# Everything You Always Wanted to Know About Writing JAWS  Scripts, But Didn't Know Whom to Ask

By Kenneth A. Gould

## Preface

The Microsoft Word version of this manual has been set up so  you are able to jump directly to any section shown in the  Table of Contents.  To do this, you can click anywhere on  the appropriate line in the Table of Contents, either on the  entry itself or on the page number to the right of each  entry.  One way to accomplish this with the keyboard is as  follows:

1. Find the section of interest in the Table of Contents by  moving there with the PC cursor.
2. Route the JAWS cursor to the PC cursor.
3. Click the left mouse button by pressing NUM PAD SLASH.  You will then jump automatically to the page of interest.

This manual also contains hyperlinked cross-references to  sections that are in the form See Section X, where X is the  hyperlinked section number.  You can jump directly to the  section by clicking on the section number reference with the  JAWS cursor.  After reading the section you have jumped to,  you can return to your original position in the Table of  Contents or to your original cross-reference at any time by  pressing ALT+LEFT ARROW.

This manual ?has been updated for version 3.7 of JAWS.

## Table of Contents

# 1 Introduction

## 1.1 What are Scripts?

First of all, let's start out with some definitions. What is a script anyway? When you perform a job on a computer, it usually involves a number of steps. You may have to press several keys to get you to the right part of a program to do a job, or you may even have to press several keys to open the program you want to use. Then, once you are where you want to be, you may have to press several more keys to enter the data you want. For example, you might be in a word processor and want to type your return address at the top of a letter. This could become a tedious task if you have to do it a lot, so wouldn't it be nice if you could have the entire return address typed out for you with one keystroke? That is exactly the type of thing a script can do. A script is a sort of mini computer program which combines many steps or keystrokes into one operation that you can activate quickly and simply. Well, at least that's one type of script. JAWS and other computer applications use scripts all of the time, and obviously they aren't always being used to type out return addresses. So, more globally, scripts are sequences of individual steps that can be used to activate and control a wide variety of computer processes from things as simple as entering repetitive strings of data to many of the things your computer does on an ongoing basis as part of its operation. It is this sort of function we are going to be concerned with in this manual. While you can certainly use what you will learn here to create a return address script, you will also learn how to control JAWS in ways which will make it behave more like you would wish.

Many aspects of computer control which we take for granted are actually under the control of scripts. We don't actually think of them as scripts since they are coded as part of the application's internal programming. For example, when you use cursor movement keys or call up a dialog box in Microsoft Word with a hot key, you are actually invoking a part of the program which performs a series of operations or steps. By our definition, this is a script. JAWS uses many different kinds of scripts which tell it what to do in certain situations or when you hit certain keys. What makes JAWS such an immensely powerful program is that the designers have decided not to hide away those scripts as part of the main program and, thus, make them inaccessible to the user. If they had done this, then JAWS would be a far less flexible tool and would be much less adaptable to the needs of unusual applications. JAWS' designers made the conscious decision to let you write your own scripts for a particular purpose and to let you see and modify the individual default scripts that control many aspects of the program's performance. The modified or newly-written scripts are then organized into individual script files which have the same name as (but a different extension from) the application for which they were created. Access to the scripts controlling JAWS' most fundamental behavior has been provided for two very good reasons. First, every user wants to be able to personalize the program to behave just as he or she would like. Second, while the JAWS designers have anticipated many of the screen reading situations which will occur with different applications, it is impossible to foresee every weird or nonstandard screen setup which the world's programmers will come up with. You have at your disposal a powerful arsenal of script commands which you can apply to the conquest of these unusually-designed applications. All you have to do is learn how to use them.

By the way, it is not always necessary to use a script to customize JAWS. For example, one can cause certain parts of the screen to speak or be silent by using the frames which can be created with the Frame Manager. As time goes by, there will surely be more features such as this added to JAWS that will allow you to do more customization without resorting to scripts. But there will always be situations too bizarre or convoluted for anything but a well-written script to cope with. Remember that after you have learned how to use the JAWS script language you have a very powerful tool that is much more incisive than any simple feature can be.

So, you're probably asking yourself now, what's the difference between a macro and a script? Well, in a word, there really isn't any difference. Scripts are macros, and that's all there is to it. Don't let the jargon confuse you. We'll call them scripts from now on, but "A rose by any other name," etc., etc.

## 1.2 Scripts versus Functions

JAWS scripts can be divided into two main categories. Those which are activated by a keystroke are called scripts, and those which are not are called functions. Let's first talk about the keystroke scripts. How does JAWS know that it should read the new line of text when you are in a word processor and hit the DOWN ARROW key? Well, it knows this because there is a script in the main JAWS script file (called the Default script file) which is tied or "bound" to the DOWN ARROW key and tells JAWS to read the new line. This particular script has the name "SayNextLine" and it is activated when you hit the DOWN ARROW key. This script is fairly complicated because it has to analyze where you are and what you are doing when you hit the DOWN ARROW key. It will check to see what kind of window you are in when you hit the DOWN ARROW key, and, if you are in a word processor text window, it will speak the line to which you move. There is a line in this script file that reads "SayLine ()" and it gets activated after your cursor moves on to the next line. This is the line of the script which actually performs the function which reads the line. There is a similar script called "SayPriorLine" which gets activated when you hit the UP ARROW key. This script uses the same "SayLine" function as part of its coding. When you hit the RIGHT and LEFT ARROW keys, two other scripts get activated, "SayNextCharacter" and "SayPriorCharacter". These scripts use a function called "SayCharacter" as part of their coding to speak the new character after your cursor moves on to it.  There are other key-bound scripts designed to operate when  many of the other keys on your keyboard are pressed.  These  scripts analyze the current situation and try to perform an  action appropriate for that situation.  If the situation is  very unusual and has not been anticipated by the JAWS  designers, then that is when you may have to do some  modifications to make it perform as required.

The scripts that are not bound to or activated by keystrokes  are called functions.  Since these functions cannot be  activated by a keystroke, they must be utilized in other  ways.  Functions also differ from key-bound scripts in that  they return some information after they finish executing, whereas key-bound scripts never do.  Once again, don't let  the jargon snow you.  A function is just a script that is  not tied to a particular keystroke and which returns some  information when it is done executing.  Why is this done?  There are two very good reasons.  First, some functions operate automatically when certain events occur.  There is,  therefore, no need to have such functions tied to keystrokes  because it is not a keystroke which activates them.  Second, sometimes a set of scripting statements is very generally  applicable to many situations and can be used over and over  again.  Instead of writing out this set of statements for  every script that needs it, it can be placed into a function  which can be utilized or "called" by any other script that needs it.

