/Receive sessions.** One of the biggest reasons for Outpost scripting is to support the ability to select and poll different BBSs automatically. This will be introduced here.

3.9. **Creating messages**. This session shows how you can create a valid Outpost message from within a script and prepare it for sending.

3.10. **Working with received messages**. As a follow-on to the previous session, this section shows how received Outpost messages can be identified, accessed, and acted on, such as for forwarding, moving, or saving to a disc file.

3.11. **File Manipulations.** This session describes how to use files in support of all of the above.

3.12. **Interacting with the outside world.** There are a series of statements that help you do things outside of Outpost Scripting.

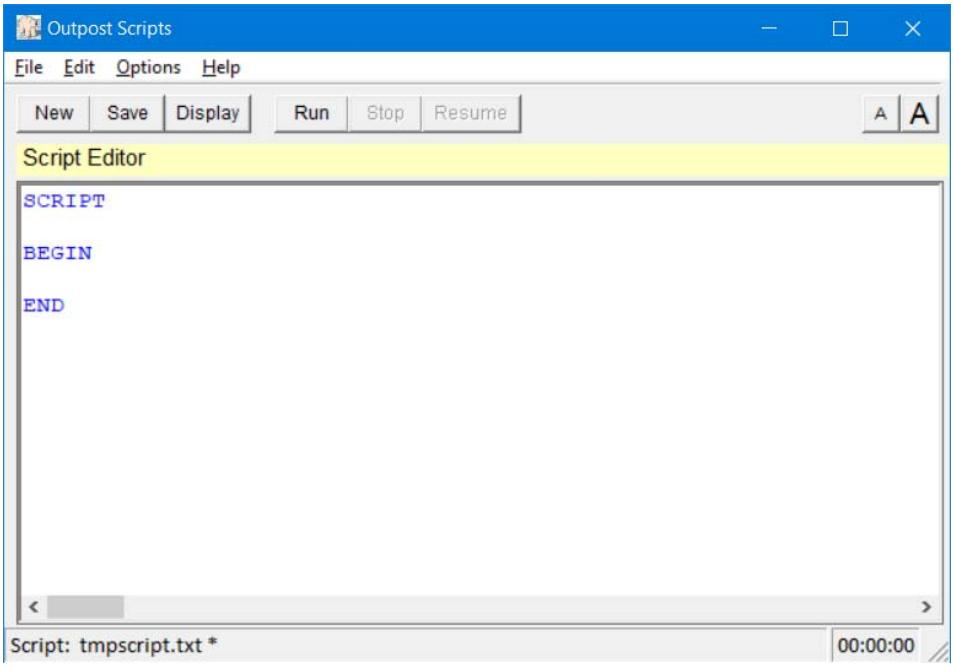
3.13. **Outpost-initiated scripts.** Outpost can call Opscripts with a script name when starting or stopping Outpost, or as an alternative to the standard Send/Receive process.

3.1 Tutorial #1 – Writing your first script

Outpost Scripting (Opscripts) and the Outpost Scripting Language (OSL) extend the automation provided in the Outpost application by allowing the user to manipulate different Outpost settings outside of the program with user-defined scripts. This first tutorial will guide you though some of the OSL basics that you will need to know.

Getting Started

1. Run Outpost. From the Tools Menu, select Scripting. Alternatively, using Windows Explorer, you can navigate to the Outpost programs directory. Verify that the **Opscripts.exe** program is there, and then double-click on the file. You should get the following display. Note that 2 options are available: **New**, and **Display**.



2. Press **New**. This button sets up Opscripts for entering a new script. After pressing **New**, 2 things happen:
    i. The window header changes to **Script Editor**, and
    ii. Three lines are inserted in the script edit window.

```
SCRIPT

BEGIN

END
```

All Outpost scripts contain these 3 lines that are the absolute minimum for a script to run.

i. Every script must have the word SCRIPT as the first executable line.

ii. The BEGIN statement marks the beginning of where the script statements are placed.

iii. The END statement must be the last statement in the file.

3. Press **Run**. With these 3 lines, the script gets compiled, the view shifts to the **Runtime Monitor** window, and the script starts running:

```
STARTING SCRIPT EXECUTION
END OF SCRIPT EXECUTION
```

Congratulations! You just created your first script! … and by pushing only the **New** and **Run** buttons.

Not surprising, this script does… nothing! But it did run. Try removing any of these lines and press **Run** to check out the different error messages.

4. Press **Display**. Note that repeatedly pressing **Display** rotates around 3 views: the **Script Editor**, the **Compiler Output**, the **Runtime Monitor**, and then back to **Script Editor**.

5. To make this script a bit more interesting, let's try to do something simple (we will discuss all these statements in later sessions).

   Press **Display** to get back to the **Script Editor** window; make the following changes EXACTLY as shown:

```
SCRIPT
VAR x AS NUMBER

BEGIN
  X = 5
  Print("Hello World!")           ' Prints to the display
  Print("The value of x is " & x) ' Prints a string and number
END
```

6. Press **Run**. Every time the script is changed, it gets compiled, the status is indicated in the status line at the bottom, and then the script is run. Note that the Display window header changes from **Script Editor** to **Runtime Monitor**. The following is the output from this run:

```
STARTING SCRIPT EXECUTION
Hello World!
The value of x is 5
END OF SCRIPT EXECUTION
```

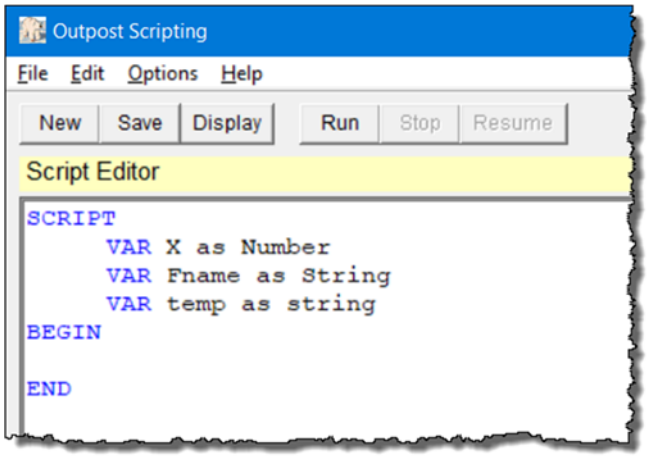
7. To save this script, press **Display** until you are at the **Script Editor** view, press **File > Save As**, select the directory and file name, such as `Tutorial-01.txt`, then press **OK.**

3.2 Tutorial #2 – OSL components

While these first scripts were not too exciting, they do start to highlight some of the capabilities that OSL offers.

1. **Numbers and Strings.** The two types of data that OSL supports are numbers (any positive or negative number, includes decimals) and strings (ASCII characters surrounded by quotations).

2. **User and System Variables.** OSL lets you define variables that can be assigned and accessed throughout your script. Variable declarations are the first real commands that you enter in the script. They are placed between the **SCRIPT** and **BEGIN** statements.



In this example, 3 user-defined variables are defined: "**X**" is defined to hold a number, and "**Fname**" and "**temp**" are defined to hold strings. Any number of variables can be defined.

3. **Assignments.** With your variables defined, OSL lets you assign data to these variables. Assignments could be as simple as:

`FName = "WX.txt"`	assigns a file name to a user-defined variable
`counter = 1`	assigns a number to a user-defined variable
`BBS = "K6FB-2"`	assigns a name to a system-defined variable

4. **Expressions.** OSL also lets you to do more than just simple assignments. You can write your own arithmetic or string expressions that support some aspect of script control and execution. Expressions could be as simple or as complex as necessary:

`X = X + 1`	a simple way to create a counter
`X = Y/(5-Z)`	a more complex arithmetic expression
`Msg = "file is " & FName`	strings can be assigned and manipulated

5. **Statements**. There are several OSL statements that help control how the script behaves. Statements are divided into 4 different categories:

General	Message	Session	Conditional
Begin	CreateMessage	SendReceive	If… Else… Endif
End		SendOnly	Loop… EndLoop
Exit			While… EndWhile
Beep			OnError
Clear			
Print			
Script			
Var			

6. **Functions**. OSL Functions are similar to statements, except they can take a parameter on which they may act. OSL has 4 different categories of functions:

General	File	I/O	Message
Pause	Delete	Play	FindMessage
FindWord	Exists	Print	MoveMessage
NextWord	FindFile	Run	NextMessage
Now	GetFileName	Runw	Expire
	MoveFile		
	NextFile		
	ReadFile		
	WriteFile		

7. **System-defined variables**. OSL predefines 16 System Variables that the user can use to help set up various script functions.

Message	Session	System
BBS	BBS	TRUE
FROM	TNC	FALSE
TO	MYCALL	ON
SUBJECT	TACCALL	OFF
MESSAGE	RETRIEVE	CRLF
MTYPE	FILTER	
URGENT	SRNOTE	
BBSMSGNO		
LMI		
RECEIPTS		
DATETIME		

8. **Comments in the Scripts.** You noticed in Tutorial 1 that there are comments on a couple of the lines. All comments begin with an apostrophe " ' " and end at the end of the line with a Carriage Return (Enter Key). Comments can be inserted anywhere on a line, and the file can contain any number of them. All comments are ignored during compilation making them a great way to document your scripts.

NOTE: As you start writing scripts, I strongly recommend that you put comments in them; it will make it easier for you as well as for the next person who comes along to understand what you wanted to do.

NOTE: See the Command Reference later in this document for a description of the above statements and commands. We will try many of them in the following sections. However, some of the more intuitive commands I will leave up to you to explore.

3.3 Tutorial #3 – Working with Text Strings

So, with some fundamentals behind us, let's jump into OSL. At the end of this tutorial series (all of them), I hope to have introduced many of the commands and concepts that will help you get your own scripts up and running. Strings are as good a starting point as any.

Definition: A string is one or more ASCII characters that has some meaning to the user either as printable text or as a value to be assigned to a variable. Once a text string is defined, it can be assigned to a variable, appended to another string, and/or printed.

Getting Started

1. All strings start and end with double quotation marks. For instance, the following are examples of strings:
   - `"Beginning Send/Receive Session # "`
   - `"K6FB-2"`
2. Strings can be assigned to a variable to be used over and over again. For example:

```
SCRIPT
Var FileName as String
Var OutMsg as String

BEGIN
  FileName = "Weather.txt"
  BBS = "K6FB-2"
  OutMsg = "The file name " & FileName & " will be sent to " & BBS
  Print(OutMsg)
END
```

3. In the above example, 2 variables -- `FileName` and `OutMsg` – are defined as user variables of type **String**. This means that only strings (characters surrounded by quotations or other string variables) can be assigned to them. With these declarations, they can then have strings assigned to them. For instance,
   - `FileName`, a user-defined variable, is assigned the string `"Weather.txt"`
   - BBS, a system-defined variable, is assigned the string `"K6FB-2"`
   - `OutMsg`, another user-defined variable, is assigned a mix of strings and variables that were concatenated together.
4. Let's try it: Enter the above script by first pressing **New**, and then type it in exactly as shown. Once entered, press **Run** to compile and run the script. It should produce the following output:

```
START OF SCRIPT EXECUTION
The file name Weather.txt will be sent to K6FB-2
END OF SCRIPT EXECUTION
```

Concatenating Strings

Take a look at the script assignment to `OutMsg`. You may have observed this line is assigned more than one thing. Two or more strings or variables holding strings can be joined together or "concatenated" using the ampersand (" & ") sign. Just as the PLUS sign (+) adds 2 numbers together, the ampersand (" & ") sign lets you join 2 or more strings together. This is an important concept since it does allow you to mix text, variables, and calculated results together in a single text line.