Thus, functions are subdivided into two categories, the ones  that trigger automatically when certain things happen and  those which only get triggered when they are "called" by  another script.  The first type is called an event function  because it runs automatically when certain system events  happen.  For example, the NewTextEvent function gets called  when any new text gets written to the screen, and the  BottomEdgeEvent function gets called when the cursor reaches  the bottom of a window.  Without the NewTextEvent function,  nothing at all would be spoken automatically by JAWS when  information appeared on your screen, and without the BottomEdgeEvent function, your PC would neither beep nor  speak a warning message when the JAWS cursor reached the  bottom of a window.  The second type of function, the type  not called automatically by certain Windows situations, has  no special name (believe it or not) so we'll just call it a  function.  These plain functions operate only when they are  called by another script or function.  This is done by  placing the name of the function within another script or  function.  If you do this, then when the script gets to the  line which has that function's name, it will cause that function to be run.  This process is termed "calling" a  function.  Yeah, I know, more jargon, but you'll just have  to get used to it.  Calling a function means, quite simply,  placing that function's name into another script so it will  get performed at the appropriate time.  For example, the NewTextEvent function mentioned above has a line in it that  reads "SayNonHighlightedText ()".  If the NewTextEvent  function has analyzed the new text written to the screen and  decided that it is properly handled by the  SayNonHighlightedText function, it will pass off the job to  that function by calling it and then let it speak the text  for you.  On the other hand, if the NewTextEvent function  decides that the screen information is highlighted text,  then it will call a different function named  "SayHighlightedText" instead, and it will let that function  do the speaking for you.  Decisions, decisions, decisions.  Well, that's what scripts do.  They analyze what's going on  and make a decision about how to handle the screen reading  for you.  If they don't do it right, your job is to modify  the relevant script so it does do it right.

By the way, you might have noticed that there was a left and  right parenthesis after the function name  SayNonHighlightedText mentioned above.  These parentheses  are always required when calling a function within a script.  Sometimes information is placed between these parentheses, but that's something we'll talk about later.  (See Chapter  7.5.2.)

The plain-Jane, non-event functions are further divided into  two types.  The first type is the list of over two hundred  functions which are available for you to use in the JAWS  script language.  These functions are called "built-in"  functions and are hard-coded into JAWS.  You can call them when creating scripts, but you cannot change them.  The  second type of plain function is the type present in the  script files, either created by JAWS' designers or added by  a user.  These are called user-defined functions.  After  these functions have been created, they will also appear in the function list for you to use within the scripts you  create.  Obviously, the main difference between these and  the hard-coded functions is that these can be modified by a  user, if necessary.

## 1.3 Constants and Variables

We need to introduce two other terms before actually getting  into script writing.  They are "variable" and "constant".  A  variable is a word of one or more letters which can be used  to store a value that is determined during the processing of scripts. This value is often not known in advance and can change one or more times during script processing. The value that is determined is assigned to the variable during the course of the script's operation, and the variable maintains that value until and unless it is changed by another script operation. The value of this variable can be used to make decisions by that script or another script. For example, in the statement Let X = 1, X is the variable, and we are assigning a value of 1 to it. X has the value of 1 until and unless we change it to something else or the operation of the script changes it for us. This is, by the way, the type of script statement we use to assign a value to a variable manually. In the course of running a particular script, we can have statements that do one thing if X equals 1, but do something else if X has a different value. Further, the value being held by X might represent the kind of window we are currently in, say a button, an edit field, or a list box. X would have a different value for each of these types of windows, and our script might need to perform different operations depending on what type of window we were in and, therefore, what value was assigned to X. If our script had the capability of looking into the Windows operating system and determining what type of window we were in, the script could then assign the appropriate value to X and the script could then continue its processing, deciding which of several courses of action to perform based on the value (i.e., type of window) being stored in the variable. JAWS does have this capability to look into the Windows operating system in this fashion, and this type of decision based on the current window type is exactly the kind of decision-making continually being performed by JAWS. The next time that same script runs, we may be in a different window, and the variable will, therefore, have a different value stored in it. The script would then decide to do something quite different, the action appropriate for that type of window. Again, scripts often must make decisions, and the value stored in a variable is one way they can decide what to do next.

Constants are quite a different matter because, as their name implies, they do not change their value. Once a value is assigned to a constant, it stays that way. This may seem to be a rather strange waste of time, but it is really quite convenient. Constants are a way of using easily-remembered names to store hard-to-remember strings of letters or numbers. For example, let's say our friend Joe has a social security number 589-43-3894. Further, let's say we have to work with ten people, each with his or her own social security number. We could make a constant called JoeSocSecNum and then place Joe's social security number in that constant by using the equation JoeSocSecNum = 589433894. We could do something similar with each of the other persons and his or her social security number. Then, when we wanted to print out a person's social security number, all we would have to do is ask the program to print out the value of each person's social security number constant. Since JoeSocSecNum is much easier to remember than 589-43-3894, we have made life much easier on ourselves by using a constant instead of trying to remember a long number. JAWS does this sort of thing a lot, and that's what we mean when we talk about constants in the context of the JAWS scripting language.

In case you are still a little confused about variables and constants, let's use another analogy that might help clarify them a little more. We're all familiar with touch-tone telephones that use a numeric keypad of buttons that we press to dial a telephone number. In addition to the numbers zero through nine and a couple of symbols, some telephones have some additional programmable buttons that can be used to store whole telephone numbers that we call frequently. One might, for example, store the sales number of his favorite software vendor in one of those buttons so he could quickly call to place an order without having to look up the number. Now, the buttons that contain the numbers zero through nine could be considered as constants since the numbers they contain never change. The first button on the top row is always a one, so it is a constant. The programmable buttons, however, can be changed at will to contain new numbers. You might, for example, have to change the software vendor button to a new phone number if the company moved to a new address. Since these programmable buttons can be changed at will, they are much more like variables, and they can be altered to contain the information we most need at any given time. Variables in scripts are like that, too. They can be changed as needed, either by the user or automatically by a script, depending upon circumstances.

## 1.4 Structure of Manual

As we mentioned above, individual JAWS scripts are arranged in files called script files. This manual contains all of the information you will need to write your own scripts and arrange them into files that will be used by your computer every time you run an application. The following is a list of the remaining chapters of this manual with a brief description of what each contains.

Part I: Getting Started With Scripts

?	Chapter 2 - "Scripts: A Brief Overview" provides a broad,  general understanding of how scripts and the JAWS Script  Manager function.  The difference between the default and application script files and how they work together is  also covered.  "Anatomy of a Script" gives a brief  introduction to the form a script takes so you can better  understand the construction of "our First Scripts" in the  next chapter.
?	Chapter 3 - "Our First Scripts" helps you to create two  simple scripts, compile them, and load them so that you  can gain a little confidence.
?	Chapter 4 - "The Script Manager" provides an in depth  tour of the Script Manager and all of its tools for  creating scripts and script files.
?	Chapter 5 - "Creating Simple Scripts" teaches script  writing on a small scale.  It is intended for those who  want to tweak the existing scripts and create scripts to  read textual information from the screen.
Part II: Creating Advanced Script Files
?	Chapter 6 - "Windows Program Structure" covers the  various identifiers used to classify and differentiate  among the many windows found in Windows programs.  It  also covers how windows are arranged in a hierarchical  order.
?	Chapter 7 - "Building Blocks of the Script Language"  provides details on the individual constructs that make  up scripts.  This is where we start to put things  together that will become scripts.  We'll also look at  when and how to use the various functions.
?	Chapter 8 - "Script Writing Techniques" starts putting  all this information to use.  It provides the techniques  used in writing advanced scripts.
?	Chapter 9 - "Debugging" covers how to work out the bugs  in your scripts along with some tips that help with the  process.
?	Chapter 10 - "Strategies for Attacking New Applications"  gives ideas on how to approach the problem of analyzing  and scripting a new application.
?	Chapter 11 - "Guidelines for Creating Distribution Script  Files" lists several recommended rules to observe for  those creating script files.
?	Chapter 12 - "Converting Macro Files" describes the  process of converting macro files written for versions of  JAWS for Windows prior to version 3into script files that  work with the current version.