Really Long Strings

5. There could be situations where you may need to create a really long string and to keep it relatively neat in the script. Here's an example of creating a multi-line string on the fly.

```
SCRIPT
VAR OutMsg as string

BEGIN

OutMsg =“Hi Cap,” & CRLF &
        “This is an OSL-generated message.” & CRLF &
        “  73, Jim”
Print(OutMsg & CRLF)

OutMsg = “In this case, the string is longer than the form ” &
 “and I want it to wrap around the Runtime Monitor.  Note that ” &
 “OSL continues to process the same assignment to OutMsg ” &
 “across multiple lines.”
Print(OutMsg)

END
```

6. After entering this script, here's the output:

```
STARTING SCRIPT EXECUTION
Hi Cap,
This is an auto-generated message.
  73, Jim

In this case, the string is longer than the form and I want it to
wrap around the Runtime Monitor.  Note that OSL continues to
process the same assignment to OutMsg across multiple lines.
END OF SCRIPT EXECUTION
```

7. In the first Print Statement, we used a system variable named **CRLF**. This is an OSL System Variable that inserts a Carriage Return/Line Feed combination on a line of text. It is joined to other portions of the string using our trusty “&”.

8. In the second Print Statement, note that one very long run-on string of text is created. Because OSL expects to see a closing quotation for a text string before the end of a line, to create really long strings, end the text string with quotation marks, add an ampersand mark to indicate a continuation, and on the next line, continue the text string within quotation marks.

9. While the above method works well, the other approach you could use is to put the text in a file, and then read the file into the string variable.

```
SCRIPT
VAR OutMsg as string

BEGIN

OutMsg = ReadFile(“MsgToCap.txt”)
  :
```

We will cover the `ReadFile` statement in a latter tutorial.

3.4 Tutorial #4 – Arithmetic Operations

Not that there is a lot of need to perform heavy math with packet radio, there may be some situations where you need to do some basic arithmetic in your script.

Definition: A number is any combination of digits. OSL supports addition (+), subtraction (-), multiplication (*), division (/) and modulo (%) operations on numbers and variables that store numbers. It also enforces operational precedence for execution.

Getting Started

1. Here are some examples of math operations:

```
SCRIPT
Var X as NUMBER

BEGIN
X = 0                    ' initially set X to 0
Print("x = " & x)         ' the result should be 1

X = X + 1                ' add 1 to X
Print("X + 1 = " & x)     ' the result should be 1

X = X*5+10               ' x was 1 from the previous calc
Print("X*5+10 = " & x)    ' now, the result should be 15

X = X*(5+10)             ' x was 15 from the previous calc
Print("X*(5+10) = " & x)  ' now, the result should be 225

Print("Modulo: 8 % 3 = " & 8 % 3)      ‘ some modulo math
Print("Modulo: -8 % (1+2) = " & (-8) % (1+2))
Print("Modulo: 8 % (-3) = " & 8 % (-3))
Print("Modulo: (-8) % (-3) = " & (-8) % (-3))
END
```

2. Enter the above script (press **New** first) and press **Run**. Here's what you will see:

```
STARTING SCRIPT EXECUTION
x = 0
X + 1 = 1
X*5+10 = 15
X*(5+10) = 225
Modulo: 8 % 3 =    2
Modulo: (-8) % 3 = -2
Modulo: 8 % (-3) = 2
Modulo: (-8) % (-3) = -2
END OF SCRIPT EXECUTION
```

3. The first calculation (x = x + 1) is a good way to create a counter when you want to keep track of how many times you have done something.
4. The 2nd calculation takes the result of the first calculation and uses it in this equation. Note how the precedence was enforced: x = x*5+10: with x as 1, we first multiple 1*5 (= 5), then add 10, equaling 15.
5. In the 3rd equation, with x previously calculated as 15, the parenthesis causes 5+10 to be executed first (equals 15), then multiplied by x (15) equaling 225.
6. Modulo returns the remainder after number1 is divided by number2.
7. Note the handling of a number that is to be evaluated as a negative number.
8. Variables are not case sensitive. The variable "X" can be referenced as both "X" (upper case) and "x" (lower case).

3.5 Tutorial #5 – Looping forever

More than likely, you may want to set up a script that repeats a task over and over again. You can do this with the **LOOP… ENDLOOP** statements. When using these statements, you are performing UNCONDITIONAL looping, meaning, it will continue to loop regardless of what else is going on.

Getting Started

1. Here's an example:

```
SCRIPT
Var x as NUMBER

BEGIN
x = 0                              ‘ set x to 0

‘ Loop on the following code.  Press STOP on the Runtime Monitor
‘ to stop processing

LOOP
  x = x + 1                        ‘ count the # of passes
  Print(“This is pass # ” & x)     ‘ write the results
  pause(10)                        ‘ pause for 10 seconds
ENDLOOP

END
```

2. Enter this script and press **Run**. Here's what you should see:

```
START OF SCRIPT EXECUTION
This is pass # 1
Pausing for 10 seconds
This is pass # 2
Pausing for 10 seconds
This is pass # 3
Pausing for 10 seconds

(I pressed STOP at this point)
```

3. After a couple of passes, press **Stop** to stop the script from running.

LOOP… ENDLOOP statements

The LOOP… ENDLOOP statements are a set that work together. Any statements in between them are part of the loop. If the ENDLOOP statement is omitted, an error occurs.

Note how the statements between the LOOP and ENDLOOP statements are indented. While not necessary, indenting helps improve readability of the script. You can either enter spaces (press the space bar) or press the TAB key.

The LOOP… ENDLOOP command set will loop forever. Because there is no programmatic way in the script to stop processing or break out of the loop, the only way to stop the script is by the user pressing the **Stop** button on the Runtime Monitor form.

The LOOP… ENDLOOP command set is an excellent way to set up a re-occurring task that you want to perform. Later on, you will see how we will use it to control polling a series of BBSs that exist on the same frequency.

Pause statements

In the above script, we introduced the PAUSE(<seconds>) statement. This statement causes the script to pause execution for the number of seconds listed within the parenthesis. So, a value of 10 means 10 seconds; 600 = 10 minutes (60*10=600); 36000 = 10 hours (60*60*10=36000). The Pause function writes its own message letting you know how long the script is about to be paused. It also initializes the count-down timer in the lower right corner of the form that shows the time left for this pause operation.

WARNING! It is strongly advised that you put at least a 1 second pause in any LOOP that you create to ensure the script does not "run-away" (you lose control to stop).

If the Pause function parameter is set with a "0" such as Pause(0), then the script will stop and prompt for the user to press the **Resume** button on the Runtime Monitor window. A Pause(0) implementation would show the following:

```
START OF SCRIPT EXECUTION
This is pass # 1
Paused... press "Resume" to continue
```

3.6 Tutorial #6 – Conditional Looping

OSL also supports looping as long as a specific condition is met. This is referred to as Conditional looping: as long as the condition is true, the loop continues.

Getting Started

1. Here's an Example

```
SCRIPT
Var X as NUMBER

BEGIN
X = 0                      ‘ set X to 0

WHILE X < 3                ‘ loop as long as X is less than 3
  X = X + 1                ‘ increment X
  Print(“This is pass # ” & X )
ENDWHILE

END
```

2. Enter the above script and press Run.

```
START OF SCRIPT EXECUTION
This is pass # 1
This is pass # 2
This is pass # 3
END OF SCRIPT EXECUTION
```

WHILE… ENDWHILE statements

The WHILE… ENDWHILE statements are another pair that work together. Any statements put between these statements will be executed as part of the loop as long as the condition on the WHILE statement is TRUE. If the ENDWHILE statement is omitted, an error occurs.

Conditions are based on relationships between both sides of an arithmetic expression. Valid OSL relationship operators are:

<	is the left side	less than	the right side?
<=	is the left side	less than or equal to	the right side?
>	is the left side	greater than	the right side?
>=	is the left side	greater than or equal to	the right side?
=	is the left side	equal to	the right side?

Think through the logic of why three passes were executed: before the 1st pass, “X” was 0… the condition is true (0 is less than 3). However, once in the loop, X is incremented to 1 and “Pass 1” is printed.

On the 2nd pass, the condition is also true (1 is less than 3), however, X is incremented to 2 and “Pass 2” is printed.

Note that the Pause function was not needed since there was a specific condition that would be met to exit the loop.

Think about this…

- Getting the initial setting of “X” correct (X = 0? X = 1?), as well as the condition to check, is critical for accurate execution of conditional loops.

- What happens if you put the counter step (x = x + 1) after the Print statement? Try this. Can you explain the results? You can place the counter wherever you wish and may need to adjust the `WHILE` condition to get the results you want.

Another type of test

3. Here is another example of conditional looping:

```
SCRIPT
Var FNAME as string

BEGIN
  FNAME = "Weather.txt"          ' initialize with a file name

  WHILE Exists(FNAME) = TRUE     ' loop as long as the file exists
       Print("The file " & FNAME & " is still there")
       pause(60)
  ENDWHILE

END
```

4. In this case, instead of checking for an arithmetic condition, we are checking for a file condition. We will look at the Exists function in a later section, but this is to demonstrate that checking the result of a function call can also be used as part of conditional check.
5. Regarding **TRUE**. True is a system variable against which functions like `Exists` can be tested. **FALSE** could also be used for tests like these.

3.7 Tutorial #7 – Other Conditional Operations

One of the standard conditional checks that exist in many program languages is the IF… THEN… ELSE… statements. OSL supports this construct that can be used to branch depending on some condition.

Getting Started

1. Here's an example:

```
SCRIPT
Var x as NUMBER

BEGIN
  x = 3
  IF x > 5 THEN
       Print("X is greater than 5")
  ENDIF
END
```

2. Not to surprising, when you load and run this, you get this reply:

```
STARTING SCRIPT EXECUTION
END OF SCRIPT EXECUTION
```

3. Note how the IF statement tested to see if X is GREATER THAN 5. In the above example, the answer was NO, and the condition was not true. As a result, the code immediately following the IF statement was skipped. If the condition was TRUE, then the code would have been executed.

4. The IF… THEN could also be written with the ELSE option…

```
SCRIPT
Var x as NUMBER

BEGIN
  x = 3
  IF x > 5 THEN
       Print("X is greater than 5")
  ELSE
       Print("X is not greater than 5")
  ENDIF
END
```

5. At least with this script, it is a bit more interesting. Enter and run it…

```
STARTING SCRIPT EXECUTION
X is not greater than 5
END OF SCRIPT EXECUTION
```

IF… THEN… ELSE… ENDIF statements

The minimal structure is IF… THEN… and ENDIF. They are best read as follows (using the above example):

"IF X is greater than 5, THEN"… because the condition is TRUE, you execute the code immediately in the next line. However, if the condition is FALSE (as it was above), the code following is skipped.

Using ELSE is optional and allows for some other action to occur if either the condition is **TRUE** or **FALSE**.

3.8 Tutorial #8 – Send/Receive Sessions

Finally, we are getting to something specific to Outpost. One of the original requests was for Outpost to poll a variety of BBSs without any user interaction.

Getting Started

1. Here's an Example

```
SCRIPT
BEGIN
  BBS = "K6FB-2"
  TNC = "GARAGE-TNC"
  MYCALL = "KN6PE"
  RETRIEVE = "PB"
SENDRECEIVE

END
```

2. It is that simple. Give it a try, except use a BBS and TNC Interface name that you have set up on your Outpost system. Make sure Outpost is running, then press Run.

```
STARTING SCRIPT EXECUTION
BbsName = K6FB-2
TncName = KPC3
StationID = KN6PE
TacCall disabled
Retrieving = PB
Filter =
Initiating a Send/Receive Session
END OF SCRIPT EXECUTION
```

System Variables

The **BBS**, **TNC**, **MYCALL**, and **RETRIEVE** are system-defined variables. You assign them values, and they then can be used to set up a valid Send/Receive Session or as part of some other output process. These 4 entries minimally identify what must be set up for the Send/Receive session.

SENDRECEIVE Statement

The **SENDRECEIVE** statement is used to initiate a Send/Receive session with Outpost. It takes the system variables defined above and passes them to Outpost. Outpost then changes its configuration to match what is defined here and executes a send/receive session with these settings. Outpost Scripting then waits for the Outpost session to end before continuing to execute.

For Retrieving, you can enter any combination of these 4 characters: P (Private), N (NTS), B (Bulletin), or F (Filtered). See the Outpost users guide for more details on these options, and the command reference for all System variables.

3. While the above was fun, I don't need OSL to check a single BBS. So, lets combine a couple of statements that we learned about to make it more interesting:

```
SCRIPT
BEGIN
LOOP
  BBS="K6FB-2"; TNC="GARAGE-TNC"; MYCALL="KN6PE"; RETRIEVE="PB"
  SENDRECEIVE
  Print(" ")

  BBS="W6SJC-1"; TNC="GARAGE-TNC"; MYCALL="KN6PE"; RETRIEVE="P"
  SENDRECEIVE
```

```
  Print(" ")

  Pause(600)
ENDLOOP
END
```

4. When this is loaded and run, here's the output.

```
STARTING SCRIPT EXECUTION
BbsName = K6FB-2
TncName = KPC3
StationID = KN6PE
TacCall disabled
Retrieving = PB
Filter =
Initiating a Send/Receive Session
Send/Receive Session complete!

BbsName = W6SJC-1
TncName = KPC3
StationID = KN6PE
TacCall disabled
Retrieving = P
Filter =
Initiating a Send/Receive Session
Send/Receive Session complete!

Pausing for 600 seconds
```
(I pressed stop at this point)

5. This script now starts to look useful. Here's a couple of things to note:
   - **Multiple statements** on one line. Yes, OSL supports this. You can separate statements with a semi-colon. The above example is 1 of the 2 times where I think this is acceptable since it keeps the script a bit more visually compact.
   - The use of the **LOOP / ENDLOOP** and the **PAUSE** statements sets up the script to loop on these 2 BBSs. 600 seconds is 10 minutes.
   - Along with changing the BBS name, you could also change the MYCALL to check for messages sent to a Club call sign, or some other member.
   - The **PRINT** statement is used more for formatting. In this example, I wanted a blank line between session listings.
   - Above all, the BBS and TNC must be previously setup in Outpost (**Setup > TNC** and **Setup > BBS**) before running this. If either are not setup, Outpost will pop up a window and report that the BBS or TNC are missing.

WARNING! This missing BBS or TNC warning may not pop to the front of the Scripting form!

6. What happens if you want to check a BBS on a different frequency? OSL cannot change your radio's frequency (yet!). However, it could tell you to do so. Here's a script snippet that addresses this:

```
BBS="K6FB-2"; TNC="GARAGE-TNC"; MYCALL="KN6PE"; RETRIEVE="PB"
SENDRECEIVE
Print("Change the Frequency to 144.970")
Pause(0)

BBS="W6SJC-1"; TNC="GARAGE-TNC"; MYCALL="KN6PE"; RETRIEVE="P"
   :
```

7. At runtime, the user is now prompted to change the frequency. Once changed, the user then presses the **Resume** button on the Runtime monitor form.

3.9 Tutorial #9 – Creating Messages

There are times when you may want to create a message automatically based on some event that you detected. For instance, one Outpost user wanted to post NOAA weather bulletins to the packet community whenever the message arrived over the internet (he figured out how to get a text file off of a website and onto his PC for Outpost to pick up, not described here).

Getting Started

1. Here's an example:

```
SCRIPT
BEGIN

  BBS="K6FB-2"
  FROM="KN6PE"
  TO="KE6AFE"
  SUBJECT="Status of the system"
  MTYPE="Private"                ' Private message
  RECEIPTS="R"                   ' Request a Read Receipt
  MESSAGE="Hi Cap, this is an auto-generated message.  73, Jim"
  CREATEMESSAGE

END
```

2. Give it one a try. When you press Run, you see the following:

```
STARTING SCRIPT EXECUTION
Initializing a new message
Saved new message ID=22
END OF SCRIPT EXECUTION
```

3. Similar to the last section, **BBS**, **FROM**, **TO**, **SUBJECT**, **MTYPE**, **RECEIPTS**, and **MESSAGE** are system-defined variables. You set them here, then when executing the **CREATEMESSAGE** statement, a valid Outpost message is created and stored in the Outpost message database.
4. **MYTYPE** needs to be set to 1 of 3 values: **PRIVATE**, **NTS**, or **BULLETIN**. If none is specified, then it defaults to PRIVATE.
5. In the above example, if you were to run Outpost and navigate to the Out Tray, you would see this message loaded and ready to be sent.
6. Because one-line messages are not always sufficient, OSL does let you read the message text from a file. For example, suppose you have a message in the file **WxReport.txt** that you want to send. You can do the following:

```
  BBS="K6FB-2"
  FROM="KN6PE"
  TO="ALLUSR"
  SUBJECT="Weather Bulletin"
  URGENT = TRUE
  MTYPE="Bulletin"
  MESSAGE=READFILE("WxReport.txt")
CREATEMESSAGE
```

7. In this instance, the **READFILE** function is used to read the contents of the file WxReport.txt and store its contents into the MESSAGE variable.

NOTE: Going back to the original statement regarding the NOAA weather bulletins, check out ***Script Example #3*** for a full view of what the script would look like. Again, it assumes that you figured out how to write a file to a location that Opscripts can find and load.

3.10 Tutorial #10 – Working with Received Messages

What happens if you are looking for a message coming in to Outpost and want to do something with it? There are a series of statements and functions that help you do things with messages that are received by Outpost.

NOTE: Before proceeding, I recommend reading the **Command Reference** on these statements: `FindMessage`, `NextMessage`, and `MoveMessage`.

1. Here's the example:

```
SCRIPT
VAR MsgID AS NUMBER
BEGIN

‘ Look for a message in the In Tray from K6KP; forward it to KN6PE
FindMessage(1, 2, “K6KP”)
MsgID = NextMessage(0)

While MsgID > 0
    TO = “KN6PE”         ‘change the destination
    MESSAGE = “OSL-forwarded from K6KP” & CRLF & MESSAGE
  CreateMessage

  SendReceive
  MoveMessage(MsgID, 4)  ‘ Move original message to the Archive
  MsgID = NextMessage(0)

EndWhile
 END
```

2. The **`FindMessage`** and **`NextMessage`** statements are another pair of statements that work together. They allow you to search for one or more messages that match a certain string pattern for any field that constitutes a message, such as BBS, From, To, Subject, and Message field.

   FindMessage needs 3 parameters to be set.

   #1 Folder: this parameter is a number that corresponds to an Outpost folder to search. Valid numbers are:

   1. InTray
   2. Out Tray
   3. Sent Folder
   4. Archive Folder
   5. Draft Folder
   6. Deleted Folder
   11. Special Folder #1
   12. Special Folder #2
   13. Special Folder #3
   14. Special Folder #4
   15. Special Folder #5

   #2 Field: this parameter is a number that corresponds to an Outpost Message field to search. Valid numbers are:

   1. BBS
   2. FROM
   3. TO
   4. SUBJECT
   5. MESSAGE

   #3 Pattern: this parameter is a string pattern to match. Wildcard use (KN6*) is allowed. The pattern to match is not case sensitive and must be surrounded by double-quotation marks.

The following are the pattern match characters that are supported:

Characters in Pattern	Matches in String
?	Any single character
*	Zero or more characters
[charlist]	Any single character in charlist
[!charlist]	Any single character not in charlist

NOTE: See the FindMessage() for more information and pattern examples in ***Section 5 Command Reference***.

3. For instance, as the above script describes, suppose you are looking for messages from K6KP. The **FindMessage** Command sets up Opscripts to look in the In Tray (1st parameter, the "1"), at the From Field (2nd parameter, the "2"), and look for matches of "K6KP".
4. We want to forward any message from K6KP to KN6PE. The **FindMessage** sets up the message search, and the 1st NextMessage statement attempts to retrieve the Outpost Message Identifier. This Message ID is an internal Outpost pointer to an Outpost message. Valid Message IDs are any number that is not zero (0).
5. The **NextMessage** statement loads all fields of the message into the Opscripts variables: **BBS**, **FROM**, **TO**, **SUBJECT**, **MESSAGE**, **DATETIME**, **LMI**, **MTYPE**. So, in this case, to forward the message, all we need to do is change the destination ("TO = KN6PE") and issue the CreateMessage statement. A new message is created with all the other fields.
6. Finally, we use the **MoveMessage** statement to move the original message from the Intray (where we found it) to the Archive Folder so that we do not detect it again when the script loops. See the Command Reference for details on the MoveMessage statement.
7. The upside of the FindMessage command is that you can actually search the body of the message for a string and trigger the forwarding or storing event on that. However, this takes coordination to ensure that the message originator puts in the string, and the Opscript is set up to look for it.
8. See ***Script Example #4*** for a more filled in view on how this set of OSL commands could work.

Pattern Matching with Regex (v370)

1. Along with the above simple pattern matching, FindMessage also support Regex. To invoke a Regex expression, a tag "**(?#REG)**" is placed in front of the expression so that the FindMessage statement would look like this:

```
FindMessage(6, 5, "(?#REG).?(\d{3}).?(\d{3}).?(\d{4})")
```

NOTE: It is beyond the scope of this tutorial to cover Regex in detail. However, if you are familiar with the Regex concepts, then implementation should be straight forward.

3.11 Tutorial #11 – File Manipulation

We already touched on a couple of File commands that you may find useful. You could work in OSL a long time without ever needing to use any of these commands. However, file commands are an excellent way to bring a real-world interaction to the entire Outpost messaging environment.

NOTE: Before proceeding, I recommend reading the **Command Reference** on these statements: `FindFile`, `NextFile`, `EXISTS`, `GetFileName`, `MoveFile`, and `Delete`.

Getting Started

Most of the file functions are self-explanatory. However, there is one command set that warrants some discussion: `FindFile` and `NextFile`. Here's an example:

```
SCRIPT
VAR FNAME as string
VAR NAMEONLY as string

BEGIN

‘ Set up the mask to look for any file that matches this pattern.
  FindFile(“c:\data\wx*.txt”)

‘ Get the first file if one is there.
  FNAME = NextFile(0)

‘ The best check is to see if the file exists
WHILE Exists(FNAME) = TRUE

‘ if the file exists, create a message and post it
    BBS=“K6FB-2”; FROM=“KN6PE”; TO=“ALLCTY”; MTYPE = “BULLETIN”

‘ Put the file name (not entire path) in the subject line as well
    NAMEONLY = GetFileName(FNAME)
    SUBJECT=“WX Bulletin: ” & NAMEONLY
    MESSAGE=READFILE(FNAME)
  CREATEMESSAGE

‘ We could either delete the file or move it.  For this example,
‘ we will move it so we don’t detect it again
  MOVEFILE(FNAME, “c:\data\sent”)

‘ Finally, get the next file name and repeat this all over again
  FNAME = NextFile(0)

  Print(“----------------------”)

ENDWHILE

END
```

The **`FindFile`** and **`NextFile`** statements are another pair of statements that work together. They allow you to search for one or more files that match a certain pattern (File Mask) of file names that may exist in a directory. For instance, suppose the following files are in the directory `c:\data`:

```
WX080604MONTEREY.TXT
WX080810PACIFICA.TXT
```

All the files begin with the 2 characters “WX”, are followed with the 6 digit date when it was created, and the region that it covers. Because you may not know when files will show up, you need some way to check for the file names so you can post them as bulletins to your

communications team. What you do know is that all file names start with "WX" and end with a ".TXT". Now you have the makings for defining a file mask.

For Opscripts to find these files, you set up the mask as follows: "`c:\data\wx*.txt`". This means:

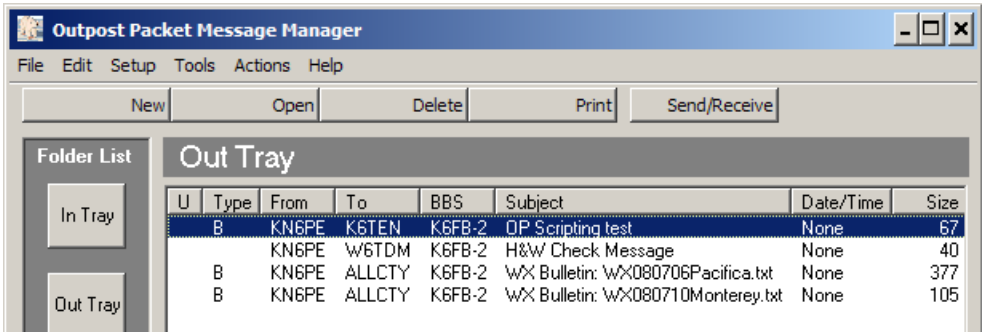
(i) You are looking at files in the `c:\data\` directory
(ii) Search for all files that start with the characters "WX".
(iii) and, match the ".txt" at the end.
(iv) The asterisk (*) means match everything else in between.