Part I: Getting Started With Scripts

# 2 Scripts: A Brief Overview

Suppose you are using an application program that provides a  description of each menu item at the bottom of its window.  A sighted user would just glance down to see that  information whenever the purpose of that particular item was  unclear.  Sure, you could turn on the JAWS Cursor, go to the  bottom of the window and read the line, but wouldn't it be  easier if you could just press a hot key to read the help  text?  Or even better, how about if JAWS could read it automatically?  And, of course, you'd want to be able to  turn that automatic reading on and off.  Well, if this  application you are using had a script file written for it,  and if this script file contained a script to do these  things for you, you would then be able to have this sort of  convenient access.  If no one else had created this special  script for you, you could then create it yourself, provided  you have learned how to use the JAWS script language to  build your own scripts.  This improved access would help to  bring the efficiency of using this program up to that of a  sighted person, and it would improve your productivity.

## 2.1 What are Script Files?

JAWS is always under the control of scripts.  As we  discussed above, a script is a small computer program that  modifies the way JAWS performs for a specific situation.  JAWS script files are collections of individual scripts  which are loaded specifically for use with a given Windows  application.  These script files are loaded automatically by  JAWS whenever you enter a new application.  It is important  to understand that there are two types of script files:  default and application .  You can think of script files as  "stacked" in a sort of pile, with the default on the bottom  of the stack.  The default script file is loaded when JAWS  starts and is always active in all sessions.  An application  script file is stacked on top of the default when its  application loads.  JAWS knows to load the application  script file on top of the stack because the main part of the application script file name (i.e., the part without the  extension) is the same as the application program name.  Thus, our default script file is named DEFAULT.JSS, and an  application named SPREADSHEET.EXE would have a script file  called SPREADSHEET.JSS.  When you leave this particular  application, the specific script file it uses will be  unloaded, and all of the default scripts will again be  active until another application's script file is loaded.

The default script file that is loaded when you first start JAWS contains hundreds of scripts. These scripts give JAWS all of the information it needs to provide proper voice output in most situations. This file tells JAWS what to speak and when to speak it in most circumstances when running well-behaved applications. In many applications, however, there are situations which deviate from the normal. There may also be particular instances when customized reading of the screen is required. If these occur it may be necessary to create a script file just for this application. If such a script file has been created and exists in the JAWS subdirectory which contains the JAWS configuration files, then it will also be loaded when that application is run. This script file is the first one that is looked at when a particular keystroke is used or when a situation occurs which needs to call one of the event functions described earlier. (See Chapter 1.2.) If the application script file does not contain a script relevant to the situation, then JAWS will look in the default file. If the script is found in one of these two places, it will be executed as soon as it is found, and no further processing is required. IF no script is found in either location, then JAWS will pass the keystroke off to the application, and that keystroke will be performed just as if JAWS were not running. This hierarchy is very important. If a particular script which exists in the default file is not appropriate for the application being used, the user can place a customized version of that script in the application's script file, and that version will be used in preference to the one present in the default file.

Since scripts higher up in the stack take precedence, a script in the application script file will be run instead of one in the default script file if they are both bound to the same keystroke, and a function in the application script file will be called in preference to one in the default script file if they both have the same name. For example, that means if you have a script in the application script file assigned to CTRL+G, and you also have a script in the default script file assigned to CTRL+G, only the one in the application will be executed when you press CTRL+G. It doesn't even matter if the two scripts have different names. It is the keystroke binding which determines which script will run. If you attempt to assign a script to a keystroke which is already in use by the default file while using the JAWS Script Manager (See Chapter 2.2 and 4.), you will be given a warning message, and you will have the opportunity to continue with the assignment or change it to another key. This is to prevent you from accidentally disabling a script in the default file without realizing you are doing so.

There are some important considerations when deciding on the names and keystroke assignments for your scripts. If, on the one hand, the user gives the customized script a different name from the one in the default script file, then the user can also assign the script in the application's script file to the same keystroke it is assigned to in the default file with no adverse consequences. If, on the other hand, the user gives the application script the same name that it has in the default file, it is not even necessary to assign a keystroke to that application script. In either case, JAWS will execute the version present in the application's script file and will ignore the one present in the default file. This is because JAWS looks in both the default keymap file, DEFAULT.JKM, and the application keymap file, APPLICATION.JKM, to find out if a script with a particular name is assigned to a key when that key is pressed. JAWS will always execute a script in the application script file in preference to one in the default script file, even if the key assignment is made in the default keymap file. In other words, if both script files have a script with the same name and the key assignment is made in the default script file, JAWS will still know that it's supposed to execute the one in the application script file.

Functions, since they are not bound to keystrokes, must use a different mechanism. If the script is a function and, thus, is not bound to a keystroke, then a function with a particular name in the application script file will be run instead of such a function in the default script file that has the same name. So in the case of functions, it is the name that decides the outcome, not the keystroke.

Note that there is a special case which occurs when you have a script in the default script file and one in the application script file which both have the same name and which are not bound to a keystroke. In this case, JAWS uses the script name to determine which is active, and the script in the application script file will take precedence over the one with the same name in the default script file. In this special case, therefore, scripts not bound to keystrokes determine precedence in the same way that functions do. Thus, if a script is run using a PerformScript statement, the script in the application script file will be utilized in preference to the one in the default script file. PerformScript statements are an alternative way of running scripts, and they will be discussed later in this manual. (See Chapter 4.5.)

It was mentioned above that, if there is an application script file available for a particular application, it will be loaded automatically when that application is loaded. JAWS knows to do this because the main part of the script file name is the same as the name of the application. Starting with version 3.3 of JAWS for Windows, an additional capability was added which allows a user to load a script file of his or her choice at any time. This script file can have any name, and that name does not have to resemble the name of the current application. If such a script file is loaded, it will replace the regular application script file in the stack if such a file is present. This special script file will, in turn, be replaced automatically any time the user switches to a new application that loads its own script file. This new capability allows a user to load a script file that is tailor made to cope with unusual situations in an application where the requirements are very different from the rest of the application. The special script file could be loaded manually with a keystroke or automatically when certain windows or items appear on the screen. The function used to load special script files is called SwitchToScriptFile. It can be called manually or automatically by techniques which will be described in the remainder of this manual.

Finally, it is important to understand that scripts present in the default script file are available from any application being run, unless, as discussed above, the default script is superceded by a script in the application script file. So if you have a script which you wish to use from more than one application, you may wish to place it in the default script file rather than in a particular application script file.

## 2.2 What is the Script Manager?

JAWS provides you with all the tools you need to modify and create scripts. These tools are present in a program called the Script Manager. The Script Manager is a full-featured, text editor program. In addition to the usual features contained in most text editors, it has built in capabilities to help you create scripts, insert functions into your scripts, display reference information about these functions and scripts that have already been written, check for errors that violate the rules of correct script writing, and save and compile your script file. In addition, there are three hot keys which allow you to look through the scripts in your script file more efficiently. The function key, F2, can be used to jump forward from script to script, while SHIFT+F2 can be used to jump backwards from script to script. The hot key combination CTRL+L can be used to bring up a dialog which lists all scripts in the current file in alphabetical order in a separate window. This makes it easier to find a particular script within a large file if you know its name but do not remember where in the file it resides.