Because we are putting this all within a WHILE loop, the script will find each file name, assign it to FNAME, create a message, move the file to a "sent" subdirectory, and then get the next file name. This loop will repeat (2 times) until all file matches are processed.

With the 2 files in `c:\data`, here's what is produced when this script is run:

```
STARTING SCRIPT EXECUTION
Reading file c:\data\WX080706Pacifica.txt...
Initializing a new message
Saved new message ID=55
File "c:\data\WX080706Pacifica.txt" moved to "c:\data\sent"
----------------------
Reading file c:\data\WX080710Monterey.txt...
Initializing a new message
Saved new message ID=56
File "c:\data\WX080710Monterey.txt" moved to "c:\data\sent"
----------------------
END OF SCRIPT EXECUTION
```

A quick check of the Outpost Out Tray shows these 2 messages are ready to be sent. Note the File Name is included as part of the Subject.



The **`NextFile`** function returns a file with its full directory path if there are more to match. If there are no more files, then it returns a blank string. This then fails the `EXISTS` test in the `While` Statement.

The **`GetFileName`** function takes a fully qualified file name (path and all), and returns the file name only. This is useful when you want to append the file name to something without the path.

The **`MoveFile`** function moves the file from the `c:\data` directory to the `c:\data\sent` directory. See the command reference for more details.

Instead of moving the file, we could have used the **`Delete`**`(Fname)` function to delete the file instead if we didn't care to keep it around.

All of the above script code could be put within a bigger unconditional loop (`LOOP…ENDLOOP`) that would frequently check if new files show up. See Script **Example #3** for how this would look.

3.12 Tutorial #12 – Interacting with the outside world

There are a couple of ways that Opscripts interacts with the outside world. We have already experimented with some of them, such as the **`Print`** function. However, there are others that can be used in ways still yet to be discovered. They break down in the following categories:

Audio Notifications

OSL supports two types of sound producing statements:

- **`Beep.`** It does just as it says. This statement causes the PC to beep.
- **`Play(wav_file).`** This command plays the named wav file. If a sound card is not present, then the PC will Beep instead.

  You can do more than just playing "tada.wav" or some of the other .wav files that come with Windows. Using the PC's Sound Recorder, you could record yourself saying something like: "Hey! Change the frequency to 145.050, then press resume!" If you are operating in a noisy environment, turn the volume up!

External Program Control

OSL also allows you to run programs either synchronously or asynchronously.

- **`Run(program_name).`** This function causes the named program to run. Opscripts will not wait for it to complete, and will continue on with the rest of the script.
- **`RunW(program_name).`** Run with Wait. This function causes the named program to run. In this case, Opscripts will wait for the called program to exit before continuing with the next statement.

3.13 Tutorial #13 – Outpost-initiated scripts

You can automatically run a script from Outpost for the following 3 situations:

1. Run a script when starting Outpost. There could be tasks that can be wrapped into a script and processed prior to more extensive Outpost processing. This is set up in Outpost and triggered immediately on running Outpost.
2. Run a script when exiting Outpost. Similarly, there could be some cleanup activities or programs to run as Outpost is shutting down.
3. Run a script when Outpost initiates a Send/Receive session. There may be times when the standard Outpost Send/Receive process is not sufficient and a custom script needs to run in its place. Outpost now lets the user select how the Send/Receive process works, either run the current native Outpost Send/Receive process, or run a user-defined script.

This is set up in Outpost from the **Tools > Script Settings…** menu.

Getting Started
The following is an example of a Startup Script.

```
SCRIPT
‘ Description: Startup script in the EOC; sends a message to
‘              county EOC letting them know that we are on line.

BEGIN

  BBS=“K6FB-2”
  MYCALL=“KN6PE”
  TACCALL=“CUPEOC”
  FROM=“CUPEOC”
  TO=“XSCEOC”
  SUBJECT=“CUPERTINO EOC Radio Room is Staffed”
  MTYPE=“Private”
  MESSAGE=“This is an auto-generated message.  73, Jim”
  CREATEMESSAGE
  Sendreceive

‘ Finally, terminate Opscripts and wait for the next run request
  EXIT

END
```

The Opscripts command that is needed whenever initiating a script from Outpost is the **EXIT** command.

- **Exit.** This command causes the Opscripts Program to exit.

This command ensures that there is isn’t a left-over copy of Opscripts running prior to the next time Outpost attempts to run a script. Here are some considerations when using **EXIT**.

1. When Opscripts encounters an **EXIT** command, it does just that… immediately terminates the Opscripts program. It will not prompt to save work or give you any other warning.
2. When developing a script using the **EXIT** command, leave this command commented out while developing and debugging the script. It will save you the time and frustration of re-entering your script when an inadvertent **EXIT** is encountered.
3. Be sure your script is saved immediately after you uncomment this command.

4 Sample Scripts

The following are examples on how OSL can be used to automate different types of Outpost tasks.

4.1 Example 1 – Poll 3 different BBSs

One common request is for Outpost to check one BBS, then another. Today, the user needs to make the change manually in Outpost. This approach can be expanded to any number of BBSs provided they are on the same frequency. See the website for other examples and methods for doing the same thing.

```
SCRIPT
‘ ***************************************************************
‘ Description:  Loop on 3 BBSs.  This script will continuously loop
‘               on these 3 BBSs.  To exit the loop, press the STOP
‘               button on Run Monitor Form.
‘ Author:       Jim KN6PE
‘ Revision:     08/05/08     Original
‘               11/15/10     Updated
‘
‘ ***************************************************************

BEGIN
TNC = “GARAGE-TNC”           ‘ use this TNC for all runs

LOOP
   Print(CRLF & “Checking K6FB-2 BBS as KN6PE…”)
   MYCALL = “KN6PE”          ‘ check for my personal messages
   BBS = “K6FB-2”            ‘ check the Las Cumbres ARC PBBS
   RETRIEVE = “PB”           ‘retrieve Private and Bulletins
   SENDRECEIVE

   Print(crlf & “Checking K6FB-2 BBS as K6KP…”)
   MYCALL = “K6KP”           ‘ check for any messages to CARES
   BBS = “K6FB-2”            ‘ check the Las Cumbres ARC PBBS
   RETRIEVE = “P”            ‘retrieve Private messages only
   SENDRECEIVE

   Print(crlf & “Checking W6SJC-1 BBS as KN6PE…”)
   MYCALL = “KN6PE”          ‘ check for my personal messages
   BBS = “W6SJC-1”           ‘ check the San Jose RACES F6FBB BBS
   RETRIEVE = “PF”           ‘retrieve Private and Filtered msgs
   FILTER = “RACES:SCCNOR:SCCSOU”
   SENDRECEIVE

   PAUSE(600)                ‘ Pause 10 minutes…

ENDLOOP                      ‘ and, repeat

END
```

4.2 Example 2 – Periodically send a Health & Welfare message

The user has this instance of Outpost operating remotely, and wants to know that it is still running.

```
SCRIPT
‘ ************************************************************
‘ Description:  Post a H&W message to the BBS every 4 hours
‘ Author:       Jim KN6PE
‘ Revision:     08/05/08     Original
‘
‘ ************************************************************

BEGIN

BBS = “K6FB-2”
TNC = “GARAGE-TNC”
MYCALL = “KN6PE”
RETRIEVE = “PB”         ‘retrieve Private and Bulletins

LOOP

   Print(crlf & “Creating and sending a H&W message…”)
   FROM = “KN6PE”
   TO = “W6TDM”
   SUBJECT = “H&W Check Message”
   MESSAGE = “Allan, system is still running.  73, Jim”
   MTYPE = “Private”
   CREATEMESSAGE

   SENDRECEIVE
   PAUSE(60*60*12)     ‘ Pause for 12 hours (12*60*60=14400 seconds)

ENDLOOP

END
```

4.3 Example 3 – Detect and send a text file

This script allows a user who captures weather bulletins off of the internet to forward to the local packet community. He figured out a way to get the bulletin down to his PC (not shown here).

```
SCRIPT
' ************************************************************
' Description:  Detect and forward weather bulletins.  The WX file
'               is written to a directory on the PC with the
'               format WXyymmddhhmmss.TXT (example).  Loop
'               continuously on the check
' Author:       Jim KN6PE
' Revision:     08/05/08:  Original
' ************************************************************

Var FNAME as string         ' variable holding the file name and path
Var NAMEONLY as string      ' variable holding the file name only

BEGIN
' Define the BBS that we will use
BBS = "K6FB-2"
TNC = "GARAGE-TNC"
MYCALL = "KN6PE"
RETRIEVE = "PF"            ' retrieve Private and selective bulletins
FILTER = "WX"              ' for Filtered, only get WX messages

LOOP
' Check if one or more files matching this mask exist
  FindFile("c:\data\WX*.txt")         ' reload the file mask

' Get the first file.  Fname will contain a file name if one exists
  FNAME  = NextFile(0)

' Check if a file exists
  WHILE Exists(FNAME) = TRUE

' if the file exists, create a message and post it
      BBS="K6FB-2"; FROM="KN6PE"; TO="ALLCTY"; MTYPE = "BULLETIN"

' Put the file name (not entire path) in the subject line as well
      NAMEONLY = GetFileName(FNAME)
      SUBJECT="WX Bulletin: " & NAMEONLY
      MESSAGE=READFILE(FNAME)
    CREATEMESSAGE

' Could delete or move it; lets move it so we don't detect it again
    MOVEFILE(FNAME, "c:\data\sent")

' Finally, get the next file name and repeat this all over again
    FNAME = NextFile(0)

' Add a line separator to the output for readability
    Print("----------------------")

  ENDWHILE

  SENDRECEIVE             ' do this regardless if we found a file
  PAUSE(300)              ' Pause 5 minutes (5*60=300) between checks
ENDLOOP

END
```

4.4 Example 4 – Forwarding Outpost messages

This script lets the user look for a specific message received by Outpost and forward them to a different BBS.

```
SCRIPT
' ************************************************************
' Description:  Detect incoming Outpost messages and forward
'               to a different BBS.  For this script, check K6FB-2
'               for specific incoming messages addressed to KN6PE
'               with a specific subject, and forward it to my
'               email address via WINLINK.
' Author:       Jim KN6PE
' Revision:     08/17/08  Original
'
' ************************************************************

VAR MsgID AS NUMBER        ' holds the message ID for found msgs
VAR counter AS NUMBER      ' counts the number of matched msgs

BEGIN

LOOP
' Set to check K6FB-2 for incoming private messages to me
  counter = 0
  BBS = "K6FB-2"; TNC = "GARAGE-TNC"; MYCALL = "KN6PE"
  RETRIEVE = "P"
  SendReceive

' I only care about certain messages.  Check the Intray (1), the
' Subject Field (4), for a subject that starts with the characters
' "NOAA", then has anything after it "*"
  FindMessage(1, 4, "NOAA*")

' Get the first message ID (Outpost internal value) if one exists
  MsgID = NextMessage(0)

    WHILE MsgID > 0             ' One exists if greater than 0

' Yes, we have one! Create the message with this file content
      Print("Forwarding message with Subject=" & subject)

      counter = counter + 1
        BBS = "SANDIEGO"          ' must be defined in Outpost
        TNC = "SANDIEGO-TELNET"   ' must be defined in Outpost
        TO = "SMTP:kn6pe@arrl.net"

' For the message, add a line up front to the message text
        MESSAGE = "OSL-forwarded from K6FB-2" & CRLF & MESSAGE
      CREATEMESSAGE

      MoveMessage(MsgID, 4)    ' Once sent, move to Archive
      MsgID = NextMessage(0)   ' get the next match
    ENDWHILE

    IF counter > 0 then
        SENDRECEIVE              ' send via WINLINK
        Counter = 0              ' and reset the counter to 0
    ENDIF

    PAUSE(600)      ' Pause 10 minutes (10*60=600) between checks
ENDLOOP

END
```