We said in the last paragraph that the Script Manager could be used to save and compile your script files. What do we mean by the word "Compile"? When you write scripts for JAWS in the Script Manager, you write them in plain text. JAWS, however, cannot use them in this form since it cannot understand plain text. When you save your script file in the Script Manager, it automatically performs an additional step which other text editors and word processors do not. It creates an additional file which has been converted from text into a binary format which JAWS can understand and use. The original text file is saved with a JSS (JAWS Script Source) file extension, and the compiled or binary version is saved with a JSB (JAWS Script Binary) file extension. This is why it is important always to create and save your script files using the Script Manager. If you edit and save your files using some other text editor such as Notepad, they will not be compiled when you save them, and JAWS will be totally unaware of the changes you have made.

There are three ways to start the Script Manager. These are (1) from an application by pressing INSERT+0, (2) pressing INSERT+F2 to invoke the Run JAWS Manager Dialog, followed by the letter S and the ENTER key, or (3) by choosing Script Manager from the JAWS Utilities menu. When you start the Script Manager from within an application with either method 1 or 2, it automatically opens the script file for that application. If no file exists, the Script Manager creates one. When you start the Script Manager from the Utilities menu, it does not open any file.

The Script manager will be discussed in much more detail below in the chapter titled "The Script Manager".

## 2.3 Anatomy of a Script

Scripts and functions can be divided into three main sections. The first line starts the script or function and is of the format of one of the following two lines:

```
Script ScriptName ()
Function FunctionName ()
```

If the script is bound to a keystroke, its first line will look like the first line above. If it is a function and, thus, is not bound to a keystroke, it will look like the  second line.  Sometimes there is an additional word before  the word Function, but that will be covered later.  (See  Chapter 7.5.3.)  In either case, the second word in the line  will be the script or function name.  This can be anything you want, but it is very helpful to make the name  descriptive of what the script or function does.  If you use  several words clumped together, capitalize the first letter  of each word so JAWS can pronounce it properly.  Spaces are  not allowed in a script name.  Some typical examples of script names are SayNextLine, SayPriorCharacter,  SayWindowTitle, and NextDocumentWindow.  Note that it is  very easy to tell from these titles what the scripts are  intended to do.  Finally, there will be a left and right  parenthesis at the end of the line.  Sometimes these  parentheses will contain some information, but that will be  discussed later.  (See Chapter 7.5.2.)

Next comes the body of the script.  This consists of all the  instructions you add during the script writing, such as  variable declarations, comments, flow control statements,  function calls, PerformScript statements, and arithmetic  operations.  You will learn what all of these types of instructions are as we proceed through the manual.  For  immediate information on the most important of these, See  Chapter 7.3.

Finally, the last line of the script will look like one of  the next two lines:

```
EndScript
EndFunction
```

Obviously, the first line is used to end a script, and the  second is used to end a function.  When you use the script  creation dialog of the Script Manager, as discussed below, a  blank script will be created with the first and last lines  already present on the screen with several blank lines in between.  All you have to do is fill in the body of the  script between these first and last lines.

## 3 Our First Scripts

Now let's write a couple of small scripts.  The first will  have JAWS speak our name on demand.  We'll place it in the  Notepad script file so that we can learn a little about  script writing without messing around in the very important  default script file.  Follow these steps exactly and you  will see that writing simple scripts is not all that hard.

1. If you do not already have JAWS for Windows running, start it.
2. Open Notepad from the Start Menu/Programs/Accessories group.
3. Choose Script Manager by pressing INSERT+0 or by pressing INSERT+F2 followed by S and ENTER. JAWS will tell you that you are in NOTEPAD.JSS, the script source file for Notepad.
4. You will note that there are no scripts in this file. Notepad needs no scripts, so opening the Script Manager from within Notepad started a brand new source file.
5. Choose New Script from the Script menu or use the accelerator key combination, CTRL+E.
6. The New Script dialog appears so we can name our script and write its documentation. For now, just follow the steps. We'll cover naming and documenting scripts in more detail later. (See Chapters 4.4 and 5.1.)
7. When the New Script dialog appears, the cursor is in the Script Name field. Type in SayName. Be sure to capitalize both the S and N, and don't put in a space.
8. Press TAB to move to the Can be attached to key checkbox, and press SPACEBAR to check it. This is the step which tells the Script Manager that this is a script and not a function. By checking this box, we make it possible to attach the script to a key.
9. Press TAB to move to the Synopsis field, and type in the words, "Say our Name."
10. Press TAB to move to the Description field, and type in the words, "The name of the person who wrote this script."
11. Press TAB to move to the Category field, and type in the word "Test." The Category field is an edit combo, so you can pick a category by typing one in or by arrowing down through the list to find one that is appropriate.
12. Press TAB to move to the Assign To field, and press CTRL+SHIFT+N to assign this keystroke to our script.
13. We won't need any other fields filled in for our script, so press ENTER to close the New Script dialog and insert our blank script into the editing area.
14. We are now back in the main text area of the Script Manager, and we have a blank script where the first line says "Script SayName ()" and the last line says "EndScript." We have been placed between these two lines. Arrow up to the first blank line after the first line, and you are ready to start writing the body of the script.
15. We want JAWS to speak a string of characters, that is, our name. We can do this by using one of JAWS built-in functions called SayString. Remember that JAWS' built-in functions are functions that are hard-coded into JAWS to

do particular jobs and cannot be modified by the user. These are the basic building blocks we use to create scripts. This one is designed to speak messages during script execution. Note that there is no space in the function name. Computer functions are often named by putting a couple of words together that describe the function. What we then need to do is tell the SayString function what we want it to speak. In other words, the SayString function tells JAWS we want it to say something, and then we have to tell the function what words we want it to say. We have to pass some information to the function for it to do its job. This information we are passing to the function is called a parameter, and that's one more piece of jargon for you to remember. In this case, the parameter we want to pass is the string of letters which comprises our name. Type your code line exactly as it appears on the next line except that you should substitute your own name for the letters XXX:

```
SayString ("My name is XXX")
```

17. You should now have the following three lines on your page exactly as they appear below. Remember that close is not good enough as computers expect us to be precise and perfect. Also, a few extra blank lines are not important. JAWS ignores blank lines in scripts.

```
Script SayName()
SayString ("My name is XXX")
EndScript
```

18. Now we need to save and compile the Notepad script file. Press CTRL+S and you should hear, "Compile Complete." If not, go back and retrace the steps and try again.

19. Close the Script Manager. This should place you back in Notepad. Test your work by pressing CTRL+SHIFT+N. How about that! Your first script.

Switch to another application and try the same hot key combination. Note that this time the script does not work. This is because we placed the script in the Notepad script file, and that file was unloaded when we left Notepad. If we had placed the script in the default file, it would have worked in any application unless that application's script file also had a script assigned to the same hot key.

Let's switch back to Notepad and try something else. Press the INSERT+1 key combination (using the 1 on the number row, not the keypad), and JAWS will say "keyboard help on." This is the help mode you can use to explore the keyboard without actually activating any of the keys. (You can leave the keyboard help mode by pressing INSERT+1 a second time.) Now press your hot key again. Wow! This time the hot key doesn't say your name, it says what the hot key does. Press the hot key combination twice very quickly, and you'll hear even more detailed information about what the hot key does. These pieces of information are the synopsis and description fields you filled out while creating the script. Now you can see why it is important to fill these fields out correctly.