5 Command Reference

5.1 Summary

General Functions and Statements

Beep	Statement	Plays a beep on the PC speaker
Begin	Statement	Required; Marks the beginning of the script
Clear	Statement	Clears the runtime monitor display
End	Statement	Required; Marks the end of the script
Exit	Statement	Causes Opscripts.exe to terminate when encountered
If… Then… Else	Statements	Conditional check
Loop… EndLoop	Statements	Unconditional loop
OnError	Statement	Determines how to proceed in the event an error occurs
Script	Statement	Required; Identifies this file as a script
SendOnly	Statement	Initiates an Outpost Send Only session
SendReceive	Statement	Initiates an Outpost Send/Receive session
Var	Statement	Defines a user variable
While.. EndWhile	Statements	Conditional loop
Now()	Functipon	Returns date and time based on user formatting
Pause()	Function	Causes the script to pause
Play()	Function	Plays a .wav file
Print()	Function	Prints a string of text to the Runtime Monitor window
Run()	Function	Run a program, does not wait for it to complete
Runw()	Function	Run a program, waits for it to complete

File Functions

Delete()	Function	Deletes a file
Exists()	Function	Tests if a file exists
FindFile()	Function	Sets up to find matches to a file mask
GetFileName()	Function	Returns the file name only from a full path file string
MoveFile()	Function	Moves a file to a different directory
NextFile()	Function	Gets the next file that matches a file mask
ReadFile()	Function	Read a file content
ValidFileName()	Function	Creates a valid file name from path and name elements
WriteFile()	Function	Writes text to a named file

String Functions

FindWord()	Function	Sets up to find matches to a comma-delimited string
NextWord()	Function	Gets the next word in a comma-delimited string
Len()	Function	Returns the length of a string

Message Statements and System Variables

CreateMessage	Statement	Creates an Outpost message based on parameters
FindMessage()	Function	Searches Outpost for a message
MoveMessage()	Function	Moves an Outpost message to a different folder
NextMessage()	Function	Gets the next Outpost message that matches the search
BBS	System Variable	Holds the BBS name
FROM	System Variable	Holds the FROM address

TO	System Variable	Holds the TO address
SUBJECT	System Variable	Holds the Subject of the message
MESSAGE	System Variable	Holds the body of the message
MTYPE	System Variable	Holds the message type
URGENT	System Variable	Holds the state of the outgoing message Urgent flag.
BBSMSGNO	System Variable	Holds the BBS message number for received messages
RECEIPTS	System Variable	Holds the Receipt flags for outgoing messages
LMI	System Variable	Holds the Local Msg ID for received messages
DATETIME	System Variable	Holds the Date Time string for received messages

Send/Receive Statements and System Variables

SendOnly	Statement	Initiates an Outpost Send Only Session
SendReceive	Statement	Initiates an Outpost Send/Receive Session
Expire()	Function	Sets up a bulletin for deletion from the BBS
BBS	System Variable	Holds the BBS name
TNC	System Variable	Holds the TNC name
MYCALL	System Variable	Holds the Station Identifier (Call Sign)
TACCALL	System Variable	Holds the Tactical Call
RETRIEVE	System Variable	Holds what message types are to be retrieved
FILTER	System Variable	Holds the categories for a Filter Retrieve
SRNOTE	System Variable	Holds the results of a Send/Receive Session

Other System Variables

TRUE	System Variable	Value against which conditions can be checked
FALSE	System Variable	Value against which conditions can be checked
ON	System Variable	Value that can be used to set items
OFF	System Variable	Value that can be used to set items
CRLF	System Variable	Value that causes a carriage return/Line feed on output

5.2 Special Characters

‘ (single quote)	***Description*** The single quote starts the beginning of a comment. Everything after the single quote is part of the comment up until the end of the line. ***Syntax*** ‘ <comment> ***Example*** ‘Comments can begin as the first character x = x+ 1 ‘ or after a statement ***Notes*** 1. All comments are preceded with a single quotation mark.
+ - / * %	***Description*** Arithmetic operators: OSL supports the standard arithmetic operations. All precedence rules apply. ***Syntax*** <var> = [var | number] <operand> [var | number] ***Example*** x = x + 1 x = x * (5 - y) / 4 ***Notes***

	1. When an expression has a mix of operators, the precedence of execution is multiple and division first, then addition and subtraction. 2. Expressions in parenthesis are always executed first. 3. Space are optional when formatting an expression.

; (semicolon)	***Description***
	Command line continuation. Placing a semicolon at the end of a line allows you to add another command to the same line.
	
	Example
	`x=x+1; y = y – 1`
	`BBS = "K6FB-2"; TNC = "GARAGE-TNC"`
	
	Notes
	1. Care should be taken with this feature since it may contribute to readability problems and debugging your script.

< > =	***Description***
	Relationship operators. These are operators are used as part of conditional tests made with `WHILE…ENDWHILE` and `IF…THEN… ELSE` statements.
	
	Example
	`#1 IF X > 5 THEN`
	`#2 WHILE Y <= 12`
	
	Notes
	1. The following relationship operators are defined:
	`<` less than
	`<=` less than or equal to
	`>` greater than
	`>=` greater than or equal to
	`=` equal to

5.3 Command Reference

Assignments	***Description***
	Assignments are statements that assign a value, variable, result of a function call, or arithmetic operation to another variable.
	
	Syntax
	`<var> = [ number | string | <var> | expression | function ]`
	
	Examples
	`#1. Temp = "Weather.txt"`
	`#2. Result = X / (Y+5)`
	`#3. Fname = NEXTFILE(0)`
	
	Notes
	1. The rules of operational precedence apply to all arithmetic calculations.
	2. There is limited type checking; use caution when mixing strings and numbers in an arithmetic expression.

BBS	***Type***
	System Variable
	
	Description
	Holds the Friendly Name of the BBS. This variable is used by the `CREATEMESSAGE` and `SENDRECEIVE` statements
	
	Syntax
	`BBS = <bbs_name>` Default = blank
	
	Example
	`BBS = "LCARC Path1" ' LCARC's K6FB-2 BBS via AA6WK-7`
	`Print("Checking BBS " & BBS)`
	`SendReceive`
	
	Notes

	1. The value that you assign to the BBS variable is the Friendly name of a BBS that is already defined in Outpost. Connect Names can also be used, but in the event of multiple BBS Friendly Name entries with the same Connect Name, the 1st BBS entry will be used (#817, #858, 16-May-10). 2. If this BBS is not set up in Outpost, at the time the Send/Receive session is attempted, Outpost will generate the message: "Either the Station ID, BBS, or TNC is not selected…". This message may not pop to the front; you may need to minimize the Script window to see it.

BBSMSGNO	***Type***
	System Variable
	
	Description
	Holds the BBS message number that was associated with the message retrieved from the BBS.
	
	Example
	`Expire(BBSMSGNO)`
	`SendReceive`
	
	Notes
	1. This field is for display purposes only after retrieving a message.

Beep	***Description***
	Causes the PC to Beep
	
	Syntax
	`Beep`
	
	Example
	`IF x > 5 THEN`
	`  BEEP`
	`ENDIF`
	
	Notes
	1. Also, see the PLAY statement as an alternate audible annunciation option.

Begin	***Description***
	Defines the beginning of the OSL Script statements.
	
	Syntax
	`BEGIN`
	
	Example
	`SCRIPT`
	`VAR x AS NUMBER`
	`BEGIN`
	`  x = 5`
	`  Print("The value of ''x'' is " & x)`
	`END`
	
	Notes
	1. This statement acts as a boundary between all variable declarations and the first script statement.
	2. After pressing NEW, this statement is 1 of 3 statements that are automatically inserted in the new script editing window.
	3. Also, see the SCRIPT, END statements.

Clear	***Description***
	Clears the runtime monitor display. Used primarily for display formatting
	
	Syntax
	`Clear`
	
	Example
	`Loop`
	`  :`
	`  Print("polling BBS " & BBS)`
	`  SendReceive`
	`  Pause(5)`

	`  Clear`
	`Endloop`
	
	Notes

CreateMessage	***Description***
	Creates a message based on the settings of the message-reserved variables, and writes the message to the Outpost message database.
	
	Syntax
	`CreateMessage`
	
	Example
	`  BBS = “K6FB-2”`
	`  FROM= “KN6PE”`
	`  TO= “K6TEN”`
	`  SUBJECT= “Repeater Update”`
	`  MESSAGE = ReadFile(RepeaterMessage)`
	`  MTYPE = “PRIVATE”`
	`CreateMessage`
	
	Notes
	1. All message reserved variables must be set prior to executing this statement.
	2. Message-reserved variables are: `BBS, FROM, TO, SUBJECT, MESSAGE, MTYPE`
	3. A valid message is written to the Outpost message database and is set for the next send/receive session.
	4. Also, see: `BBS, TNC, MYCALL, TACCALL, RETRIEVE, FILTER`

CRLF	***Type*** System Predefined Variable ***Description*** Contains the 2 characters for Carriage Return and Line Feed. It is used to insert a Carriage Return / Line Feed (same as pressing the Enter Key) in a string so that a single string can display multiple lines. ***Example*** `Msg = “Hi Cap,” & CRLF & “Hope all is well.” & CRLF & “73, Jim”` ***Notes*** 1. The `CRLF` is a variable and not part of the string that you define. It is appended to other portions of the string with an “&”.
DATETIME	***Type*** System Variable ***Description*** Holds the retrieved message Date time as listed on the BBS and the Outpost message listing. ***Example*** `Print(DATETIME)` ***Notes*** 1. This field is for display purposes after retrieving a message. There is no effect to set this field.

Delete()	***Description***
	Deletes the named file.
	
	Syntax
	`DELETE( file_name )`
	
	Return
	`none`
	
	Example
	`#1. Delete(“weather-report.txt”)`
	`#2. FName = “weather-report.txt”`

	`Delete(FName)` ***Notes*** 1. In the event the file does not exist, is open, or is write-protected, the file will not be deleted and an error message will be displayed on the Runtime Monitor.

End	***Description***
	The last statement in the script, Required.
	
	Syntax
	`END`
	
	Example
	`SCRIPT`
	`BEGIN`
	`Print("Hello World!")`
	`END`
	
	Notes
	1. This statement must be the last statement in the script.
	2. After pressing "NEW", this statement is 1 of 3 statements that are automatically inserted in the new script editing window.
	3. Also, see the `SCRIPT`, `BEGIN` statements.

Exists()	***Description***
	Tests whether the named file exists.
	
	Syntax
	`EXISTS( file_name )`
	
	Return
	`Number: 0 - FALSE`
	`1 – TRUE`
	
	Example
	`#1. If EXISTS("weather-report.txt") = TRUE then`
	
	`#2. FName = "weather-report.txt"`
	`Result = Exists(FName)`
	`IF Result = FALSE THEN`
	`:`
	
	Notes
	1. The function will return either a 0 or 1 depending on the outcome.
	2. When using with the `IF` statement (1st example), use the System Variables `TRUE` or `FALSE` for the test.

Exit	***Description***
	Terminate Opscripts.exe when encountered (#748)
	
	Syntax
	`EXIT`
	
	Example
	`SCRIPT`
	`BEGIN`
	`Print("Hello World!")`
	`EXIT ' terminate Opscripts`
	`END`
	
	Notes
	1. This command is used whenever you want to run a script from Outpost and terminate scripting when done.
	2. Save your work before running with this command. It will exit without prompting to save your work.

Expire()	***Description*** Delete a bulletin message that belongs to you. ***Syntax***

	`EXPIRE( 0 | Bbs_Msg_ID )`
	
	Example
	`FindMessage(1, 4, "*WX ADVISORY*")` ' 1=Intray, 4=Subj Field of Bull name
	`MsgID = NextMessage(0)`
	
	`WHILE MsgID > 0` ' One exists if greater than 0
	`IF FROM = "KN6PE" then` ' is it from me? If so, its my Bulletin
	`Print("Deleting " & subject)`
	`EXPIRE(0)` ' set it up to delete next S/R cycle
	`movemessage(MsgID,4)` ' move the message to Archive Folder
	`ENDIF`
	
	`MsgID = NextMessage(0)` ' get the next match, if any
	`ENDWHILE`
	
	Notes
	1. Use a "0" with the Expire command to use the BBS message number associated with the last message loaded by the NextMessage function.
	2. In the above example, suppose you periodically post a bulletin message that contains the subject line phrase "WX ADVISORY". This lets you find the message again so we can delete it when a new update comes along.
	3. The user needs to test to determine if the message being retrieved is in fact a bulletin that (i) exists, and (ii) the user originally posted. Only the bulletin owner can delete a posted bulletin.