For our next script, let's do something a little more practical. Open the WordPad application from the Start Menu/Programs/Accessories group. Now drop down the file menu by pressing ALT+F. Press the DOWN ARROW seven times, and you will land on a line that speaks the name of the last file you opened in WordPad. (If you've never opened any files in WordPad, do so now so you will have this line in your file menu.) Wouldn't it be nice if we could read this line and know what the most recently opened file was without using all of those keystrokes? Let's write a script to do just that. As before, follow the steps as described below.

1. Open the Script Manager by pressing INSERT+0, or INSERT+F2 followed by S and ENTER.
2. As before, you should be placed in a blank script file named WORDPAD.JSS. Start a new script by selecting New Script from the script menu or by pressing the CTRL+E accelerator key combination.
3. You'll be placed in the Script Name field. Type in LastOpenedFile without any spaces and capitalized as shown.
4. TAB to the Can Be Attached to Key check box and check it.
5. TAB to the Synopsis field and type in "Says last opened file." (Do not include the quotation marks, they are only used here to show you what to type. This comment applies to all quotation marks used to tell you what to type.)
6. TAB to the Description field and type in "Says the name of the last file opened in WordPad."
7. TAB to the Category edit combo and arrow down until you reach the Say category.
8. Tab over to the Assign to Hot Key field and press CTRL+SHIFT+L to make this your hot key.
9. TAB over to OK, and press ENTER.
10. We are now back in the main text area of the Script Manager, and we have a blank script where the first line says "Script LastOpenedFile ()" and the last line says "EndScript". We have been placed between these two lines. Arrow up to the first blank line after the first line, and you are ready to start writing the body of the script.
11. This script is quite a bit more complicated than the

last one. Shown below is the completed script, including the first and last lines that were placed in the blank source file by the Script Manager. Type the second through the next to last lines into your blank script, and you should end up with something that looks just like what is shown below

```
Script LastFile ()
SpeechOff ()
{Alt+F}
Pause ()
NextLine ()
NextLine ()
NextLine ()
NextLine ()
NextLine ()
NextLine ()
NextLine ()
SpeechOn ()
SayLine ()
SpeechOff ()
{escape}
{escape}
Pause ()
SpeechOn ()
EndScript
```

So, what in the world are all of these statements doing? Well, you may have noted that when you pressed ALT+F in WordPad to open the file menu, JAWS said "menu active, file, new, CTRL+N." We don't want to hear all that stuff during our script execution, so the second line in the script, SpeechOff () turns off speech until we're ready for it. Then the statement {ALT+F} sends the ALT+F command to WordPad, just as if we'd typed it in from the keyboard. As you'll learn later (See Chapter 8.8.), we always place keystrokes we want to send to the application between left and right curly brackets or braces. Then we have a pause statement that gives the application a chance to drop down the menu before the script proceeds. This is done a lot when asking applications to perform activities. Without this, our script might get ahead of the application and do things before the application is ready. Then we arrow down seven times with the seven NextLine statements. This brings us to the line where the last opened file is listed. Before we can speak this line, we must turn the synthesizer back on with the SpeechOn statement. Next we have a SayLine statement which speaks the contents of the current line. That's the actual function which gives us the data we want.  Now we have to turn the speech back  off with the SpeechOff function so we don't hear all the  stuff JAWS normally speaks when exiting a menu.  Now two  Escape statements will get us out of the menu.  Note that  here, again, these must be between curly brackets.  Then  we have another Pause statement to allow the menu time to  disappear, and finally we finish by turning the speech  back on with a final SpeechOn statement.

1. Now you should save and compile your script by pressing  CTRL+S.
2. Return to WordPad, and press your new hot key,  CTRL+SHIFT+L.  You should be rewarded by hearing JAWS  speak the name of the last file you opened.

If you think this is great, just wait.  You are just  beginning to experience the kinds of things you can do with  this fabulous scripting language.  No longer will you be at  the mercy of some dumb programmer who didn't follow normal  rules and write the application so it could be easily accessed with speech.  No longer will you be forced to  execute a dozen keystrokes to get a piece of information off  the screen that your sighted friends can access just by  glancing at the screen.  No longer will you have to wait for  someone else to configure a new application for you.  It may  take some work to learn this language, but you will be  rewarded with the capability of making your applications  speak and behave much more like you wish.  Learn this stuff  well, and you will be able to access your applications much  better and more efficiently than you would ever have  imagined could be possible.

One more word of encouragement is necessary before getting  to the real nitty gritty of the scripting language.  Sometimes it's daunting to see a completed script such as  the one above that works just the way it should.  The new  script writer looks at such a script and says "How in the  world did they figure out which statements to use and the  correct order to put them in?"  It must be understood that  scripts do not pop out of thin air as a correctly and  completely-finished product.  The author tried many  different configurations of the above script before coming  to the one that worked best, the one shown above.  The first  version did not contain any SpeechOff or SpeechOn  statements, and the resulting script spoke much too much  information.  So these statements were added.  But several  trials were necessary before the best placement was found.  Then the script still did not work correctly until Pause  statements were put in the right places, a process that  required more trial and error.  The point is that you will have to make your best guess and try something. If that doesn't work, try something else. As you gain more experience, you will get it right more quickly. But the only way to learn this new language is by the trial and error process which occurs as you make your scripting attempts. Don't be afraid to try things. The worst that will happen is that the script won't do what you want it to, and you'll have to try something else. You'll make mistakes, but that's part of the process. The more experience you gain, the faster and easier it will become. But if you don't try to write some scripts, you'll never learn how to do it.

By now you should be getting a mental picture of why we need scripts , how JAWS uses them to improve your efficiency, and how the default and application script files work together. The next two chapters of Part 1 are titled "The Script Manager" and "Creating Simple Scripts." These are followed by Part II, "Creating Advanced Script Files." Anyone who is not thoroughly familiar with the Script Manager and all of the tools it contains for creating and documenting scripts should read the chapter describing the Script Manager. Then, if you want to learn to create scripts to read textual information from the screen and to perform other simple functions, continue with the "Creating Simple Scripts" chapter. Those who want to learn to create advanced scripts that involve dealing with windows structure, using programming constructs and manipulating JAWS settings and who already understand how to write simple scripts should skip this chapter and continue with Part II, "Creating Advanced Script Files."

## Homework Assignment #1

Here is your first homework assignment and a chance to get a little more experience in script writing. You probably noticed during the writing of the last script that the file menu in WordPad doesn't just show the last file you opened, it shows the last four files opened. Modify the preceding script so that it reads all four of these files, not just the first one. One solution to the problem is shown in Appendix A (See Chapter 0.), but don't look at it until you've had a good try at making it work yourself.

# 4 The Script Manager

The use of the Script Manager, the script creation dialog, and the various other tools for creating scripts and script files will be discussed in this section. As briefly described earlier (See Chapter 2.2.), the Script Manager is more than just a text editor. It provides the user with a suite of tools that make script and script file creation much easier. We'll divide the description of the Script Manager into five parts. These are (1) the menus, (2) the various Script Manager file types, (3) include statements, (4) the New Script and Script Information dialogs, and (5) the Insert Function and Insert PerformScript dialogs.

## 4.1 The Menus

A detailed description of the Script Manager menu items can be found in Appendix B.

## 4.2 Script Manager File Types

We have already mentioned that there is more than one type of file which the Script Manager can generate. Listed below are the five types of files associated with the Script Manager and what each one is for.

? JSS - As discussed above, this is the JAWS Script Source file which the user creates to make application-specific script files.
? JSB - This is the JAWS Script Binary file which is created by the Script Manager when you save and compile a JSS file. This file is neither directly created nor edited by the user.
? JSM - This is the JAWS Script Message file and contains all of the spoken messages used by script Say and SayMessage functions. When one creates advanced script files that are to be distributed internationally, it is preferable to use message numbers in the Say and SayMessage functions rather than place the actual text there. This makes it easier to translate the messages into other languages. The JSM file contains the statements which tell the script file what to speak for a given message number. These message numbers are actually just constants that are defined as being equal to the message you want spoken. If you look at the structure of the DEFAULT.JSM file, you will see that the message numbers are defined in exactly the same way as all other constants. For example, the default JSM file, DEFAULT.JSM, contains statements like MSG1_L = "End" and MSG2_L = "home". You will note that the statements are placed one to a line and a comma is at the end of every statement except the last one in the file. If a script in DEFAULT.JSS encounters the line Say (MSG1_L, OT_JAWS_MESSAGE), it will say "end", and if it encounters the line Say (MSG2-L, OT_JAWS_MESSAGE), it will say "Home". If you are creating small script files for

personal use, you may not wish to bother with JSM files, but if you encounter message numbers in scripts written by others, you will know to look in the associated JSM file to find out to what the message number refers. A discussion of long (-L) and short (-S) messages and their utilization for customized verbosity settings is presented later. (See Chapter 8.9.)

? JSH - This is the JAWS Script Header file, and it contains other information relevant to the associated JSS file such as definitions of constants and global variables. We have already briefly discussed constants and variables and will be talking about them more extensively below. Global variables are simply variables that can be used by more than one script. It is necessary to tell JAWS what the value of a constant is, or it will not know that information. Similarly, it is necessary to tell JAWS that you will be using a certain variable with a certain name, or JAWS will have no idea that this particular string of letters is to be considered a variable. This process of telling JAWS about the constants and variables we will be using is called declaration. If you just have a few constants and global variables, you can place them directly into the top of the JSS file you are creating. This procedure will be discussed more fully below. Regular or local variables are always declared within the script itself. However, if you have large numbers of constants and global variables, it is better to place them in a JSH file. Examine the files HJGLOBAL.JSH and HJCONST.JSH for examples of files containing global variables and constants.

? JSD - This is the JAWS Script Documentation file and contains all of the descriptive information about each script which you enter when you create a new script by using the New Script Dialog. This information is entered into the JSD file automatically when you create a script in this fashion. This file can be edited manually if desired.

## 4.3 Include Statements

Let's say you decide to use a JSM and/or JSH file as parts of a set of script files you are writing for a new application. So you go ahead and create these files, but then what? How does the main script file, the JSS file, know that these other files exist and are part of the script set? Take a look near the top of DEFAULT.JSS, and you will see statements like the following:

```
Include "hjglobal.jsh" ; default HJ global variables
```

Include "HJCONST.JSH" ; default HJ constants  Include "default.jsm" ; message file

These are called include statements, and their purpose is to  tell the JSS file that all of the information in the  included files is part of the set.  When you save the JSS  file and create the JSB file, all of the information in the  included files will also be compiled into the resulting JSB file.  Thus, the scripts in the JSS file will know about the  definitions contained in the included files.  The syntax for  these include statements should be followed exactly as  shown.  Note that the file name includes the extension and  is enclosed in quotation marks.  Everything after the semicolon is a comment and is simply there to provide the  reader with information.  Everything following a semicolon  on a given line is ignored by the compiler.  These comments  are optional.

## 4.4 The New Script and Script Information Dialogs

The New Script dialog is found under the Script menu, and  the Script Information dialog is found under the View menu  as the Documentation option.  These two dialogs are the  same, except that the New Script dialog is displayed when  one first creates a script, and the Script Information dialog is displayed when one wishes to review documentation  for a script which already exists.  The accelerator keys for  these two dialogs are CTRL+E AND CTRL+D, respectively.  Most  of the menu options in the Script Manager have accelerator  keys, and the user can easily become familiar with the ones  used frequently by examining the menus.

There are two tabs in these multi-page dialogs, the General  tab and the Parameters tab.  Each of these is described  below.

### 4.4.1 General Tab

The entries in the General tab are as follows:

? Script Name - This is where you enter the name of your  script or function.  It is helpful to use a name  descriptive of what the script does.  You may use several  words concatenated together.  Start each word with a  capital letter so JAWS will pronounce the name as  separate words.  No spaces are allowed in the name.

? Can Be Attached to Key - This is a checkbox.  If you  check it you will create a script.  If not, you will  create a function.  Remember that scripts are attached to keys and functions are not.

? Synopsis - This field should contain a brief statement of what the script does. This is used if you enter the Keyboard help mode (INSERT+1) or Key Word help (SHIFT+F1) in the Script Manager. The synopsis is accessed by pressing the key combination you are interested in after turning on Keyboard Help.

? Description - This field should contain a more detailed explanation of what the script does. This description is used if you enter the Keyboard help mode (INSERT+1) or Key Word help (SHIFT+F1) in the Script Manager. The description is accessed by quickly pressing the key combination you are interested in twice after turning on Keyboard Help.

? Category - You can type in a category name or choose one from the combo box. This Category field isn't currently used by any JAWS feature, but it may be implemented in the future to allow sorting of your scripts by category.

? Assign to Hot Key - This field is only available if you checked the "Can Be Attached to Key" checkbox. Depress the key combination you wish to use for a hot key to enter your choice. If the choice you make is already assigned to another script, you will hear a warning message and be given the opportunity to continue with the assignment or choose a different one.

? Function Returns - This choice is only available if you did not check the "Can Be Attached to Key" checkbox. The five choices are Handle, Int, Object, String, and Void. Select one of the first four if your function is designed to return one of these types of data to the calling script. Select Void if you don't need to use any value returned by the function. Whichever one you choose will appear on the first line of the script before the word Function. A more detailed discussion of returns will be given later. (See Chapter 7.5.3.)

? Return Description - This choice is only available if you did not check the "Can Be Attached to Key" checkbox. This field should contain a brief description of what information is being returned by the function and how the information is meant to be used.

## 4.4.2 Parameters Tab

This tab contains information about parameters used by the function, if any.  A parameter is data that the function  needs to have in order to do its job.  As with variables,  the data can be in the form of an integer, string, handle,  or object.  This tab is never used if you are creating a  script rather than a function.  It is also not used if the  function needs no parameters.  Entries for this tab are as  follows:

? Existing Parameters - This listbox will show already  existing parameters, if any.  It will also show  parameters as you add them.  You can arrow up and down to  select a parameter for subsequent deletion.

? New Parameter - If you wish to add a parameter, type its  name in this field.