FILTER	***Description***
	System Predefined Variable. Holds the string of concatenated filter values that will be used during a Filter Retrieval.
	
	Syntax
	`FILTER = "filter1:filter2:...:filtern"`
	Default = blank
	
	Example
	#1. RETRIEVE = "PF"
	`FILTER = "QST"`
	#2. `FILTER = "LINUX:KEPS:SOCTY"`
	
	Notes
	1. `FILTER` must be set if the `"F"` Filter `Retrieve` option is set.
	2. All filters must be separated with colons `":"`.
	3. The entire `Filter` assignment enclosed in quotations.
	4. Any number of filters can be assigned to the `FILTER` variable.

FindFile()	***Description***
	Searches for and collects all file names that match a particular string pattern.
	
	Syntax
	`FINDFILE( pattern )`
	`pattern`: some or all of the file name to match; use "*" to fill. For instance
	`c:\data\WX*.txt` : finds files that start with WX and end with `.TXT`
	`*.*` : finds all files in the current directory
	
	Example
	`SCRIPT`
	`VAR NameOnly as string`
	`VAR FullName as string`
	`VAR ctr as number`
	
	`BEGIN`
	`ctr = 0`
	`FINDFILE(``"c:\data\`**`*.txt"`**`)`
	`FullName` **`= NextFile(0)`**
	
	`While Exists(FullName) = TRUE`
	`NameOnly = GetFileName(FullName)`

```
    Print(FullName & " -- " & NameOnly)
    FullName = NextFile(0)
    ctr = ctr + 1
  ENDWHILE

Print("Files Found: " & ctr)
END
```

Notes

1. This function initializes the File Mask function allowing the **NextFile** function to retrieve each file that matches the mask.
2. Each file returned will contain the equivalent amount of the path as was set up. For instance:

If `FindFile` contains…	then `NextFiles` will include
`scripts\test*.txt`	`script\full_file_name`
`c:\data\*.*`	`c:\data\full_file_name`

3. On entering another file mask, the retrieval is reset.
4. Use the "*" to match any character(s) between characters
5. See the `NextFile` Function

FindMessage()

Description

Searches all Outpost messages that match a particular string pattern or Regex.

Syntax

`FINDMESSAGE( <folder>, <field>, "<pattern>" )`

`folder`: A number corresponding to an Outpost folder to search.
Valid numbers are:

1. InTray
2. Out Tray
3. Sent Folder
4. Archive Folder
5. Draft Folder
6. Deleted Folder
11. Special Folder #1
12. Special Folder #2
13. Special Folder #3
14. Special Folder #4
15. Special Folder #5

`field`: A number corresponding to an Outpost Message field to search.
Valid numbers are:

1. BBS
2. FROM
3. TO
4. SUBJECT
5. MESSAGE

`pattern`: The string pattern or Regex to match. Wildcard use (KN6*) is allowed.

Characters in Pattern	Matches in String
?	Any single character
*	Zero or more characters
[charlist]	Any single character in charlist
[!charlist]	Any single character not in charlist

For Regex: the pattern format is: `"(?#REG)<pattern>"`

Return

none

Example

```
SCRIPT
VAR MsgID as number
VAR ctr as number
```

```
BEGIN
  ctr = 0
  FindMessage(1,4,"NOAA*")
  MsgID = NextMessage(0)

  while msgid > 0
    Print("Found Msg: " & SUBJECT)
    MsgID = NextMessage(0)
    ctr = ctr + 1
  endwhile

  Print("Messages found: " & ctr)
END
```

Notes

1. This function initializes the Message Mask function allowing the **NextMessage** function to retrieve each message that matches the mask. For instance:

If `FindMessage` contains…	then `NextMessage` will include
`CUP*`	`CUPertino, CUP043…`
`*`	`<anything>`

2. On entering another message mask, the retrieval is reset.
3. See the `NextMessage` Function
4. The pattern match is not case sensitive, meaning that a mask of "repeater" will match to a string "REPEATER".
5. Pattern Match Examples:

Pattern	Target String	Result
"F"	"F"	Matches
"f"	"F"	Matches
"FFF"	"F"	No match
"a*a"	"aBBBa"	Matches
"[A-Z]"	"F"	Matches
"[!A-Z]"	"F"	No Match
"a#a"	"a2a"	Matches
"a[L-P]#[!c-e]"	"aM5b"	Matches
"B?T*"	"BAT123Kng"	Matches
"B?T*"	"CAT123Kng"	No Match

6. To match the special characters left bracket ([), question mark (?), number sign (#), and asterisk (*), enclose them in brackets.
7. The right bracket (]) cannot be used within a group to match itself, but it can be used outside a group as an individual character.
8. By using a hyphen (–) to separate the lower and upper bounds of the range, charlist can specify a range of characters.
9. To specify multiple ranges for the same character position, put them within the same brackets without delimiters. Example: [A–CX–Z] matches letters A thru C, and X thru Z.
10. A hyphen (–) can appear either at the beginning (after an exclamation point, if any) or at the end of charlist to match itself.

FindWord()

Description

Sets up to return the individual words found within a comma-delimited string.

Syntax

`FINDWORD( <string> )`

`<string>`: string contains individual words that need to be retrieved

Example

```
SCRIPT
VAR ListOfBBS as string
VAR SingleBBS as string
VAR ctr as number

BEGIN
  ctr = 0
  ListOfBBS = "K6FB-1, W6XSC-1, K6TEN, SANDIEGO"
  FINDWORD(ListofBBS)
  SingleBBS = NextWord(0)

  While LEN(SingleBBS) > 0
    Print("Next BBS name is " & SingleBBS)
```

	SingleBBS = **NextWord(0)** ctr = ctr + 1 ENDWHILE Print("Number of BBSs Found: " & ctr) END ***Notes*** 1. This function initializes the String Search function allowing the **NextWord** function to retrieve each word from the array of comma-delimited words. 2. On entering another Word search, the retrieval is reset. 3. The list of strings must be in a set a quotes. Individual words must be separated by commas. 4. See the NextWord Function
FROM	***Description*** System Predefined Variable. Holds the call sign or tactical calls for the message From field. ***Syntax*** FROM = "<call_sign>" Default = blank ***Example*** From = "KN6PE" ***Notes*** 1. The FROM assignment is enclosed in quotations.
GetFileName()	***Description*** Returns the file name portion of a string that includes the file name and path ***Syntax*** <var> = GetFileName(<full_name>) ***Example*** SCRIPT Var FullName as String Var FileName as String BEGIN FullName = "c:\data\Weather.txt" FileName = GetFileName(FullName) returns "Weather.txt" Print(FullName & " " & FileName) END ***Notes*** 1. This command is useful if you intend to create messages with the subject name embedded in it

If… Then [Else] Endif	***Description***
	Conditionally executes a block of statements dependent on the state of the condition.
	
	Syntax
	IF <condition> THEN
	<statements>
	[ELSE
	<statements>]
	ENDIF
	
	Example
	#1. If x > 5 THEN
	x = x + 1
	ENDIF
	
	#2. If x > 5 THEN
	Print(x)
	ELSE
	x = x + 1
	ENDIF

	Notes 1. The ELSE statement is optional and not required 2. See Also: WHILE, LOOP
LEN()	***Description*** Returns the length of a string (number of characters) ***Syntax*** <result> = LEN(<string>) result : integer, indicates the number of characters in the string string : the string to be tested ***Example 1*** **SingleBBS =** "K6FB-1" **WordLen = LEN(SingleBBS)** ***Example 2*** **SingleBBS =** "K6FB-1" While LEN(SingleBBS) > 0 : ***Notes***
LMI	***Description*** System Predefined Variable. Holds the Local Message ID (LMI) if enabled in Outpost for incoming messages. ***Syntax*** LMI = "[blank | <LMI value>" Default = depends on Outpost setting ***Example*** No example ***Notes*** 1. This field is for display purposes after retrieving a message. There is no effect to set this field. 2. See the Outpost Users Guide for a description of LMI.
Loop… EndLoop	***Description*** Continuously loops on a block of statements ***Syntax*** LOOP <statements> ENDLOOP ***Example*** LOOP SendReceive Pause(300) ENDLOOP ***Notes*** 1. The only way to exit this loop is to press the "STOP" button on the Runtime control form. 2. See Also: IF, WHILE
MESSAGE	***Description*** System Predefined Variable. Holds the body of the message. ***Syntax*** MESSAGE = "<message text>" Default = blank ***Example*** #1. Message= "Hi Vince, All is still OK here. 73, Jim" #2. Message= ReadFile("Message.txt")

	Notes 1. Use a string assigned to MESSAGE for short messages. 2. Use the `ReadFile()` function to read in the contents of a file to set the message. See Script example #3.
MoveFile()	***Description*** Moves the named file from one location to another. ***Syntax*** `MOVEFILE( <path\file_name>, <dest_path> )` `path\file_name` : The current path and file name of the file to be moved `dest_path` : The Path only of where the file will be moved. Do not include any trailing back slashes ***Return*** none ***Example*** #1. `MoveFile( "c:\data\wx.txt", "c:\data\sent" )` #2. `MoveFile( InName, "c:\data\sent" )` ***Notes*** 1. If the source file is not found, a runtime error will occur and the script will stop. It is recommended that you check for the presence of the file with the `Exists()` function prior to moving or reading a file.
MoveMessage()	***Description*** Moves a message from one Outpost folder to another. ***Syntax*** `MOVEMESSAGE( <msg_id>, <folder_no> )` `Msg_id`: Outpost message pointer. Usually returned by the NextMessage statement `folder_no`: is defined as: 1. InTray 2. Out Tray 3. Sent Folder 4. Archive Folder 5. Draft Folder 6. Deleted Folder 11. Special Folder #1 12. Special Folder #2 13. Special Folder #3 14. Special Folder #4 15. Special Folder #5 ***Return*** none ***Example*** #1. `MoveMessage(MsgID, 4)` message is moved to the Outpost archive folder #2. `MoveMessage(MsgID, 6)` message is moved to the Outpost deleted folder ***Notes*** 1. The Message ID is an internal Outpost identified not typically used in the normal operation from the Outpost forms. From an OSL perspective, the Message ID typically comes from the `NextMessage` function. 2. Any folder value other than those listed above will cause and error and the script to stop.
MTYPE	***Description*** System Predefined Variable. Holds the message type for a message being created. ***Syntax*** `MTYPE= "PRIVATE" | "NTS" | "BULLETIN"` Default = blank ***Example***

	#1. MTYPE = "Private" #2. MTYPE = "NTS" ***Notes*** 1. Only one message type can be set for each message. If more or set, the last Message Type set will the one applied the next time the CreateMessage statement is executed. 2. If not provided, MTYPE defaults to "PRIVATE"
MYCALL	***Description*** System Predefined Variable. Holds the value of the Call Sign that is used to initialize the interface. This variable is used by the SendReceive statement. ***Syntax*** MYCALL = <call_sign> Default = blank ***Example*** MYCALL = "KN6PE" ***Notes*** 1. If left blank, then Outpost will use the currently defined Call Sign as defined from Outpost's **Setup > Identification** form (#758).

NextFile()	***Description***
	Retrieves the next file name that was previously collected by the FindFile function
	
	Syntax
	<Var_name> = NEXTFILE(0)
	
	Return
	String: Next file name (only) that matches the pattern
	If non-blank, valid file name
	If blank (null string), no file found, or reached the end of the list
	
	Example
	SCRIPT
	VAR NameOnly as string
	VAR FullName as string
	VAR ctr as number
	
	BEGIN
	ctr = 0
	FINDFILE("c:\data***.txt")**
	FullName **= NextFile(0)**
	
	While Exists(FullName) = TRUE
	NameOnly = GetFileName(FullName)
	
	Print(FullName & " -- " & NameOnly)
	FullName **= NextFile(0)**
	ctr = ctr + 1
	ENDWHILE
	
	Print("Files Found: " & ctr)
	END
	
	Notes
	1. This function retrieves the next file previously initialized by the FindFile function. The function returns the file name with whatever path was set up as the FindFile() parameter.
	2. The parameter "0" is required. This is for future use.
	3. Each time this function is called, the next file that matches the mask is returned.
	4. When there are no other matches, a blank string is returned. Use the EXISTS() Function to test whether a valid file name was returned.