? By Reference -To understand this item, you must  understand that parameters are used to transfer data from  the calling script to the function.  Normally, this data  transfer is a one way street.  This default situation is  known as passing the parameter "by value."  Thus, when  you call the function from the script, the current value  of the parameter as it exists in the script will be copied and sent to the function.  During the execution of  the function, it is possible for the value of the  parameter to change.  Since the parameter information  exchange is normally a one way street when the data is  passed by value, the script will not be aware that the  parameter's value has changed.  When the function is done  and returns control to the script, the script will  continue on with its original value for the parameter.  Checking the By Reference checkbox will change the one  way street into a two way street.  If this checkbox has  been checked, changes in the value of the parameter that  occur during the execution of the function will be known  by the calling script.  This makes it possible to change  a parameter's value within a function and then have the script use that changed value.  This is because passing  by reference passes the data's memory address rather than  the value to the function.  If the function changes the  value at this address, the calling script or function,  which uses the data at the same address, will use the  changed value.

? Description - This field should contain a very brief  description of what the parameter is for.

? Available Types - You should choose either Handle, Int,  Object, or String from this listbox, depending on which  type of information this parameter is meant to pass.  As will be discussed later, handle refers to a window  handle.  (See Chapter 6.)

?	Add - This button will be available if you have filled in  the preceding fields.  Use the SPACEBAR or ENTER key to  add your new parameter to the Existing Parameters  listbox.

?	Remove - This button will be available if you have  highlighted an entry in the Existing Parameters listbox.  Choosing this button with the SPACEBAR or ENTER key will  delete the highlighted parameter from the list.

## 4.5	The Insert Function and Insert PerformScript Dialogs

Under the Script menu, there are two selections, Insert  Function Call and Insert PerformScript.  These are used to  insert functions and to call other scripts from within a  script.  The use of these two tools is described below.

?	Insert Function Call - This brings up the Insert Function  dialog, which is a list of over two hundred functions  which you can use in your script construction.  You will  hear the title "Insert Function 1."  The term Function 1  means you are at the first level of this dialog.  The  first field in this dialog is an edit box.  The next tab  is an alphabetical list of all the available functions.  If you happen to know the name of your function, you can  start typing it in the edit box, and the highlight in the  alphabetical list will automatically move to the function  whose name starts with the same letters you've just typed  in.  You will also notice that a description of the  highlighted function is read to you each time you press  another keystroke.  As soon as you've typed in enough  letters to hear the name of the function you are  searching for, just press ENTER to select that function.  If you don't remember the exact name of the function,  just type in enough of the name to get you to the right  part of the list, then TAB over to the list, and arrow up  and down until you find the function you want.  You can  also type in function names while you are in the list, and the Script Manager will move to the function with the  correct name.  You will also hear function descriptions  as you arrow up or down in the list.  When you hit the  ENTER key to select a function, one of two things will  happen.  If this function does not require any  parameters, you will be returned to the main Script  Manager editing window, and you will see that the  function has been added to your script.  However, if the  function does require one or more parameters, another dialog containing an edit field will pop up asking you to  insert the parameter.  For example, if you choose  SpellString as a function, you will be placed in an edit  field where you can insert the text or message you wish  spelled.  This is the parameter required by the  SpellString function.  If the function you have chosen  requires you to choose another function as the parameter,  you can hit the key combination ALT+I, and a new dialog  will open which looks very much like the Insert Function  Dialog.  You will hear the title "Insert Function 2".  The only difference is that the only functions that show  up in this list are the ones the Script Manager thinks  are appropriate choices for this parameter.  Therefore,  not all of the functions of the main list will appear in  this sub list.  You must either choose a function from  this list or type the function name directly into the parameter edit field when it first appears.  Then hit  Enter.  Continue with this process until all of the  required parameters have been entered.  Sometimes you  will even have to go to a third or higher level, i.e.,  "Insert Function 3", "Insert Function 4", etc. by hitting  ALT+I again.  (This process of using functions as the  parameters for other functions is called "nesting", and  it will be discussed more fully later in the section  titled "Using Nested Functions."  See Chapter 8.6.2.)  The Script Manager will then return you to the main  editing window, and your function with all of its  parameters will be present in the script.  While you are  in the Insert Function dialog, you can also TAB over to  the Description and Returns fields to examine information  about the function description and return information, if  any, in more detail.

It is also worth noting that the function list in this  dialog contains both the hard coded functions built into  JAWS and the functions that have been defined by any users  in the current script file.  Thus, if you define a new  function for the script file, you will see it appear in the function list after you compile.

? Insert PerformScript - When you select this option, you  will be placed in a list box containing all of the  available scripts you can call from the present script  you are writing.  You can arrow through this list or  start typing the name of the script to jump to it.  Pressing ENTER on one of the names, for example, a script  called "SayMyName," will cause the line shown below to appear in your script.

```
PerformScript SayMyName()
```

This line will cause the script "SayMyName" to be executed, just as if you had hit the keystroke yourself. This is a way to reuse scripts you have already written without reproducing the code all over again.

## 5 Creating Simple Scripts

An example of a simple script is one in which you might wish to read some text located somewhere on the screen with a single keystroke. One way to do this would be to select a cursor, move it to the specific point on the screen where the text is, and then read the text. In this section, we'll go over how to create scripts by putting together functions that do those sorts of things.

### 5.1 Script Documentation

Each script has associated documentation. As discussed above, you enter this documentation as you create a new script. Simple scripts are always assigned to keys and require only five pieces of documentation. The following is a description of each field that is required for a simple script.

? Script Name - Enter the name for your script.
? Can be attached to Key checkbox - Always check this box for key-bound scripts.
? Synopsis - Provide a brief description of your script's purpose.
? Description - Provide additional information about your script.
? Category - Choose a category for your script. You assign the category names either by typing them in or choosing them from the edit combo list. Try to choose meaningful ones so that you can list them by logical category later, in the event that this capability is added to the Script Manager.

### 5.2 Individual Script Structure

Remember that a script file is made up of one or more scripts. As discussed above, a script must have the following pieces in this exact order:

1. First is the script begin keyword. This is simply the word "Script" followed by a space.
2. On the same line following the word "Script" is the script name. This is usually several words concatenated together that describe the action of this script, such

as, CloseDocumentWindow.  Note that each of the  concatenated words is initial capped, so JAWS can read  the name correctly.  () follows the name to complete the  script begin statement. All of this is automatically  entered by the Script Manager when you create a script  using the New Script Dialog.  (If you find that your copy  of JAWS is not pronouncing concatenated words like this  properly, check the Mixed Case Processing option of the  Text Processing section of the Set Options menu in the  Configuration Manager.  This checkbox should be checked.)
3.	Next comes the body of the script.  This consists of all  the instructions you add during the script writing, such  as variable declarations, flow control statements,  function calls, PerformScript statements, arithmetic  operations, and comments.  (You will learn what all of these types of instructions are as we proceed through the  manual.)  This could be as simple as just saying a string  as we did in "Our First Script" or many functions to  perform a complex operation.
4.	The script end keyword , EndScript, is always the last  line of a script.  This is automatically entered by the  Script Manager when you create a script.

## 5.3	Using Script Functions

Now, let's go over some of the commonly used functions for  moving around on the screen and reading text.

### 5.3.1	Reading Text

Below are some functions used to read text from the screen.