NextMessage()	***Description*** Retrieves the next message ID that was previously collected by the FindMessage function. ***Syntax*** <Var_name> = NEXTMESSAGE(0)

	Return
	`Integer:` next file that matches the pattern
	If > 0: a valid Outpost message ID
	If = 0: no message found, or reached the end of the list
	
	Example
	`SCRIPT`
	`VAR MsgID as number`
	`VAR ctr as number`
	
	`BEGIN`
	`  ctr = 0`
	`  FindMessage(1,4,"NOAA*")`
	`  MsgID = NextMessage(0)`
	
	`  while MsgID > 0`
	`  Print("Found Msg: " & SUBJECT)  ' only print the subjects`
	`  MsgID = NextMessage(0)`
	`  ctr = ctr + 1`
	`  endwhile`
	
	`  Print("Messages found: " & ctr)`
	`END`
	
	Notes
	1. This function retrieves messages based on the selection criteria set up by the `FindMessage()` function.
	2. The Parameter "0" is required. This is for future use.
	3. Each time this function is called, the next message that matches the mask is returned.
	4. When there are no other matches, a value of 0 is returned. Use an `IF… Then` to test whether there is a valid message returned.

NextWord()	***Description***
	Retrieves either the sequentially next word or a specific word that was previously collected by the `FindWord` function
	
	Syntax
	`<Var_name> = NEXTWORD( <index> )`
	`index:` 0 (zero), returns the next word from the list
	1 .. n, returns the indexed word from the list
	
	Return
	`String:` Next word name that was set up
	If non-blank, valid word name
	If blank (null string), no word found, or reached the end of the list
	
	Example 1
	`SCRIPT`
	`VAR SingleBBS as string`
	`VAR ctr as number`
	
	`BEGIN`
	`  ctr = 0`
	`  FINDWORD(“K6FB-1, W6XSC-1, K6TEN, SANDIEGO”)`
	`  SingleBBS = NextWord(0)`
	
	`  While LEN(SingleBBS) > 0`
	`  Print(“Next BBS name is ” & SingleBBS)`
	`  SingleBBS = NextWord(0)`
	`  ctr = ctr + 1`
	`  ENDWHILE`
	
	`Print(“Number of BBSs Found: ” & ctr)`
	`END`
	
	Example 2
	`SCRIPT`
	`VAR SingleBBS as string`
	`VAR ctr as number`
	`BEGIN`

```
  ctr = 4
  FINDWORD("K6FB-1, W6XSC-1, K6TEN, SANDIEGO")
  SingleBBS = NextWord(ctr)

  While ctr > 0
    Print("Next BBS name is " & SingleBBS)
    ctr = ctr - 1
    SingleBBS = NextWord(ctr)
  ENDWHILE
END
```

Notes

1. This function retrieves the next word previously initialized by the `FindWord` function.
2. If the parameter is 0 (zero), then the next sequential word is returned.
3. If the parameter is > 0, then the word that is indexed by the parameter is returned.
4. A parameter is less than 0 or greater than the count of the number of words will returned a blank string.
5. For sequential (0) calls, each time this function is called, the next word is returned. The original string is not affected.
6. When there are no other matches, a blank string is returned. Use the `LEN( )` Function to test whether a string with any length was returned.

Now()

Description

Returns the date and/or time in a format specified by "user."

Syntax

`NOW( "<blank>" | "<format>" )`

Where <format> is:

Date options

Symbol	*Range*
d	1-31 (Day of month, with no leading zero)
dd	01-31 (Day of month, with a leading zero)
w	1-7 (Day of week, starting with Sunday = 1)
ww	1-53 (Week of year, with no leading zero; Week 1 starts on Jan 1)
m	1-12 (Month of year, with no leading zero, January = 1)
mm	01-12 (Month of year, with a leading zero, January = 01)
mmm	Displays 3-character abbreviated month names
mmmm	Displays full month names
y	1-366 (Day of year) This essentially is the Julian day, a continuous count of days since the beginning of the year.
yy	00-99 (Last two digits of year)
yyyy	100-9999 (Three- or Four-digit year)

Time options

Symbol	*Range*
h	0-23 (1-12 with "AM" or "PM" appended) (Hour of day, with no leading zero)
hh	00-23 (01-12 with "AM" or "PM" appended) (Hour of day, with a leading zero)
n	0-59 (Minute of hour, with no leading zero)
nn	00-59 (Minute of hour, with a leading zero)
m	0-59 (Minute of hour, with no leading zero). Only if preceded by h or hh
mm	00-59 (Minute of hour, with a leading zero). Only if preceded by h or hh
s	0-59 (Second of minute, with no leading zero)
ss	00-59 (Second of minute, with a leading zero)

Example

#1. `Now("")`	12/31/2021 5:24:58 PM
#2. `Now("m/d/yy")`	12/31/21
#3. `Now("mm/dd/yyyy")`	12/31/2021

	#4. Now("dd-mmm-yyyy") 31-Dec-2021 #5. Now("dd-mmmm-yyyy, hh:mm") 31-December-2021, 17:25 #6. Now("dd-mmm-yyyy, hh:mm AM/PM") 31-Dec-2021, 5:25 PM ***Notes*** 1. A parameter must always be entered with this function.
ON, OFF	***Description*** System Predefined Variable. CONSTANTS ***Notes*** 1. Can be used as a setting and for checking. ON = 1, OFF = 0

OnError	***Description***
	Sets how Opscripts will handle specific types of errors
	
	Syntax
	`ONERROR [ STOP | PAUSE | CONTINUE ]` Default = STOP
	
	Example
	`ONERROR STOP`
	`:`
	`ONERROR CONTINUE ' don't worry on an error`
	`DELETE(fname)`
	
	Notes
	1. Setting a STOP condition will cause the script to report the error on the Runtime form, and stop execution of the script.
	2. Setting a PAUSE condition will pop up a box telling the user to either press **STOP** to stop processing the script, or **RESUME** to continue
	3. Setting a CONTINUE condition will indicate on the Runtime form that an error occurred, and we are continuing anyway.
	4. ONERROR is used to handle the following situations:
	▪ Divide by zero
	▪ RUN(), RUNW(): Running a program, and the program is not found
	▪ DELETE(): deleting a file, but it does not get deleted (could be read-only, or opened to another program)
	▪ MOVEFILE(): Moving a file, but it the destination directory does not exist
	▪ MOVEFILE(): Moving a file, but it the source file does not exist
	▪ READFILE(): Reading a file, but it the file does not exist
	▪ WRITEFILE(): Creating a file, but it did not happen (could be read-only, or opened to another program)
	▪ FINDMESSAGE(): the Folder number is not between 1 and 6 (In tray thru Deleted folder). If the CONTINUE option is set, the Folder value is overridden to a value of "1" (In Tray), and processing continues.
	▪ FINDMESSAGE(): the Field number is not between 1 and 5 (BBS thru MESSAGE) If the CONTINUE option is set, the Field value is overridden to a value of "1" (BBS), and processing continues.
	5. Once an ONERROR condition is set, all errors after that point will be processed with that setting that until a different ONERROR condition is set.

Pause()	***Description*** Causes the script to pause. ***Syntax*** `PAUSE( seconds )` ***Example*** #1. `pause(60)` Pauses for 60 seconds #2. `pvalue = 60` `Pause(pvalue)` Pauses for 60 seconds #3. `pause(0)` Script stops, waits for user interaction ***Notes*** 2. Any value greater than zero will cause the script to pause for the number of seconds indicated. Once this statement is called, the script pauses and the time remaining will count down and be displayed in the lower right portion of the status bar.

	3. A value of "0" will cause the script to pause, and requires the user to press the **Resume** button on the Runtime Monitor window. This may be useful when there is something that the user needs to do prior to letting the script proceed.
Play()	***Description*** Causes the script to play the named .wav file. ***Syntax*** `PLAY( wav_file_name )` ***Example*** #1. `Play("tada.wav")` #2. `WavName = "tada.wav"` `Play(WavName)` same, with string variable ***Notes*** 4. The file must be locatable either by a fully qualified path or by the system path statement. 5. In the event the file is not found or there is no sound card on your PC, the PC will sound a"beep."

Print()	***Description***
	Prints a string of text to the Runtime Monitor window.
	
	Syntax
	`PRINT( <text_string> )`
	
	Example
	#1. `Print(15)` prints the number 15
	
	#2. `x = 15` set "x" to 15
	`Print(x)` print "x"; same result as above
	
	#3. `Print("Starting Process")` print a string
	
	#4. `x = x + 1` use "x" as a counter
	`Print("Pass #" & x)` print a string and variable
	
	#5. `FName = "Weather.txt"` assign a file name to Fname
	`Print("The file is " & FName)`
	
	Notes
	1. Print will output a single or concatenated string to the runtime monitor window.
	2. Multiple string components can be added and separated by an ampersand "&" sign.
	3. Content can be a mix of explicit string values and variables.

ReadFile()	***Description***
	Reads the content of the named file and assigns its contents to a string variable.
	
	Syntax
	`<Var_name> = READFILE( file_name )`
	
	Return
	`String: file contents`
	
	Example
	#1. `x = ReadFile("c:\data\wx.txt")`
	#2. `MESSAGE = ReadFile(Fname)`
	
	Notes
	1. In the event the file does not exist, or the path is wrong, a "file not found" message is displayed, and the script continues to run.

RECEIPTS	***Description*** System Predefined Variable. Holds the settings for overriding the Receipt Requests for this message.

	Syntax `RECEIPTS = "[ <blank> | R [ D]] "` Default = blank ***Example*** `RECEIPTS = "RD"` ' Request both a Delivery and Read Receipt ***Notes*** 1. The RECEIPTS assignment is enclosed in quotations.
RETRIEVE	***Description*** System Predefined Variable. Holds the string representation of the types of messages to be retrieved. This variable is used by the SENDRECEIVE statement. ***Syntax*** `RETRIEVE = <"P" "N" "B" "F">` Default = "P" ***Example*** #1. `RETRIEVE = "P"` retrieve only Private messages #2. `RETRIEVE = "PNB"` retrieve all message types #3. `RETRIEVE = "PF"` requires Filters to be set ***Notes*** 1. The coding for `RETRIEVE` is as follows: P = Private messages N = NTS messages B = Bulletins F = Filtered 2. If the "F" Filter and "B" Bulletin options are both set, then only the "F" Filter option will be used and the "B" will be ignored. 3. If the "F" Filter option is set, then the Filter string must also be set. If Filter string is not set, then the "F" Filter option is ignored. 4. `RETRIEVE` must be set prior to the next `SendReceive` statement.
Run()	***Description*** Causes the script to run a program, and does not wait for the program to complete before continuing with the script. ***Syntax*** `RUN( exe_file_name )` ***Example*** #1. `Run("notepad.exe")` #2. `Run(PName)` ***Notes*** 1. The executable file must be locatable either by a fully qualified path or by the system path statement. 2. In the event the program does not exist, a "program not found" message is displayed, and the script continues to run.
Runw()	***Description*** Causes the script to run a program, and will wait for the program to complete before proceeding with the rest of the script. ***Syntax*** `RUNW( exe_file_name )` ***Return*** none ***Example*** #1. `Runw("notepad.exe")` #2. `Runw(PName)` ***Notes*** 1. The executable file must be locatable either by a fully qualified path or by the system path statement.

	2. In the event the program does not exist, a "program not found" message is displayed, and the script continues to run.

Script	***Description***
	The first OSL statement that appears in the file.
	
	Syntax
	`SCRIPT`
	`BEGIN`
	`  Print("Hello World!")`
	`END`
	
	Notes
	1. This must be the first OSL command in the script file.
	2. After pressing **NEW**, this is 1 of 3 statements that are automatically inserted in the new script editing window.
	3. Also, see: `BEGIN`, `END`

SRNOTE	***Description***
	System Predefined Variable. Holds any Send/Receive Notification message that may occur from the last Send/Receive Session
	
	Syntax
	`SRNOTE = "[ <blank> | <Notification string>]"` Default = blank
	
	Example
	`IF Len(SRNOTE) > 0 then`
	`  Print("Send/Receive problems, message was " & SRNOTE)`
	`ELSE`
	`  Print("Last Send/Receive session was successful!")`
	`ENDIF`
	
	Notes
	1. This field is for display purposes after retrieving a message. There is no effect to set this field.

SendOnly	***Description***
	Initiates an Outpost send only session based on the settings of the system variables. Messages in the out tray will be sent. No check for incoming messages is made.
	
	Syntax
	`SENDONLY`
	
	Example
	`  FROM = "KN6PE"`
	`  TO = "K6KP"`
	`  SUBJECT = "Will miss tonight's net"`
	`  MESSAGE = "Stuck in traffic; start the net without me" & CRLF & "73, Jim o  KN6PE"`
	`  MTYPE = "PRIVATE"`
	`  CREATEMESSAGE`
	`  SENDONLY`
	
	Notes
	1. All session-specific variables must be set prior to executing this statement.
	2. Related System variables used by the `SendOnly` statement are: `BBS, TNC, MYCALL, TACCALL`
	3. Opscripts does not perform any error checking on the existence of the BBS and TNC names entered on these variables. On a Send Only error, Outpost will report the problem, not Opscripts.
	4. Outpost must be running for this statement to work. An error will occur if Outpost is not running.

SendReceive	***Description*** Initiates an Outpost send/receive session based on the settings of the system variables. ***Syntax*** `SENDRECEIVE` ***Example***

	MYCALL = "KN6PE" BBS = "K6FB-2" TNC = "GARAGE-TNC" RETRIEVE = "PB" SENDRECEIVE ***Notes*** 5. All session-specific variables must be set prior to executing this statement. 6. Related System variables used by the SendReceive statement are: BBS, TNC, MYCALL, TACCALL, RETRIEVE, FILTER 7. Opscripts does not perform any error checking on the existence of the BBS and TNC names entered on these variables. On a Send/Receive error, Outpost will report the problem, not Opscripts. 8. Outpost must be running for this statement to work. An error will occur if Outpost is not running.
SUBJECT	***Description*** System Predefined Variable. Holds the subject for this message. ***Syntax*** SUBJECT = "<subject text>" Default = blank ***Example*** #1. Subject = "Status of the W6TDM Repeater" #2. Subject = ReadFile("WX080608.txt") ***Notes*** 1. Subject Line prefixes will be inserted based on Outpost settings.
TACCALL	***Description*** System Predefined Variable. Holds the value of the tactical call. This variable is used by the SendReceive statement. ***Syntax*** TACCALL = <tac_call> Default = "-" ***Example*** #1. TACCALL = "CUPEOC" sets tactical call to CUPEOC #2. TACCALL = "-" turns off tactical call ***Notes*** 1. TacCall is turned off by setting the variable to "-".
TNC	***Description*** System Predefined Variable. Holds the value of the TNC. This variable is used by the SENDRECEIVE statement. ***Syntax*** TNC = <TNC_name> Default = blank ***Example*** TNC = "GARAGE-TNC" ***Notes*** 1. The value that you assign to the TNC variable is the name of a TNC that is already defined in Outpost. For instance, suppose you have a KPC3 that you define in Outpost and give it a name of "GARAGE-TNC". This assigned name is what you assign to the TNC variable. 2. If this TNC is not set up in Outpost, at the time the Send/Receive session is attempted, Outpost will generate the message: "Either the Station ID, BBS, or TNC is not selected…"
TO	***Description*** System Predefined Variable. Holds the call signs or tactical calls of the users for whom this message is intended. ***Syntax*** TO = "<call_sign> [, 2nd_address]" Default = blank

	Example #1. `To = "KN6PE"` #2. `To = "KN6PE, SMTP:kn6pe@arrl.net"` #3. `DistList = "K6KP, W6TDM, SMTP:kn6pe@arrl.net"` `To = DistList` ***Notes*** 1. All standard address rules are in force when addressing messages to a Winlink station.
TRUE, FALSE	***Description*** System Predefined Variable, CONSTANTS, used as part of a conditional test. ***Example*** `IF Exists(Fname) = TRUE then` ***Notes*** 1. TRUE and FALSE can be used to check for this case. Additional functions may be added in the future to take advantage of this.
URGENT	***Description*** System Predefined Variable. Holds the outgoing message URGENT Flag. ***Syntax*** `URGENT = TRUE | FALSE` Default = FALSE ***Example*** #1. `URGENT = TRUE` #2. `URGENT = FALSE` ***Notes*** 1. Once the URGENT flag is set, it is applied to all subsequent created messages. It is recommended that you explicitly declare whether a message should be URGENT or not. 2. Initially, URGENT defaults to FALSE.

ValidFileName()	***Description***
	Creates a valid full-path file name from a path and name components. This is typically used when creating files from Outpost messages, and there may be invalid file name characters in the Subject name.
	
	Syntax
	`<Var_name> = ValidFileName(<string>)`
	
	Example
	`SCRIPT`
	`Var FullName as String`
	`Var FixedName as String`
	
	`BEGIN`
	`SUBJECT = "CUP103: c:\data\Weather report.txt"`
	`FixedName = ValidFileName(SUBJECT)`
	`FullName = "c:\data\" & FixedName`
	`Print(FullName)`
	`END`
	
	Notes
	1. The following 9 characters work for Outpost subjects but are invalid file name characters: **: / \ * ? | < > "**
	2. The "**:**" character will be replaced with a "**;**"
	3. The **/ \ * ? | < > "** characters will be replaced with a "**~**"
	4. So, in the above example, the `FixedName` is set to…
	`CUP103; c;~data~Weather report.txt`

Var	***Description*** Declares a user-defined variable that can be subsequently assigned and manipulated ***Syntax*** `VAR <var_name> AS [STRING | NUMBER]` ***Example***

	Script VAR Fname as string VAR Shelter24 as string VAR x as number BEGIN ***Notes*** 1. All user-defined variables must be defined after the SCRIPT statement and before the BEGIN statement. 2. Variable names must start with a letter and follow with any combinations of letters, numbers and the underscore (_) character. All other punctuation are not allowed. 3. Var types are String or Number

**While…
Endwhile**
	Description
	Executes a block of statements as long as the condition is true.
	
	Syntax
	WHILE <condition>
	<statements>
	ENDWHILE
	
	Example
	SCRIPT
	VAR Fname as string
	
	BEGIN
	FINDFILE("c:\data\" & "*.txt")
	Fname = NextFile(0)
	
	While Exists(Fname) = TRUE
	Print(Fname)
	Fname = NextFile(0)
	ENDWHILE
	END
	
	Notes
	1. See Also: IF, LOOP

WriteFile()	***Description***
	Writes data to a named file
	
	Syntax
	WRITEFILE(<data>, <file_name>)
	data: a text string or variable of the data to be written
	file_name: a string or variable of the name of the file to be created
	
	Example
	SCRIPT
	VAR MsgID as number
	
	BEGIN
	FindMessage(1,4,"NOAA*") ' set up the msg search
	MsgID = NextMessage(0) ' loads the current msg
	
	while msgid > 0
	Print("Found Msg: " & SUBJECT)
	WriteFile(MESSAGE, Subject & ".txt")
	MsgID = NextMessage(0)
	endwhile
	
	END
	
	Example #2
	' Append a line of text to an existing file
	SCRIPT
	VAR Fname as string ' Name of a file
	VAR Fdata as string ' Contents of the file
	
	BEGIN
	Fname = "C:\data\Master.ini" ' set the file name

```
  Fdata = ReadFile(Fname)      ' Read the file contents

  Fdata = Fdata & CRLF & "Cmd=0"  ' append a line of text
  WriteFile(Fdata, Fname)      ' Write the new file contents
END
```

Notes

1. Any content can be written to a file. If the file already exists, it will first be deleted.
2. The data to be written can be the explicit string in quotations, or a variable containing the string.
3. In the above example, the `NextMessage` loads the next message and all its variables into the system variables: `BBS, FROM, TO, SUBJECT, MESSAGE`. The `WriteFile` statement writes the content of the variable `MESSAGE` (the current Outpost message) to the file by the name "`<subject>.txt`"; the file has the subject string in the title.
4. In the 2nd example, this is a way to append data to a file. Essentially, read the contents, append the addition, and write it back.

6 Error Messages

6.1 Compiler Errors

Anytime the compiler detects an error, it reports it with the offending script line number and ends. The following are the error messages that may be generated at compilation time

Error Message	**Meaning**
Error: "Variable Name" expected at line *x*	A VAR statement was executed and the variable name was not specified. The required format is: `VAR <var_name> AS [ STRING | NUMBER ]`
Error: "String or Number" expected at line *x*	A VAR statement was executed and the variable type was not specified. The required format is: `VAR <var_name> AS [ STRING | NUMBER ]`
Error: "Math Factor" expected at line *x*	Could not identify the parameter in parenthesis.
Error: "Identifier" expected at line *x*	Compiler was expecting a Variable name, but something else showed up.
Error: "Number" expected at line *x*	Compiler was expecting a number, but something else showed up.
Error: "String" expected at line *x*	Compiler was expecting a string, but something else showed up.
Error: "Quotation Marks" expected at line *x*	Compiler was expecting the ending Quotation Mark for a string, but none was found prior to the end of the line. **Resolution:** check each string for an ending quotation mark.
Error: "<character>" expected at line *x*	The compiler is anticipating a character or word, but it is missing. This type of error represents a syntax error with the script. The following may be reported as missing: ▪ `(` `)` usually missing the closing parenthesis around a function or arithmetic expression ▪ `,` missing the comma in a list of data items, usually with the `MoveFile` function ▪ `=` missing the assignment operator (equals sign) for an expression ▪ `SCRIPT, BEGIN,` or `END` - Minimum required script statements. ▪ `AS` - required in a Variable declaration statement ▪ `THEN, ENDIF` - minimum required constructs in an `IF` statement ▪ `ENDWHILE` - minimum required constructs in an `WHILE` statement ▪ `ENDLOOP` - minimum required constructs in an `LOOP` statement
Undefined identifier "__" at line *x*	A word was encountered in an expression that was not previously defined as a variable. One possible cause is misspelling a Reserved System Name. These are: ▪ `MYCALL` ▪ `TACCALL` ▪ `BBS` ▪ `TNC` ▪ `FILTER`

Error Message	Meaning
	▪ RETRIEVE ▪ FROM ▪ TO ▪ SUBJECT ▪ MESSAGE ▪ MTYPE ▪ SRNOTE ▪ LMI ▪ TRUE ▪ FALSE ▪ ON ▪ OFF ▪ CRLF
Duplicate identifier "__" at line *x*	A VAR statement is attempting to define a variable that is previously defined. The biggest contributor to this is the user attempting to define a variable with a name that matches a Reserved System Name. See the above list of reserved words already defined.

6.2 Runtime Errors

All Runtime errors will be written to the Runtime Monitor display.

Also, see the OnError Statement for setting up how to handle these errors.

Error Message	Meaning
Deleting file <name> ... failed	Source: `Delete` The script attempted to delete a file name but could not. Possible causes include: (i) file does not exist, (ii) file permissions are set as READ-ONLY.
>>> Move Error: Source file <path\name> does not exist	Source: `MoveFile` The script attempted to move a file name that did not exist.
>>> Move Error: Destination <path> does not exist	Source: `MoveFile` The script attempted to move a file name to a destination that did not exist.
>>> File Open Error: file <name> not found	Source: `FILEREAD` The script attempted to open and read a file name that did not exist.
>>> Outpost is not running; no Send/Receive Initiated.	Source: `SENDRECEIVE` The script attempted to initiate an Outpost Send/Receive session. However, Outpost is not running.
>>> Program <name> not found	Source: `Run, Runw` The identified program name was not found. Check the spelling and path to the program name.
>>>FINDMESSAGE: Folder value out of range (valid: 1..6)	Source: `FindMessage` The Folder value was either less than 1 or greater than 6. Check the value entered and try again..

Error Message	**Meaning**
>>>FINDMESSAGE: Field value out of range (valid: 1..5)	Source: `FindMessage` The Field value was either less than 1 or greater than 5. Check the value entered and try again..