?	SayCharacter() - Reads the character at the position of  the active cursor.
?	SayWord() - Reads the word at the position of the active  cursor.
?	SayLine() - Reads the line at the active cursor.
?	SayToCursor - Reads from the beginning of the line up to  the active cursor.
?	SayFromCursor - Reads from the active cursor to the end  of the line.
?	SayColor - Says the font color at the active cursor.
?	SayFont - Says the font style and point size at the  active cursor.
?	SayTextBetween - Speaks all text between two column  coordinates on the line of the active cursor.
?	SayWindow - Reads the window located at the active  cursor.
?	SayToBottom - Reads all text from the position of the  active cursor to the bottom of the window.

? SayString - Speaks string data, usually either the contents of a string variable or a specific message. If actual text is used, it should be placed inside of quotation marks within the parentheses that follow the function name. If a variable or constant is used, no quotation marks should be present. It is now recommended that the SayString function be avoided in favor of the SayMessage function (see below).

? Say - This function speaks a string of text, much like the SayString function, but it has a second parameter called an output mode. These output modes allow the message to be spoken with a particular set of speech characteristics. It is possible to use separate output modes to speak title lines, dialog controls, menu items, etc, with different voices or at different verbosity levels. By using the SayMessage function instead, it is possible to assign short and long messages to many output types for JAWS Help and other information.

? SayMessage - This function takes short and long messages supplied as parameters and speaks the appropriate message based on the specified output type, also supplied as a parameter. This function is similar to Say, except that it is possible to assign short and long messages to many output types for JAWS Help and other information. It is now recommended that the SayMessage function be used in preference to the SayString function.

? SayAll - Says all readable information from the point of the active cursor to the bottom of the window. If the PC cursor is active, JAWS scrolls the screen by moving the PC cursor down. If the JAWS cursor is active, the rest of the window is read by moving the JAWS cursor down a line at a time.

? SayCharacterPhonetic - Uses special pronunciation rules to read the character located at the position of the active cursor. Thus, A is pronounced alpha, B bravo, etc. The associations between characters and their phonetic pronunciations are made in the [PhoneticSpell] section of .JCF files. Which words are used can be changed by the user, if desired.

SayControlInformation - Handles the speaking of controls for which SayWindowTypeAndText and related functions do not sufficiently describe the control. This function is designed to speak a control that requires custom processing (any control for which SayWindowTypeAndText do not properly speak the name and type of the control). This function is designed to honor the user's output mode settings for each component of the control's description. This function takes six parameters, five of which are string parameters. The first parameter is the window handle of the control that is to be spoken. The next five parameters include one parameter for each component of a control's description.  Each string parameter has a corresponding Output Mode.  The  5 string parameters are String strControlName = Either OT_DIALOG_NAME, OT_DOCUMENT_NAME, or OT_CONTROL_NAME,  depending on the type of window that is to be spoken, String  strControlType = OT_CONTROL_TYPE, String strControlState =  OT_ITEM_STATE, String strContainerName = OT_CONTROL_GROUP_NAME, and String strContainerType =  OT_CONTROL_TYPE.  This function works by building a string  based upon the five components of the control description, adding each component only if the user has specified that  this item should speak in the current verbosity level.  Then  this function calls the Say function with the constructed  string as the first parameter and OT_NO_DISABLE as the  second parameter.

? TsayFocusRect - Says the contents of a focus rectangle.  Returns TRUE if any text was spoken, FALSE otherwise.  A  TrackFocusRect=1 statement should be added to the [OSM] section of the application's JCF file for this function  to work properly.

? SayFocusRects - Says the contents of focus rectangles.  If there is only one such rectangle, this is exactly like  SayFocusRect.  If there is more than one, this function  says the contents of all of them, while SayFocusRect says  only the first one.  A TrackFocusRect=1 statement should  be added to the [OSM] section of the application's JCF  file for this function to work properly.

? SayChunk - Speaks the chunk of information to which the  active cursor is pointing.  A "chunk" is text and graphic  information that was written to the screen in a single  operation.  SayChunk is similar to SayField, however, the  SayField function uses logic to determine the text that  is to be spoken, while SayChunk simply reads the text  that was stored in the Off Screen Model as a single unit.

? SayField - Reads the field of text where the active  cursor is pointing.  A "field of text" is a section or  block of text that has a common attribute, i.e., bold,  underlined, italics, or strikeout.  The use of the  attribute must be contiguous.  The SayField function uses  logic to determine the text that is to be spoken, while  the SayChunk function simply reads the text that was  stored in the JAWS Off Screen Model as a single unit.

? SayFrame - Speaks the contents of the specified frame.

? SayFrameAtCursor - All text within the boundaries of the  frame that contains the active cursor is spoken.

? SayInteger - Speaks numeric, integer data, often the  contents of an integer variable.

? SayParagraph - Reads the current paragraph from the  beginning

? SaySentence - Reads the sentence containing the character

on which the active cursor is positioned.
? SayUsingVoice - Speaks a string of text using a specific set of speech characteristics called voice context.
? SayCell - When in a table or spreadsheet, speaks the contents of the current cell.
? SayColumnHeader - When in a table or spreadsheet, speaks the contents of the column header.
? SayRowHeader - When in a table or spreadsheet, speaks the contents of the row header.

## 5.3.2 Moving Around on the Screen

Before you attempt to read something from the screen with a script, you'll probably have to move one of the cursors to the location of the item you wish to read. Therefore, it's important to understand something about cursors. For those using JAWS without a Braille display, there are four kinds of cursors, the PC cursor, the JAWS cursor, the Invisible cursor, and the Virtual PC cursor. Those using a Braille display will have a fifth cursor, the Braille cursor. (This cursor is only used internally by the Braille scripts and should never be left on after a script completes its work.) If you are editing or entering text, the PC cursor is where characters would be placed when you type. (This editing or text insertion form of the PC cursor is also called a caret.) However, when you are using a menu or tabbing through some selections such as buttons in a dialog box, there is no caret on the screen. In these situations, the PC cursor is the focus. If you wish to read other areas of a window or read a part of the screen where the PC cursor cannot go, you can switch to the JAWS cursor. The JAWS cursor can be moved anywhere within the boundaries of the current real window, making it possible to explore areas of the screen where the PC cursor cannot go. A real window is defined as a window that has a title. This means that your JAWS cursor is normally only restricted to movement within the boundaries of the highest level parent window with a title, starting from the window you are in currently. This is what we mean when we say the JAWS cursor is restricted by the boundaries of the current real window. (If you wish to restrict the JAWS cursor to the window you are currently in, even if it is not a real window, you can do so by using the keystroke combination, INSERT+R.) Since the JAWS cursor always moves the mouse pointer to its location, the mouse pointer is always positioned and ready to be clicked at the location of the JAWS cursor should you decide to do so. The third kind of cursor is called the Invisible cursor. In one way it is just like the JAWS cursor as its movement is also only restricted by the boundaries of the real window, but it has no visible entity on the screen since it does not bring the mouse pointer with it.  The Virtual PC cursor is a  special case of the PC cursor which is turned on  automatically in certain applications such as Internet  Explorer 5, Outlook/Outlook Express, and Eudora 4.X where a  Microsoft browser or browser style viewer is being used.  The Virtual PC cursor is created within a special JAWS  buffer and is employed to allow the user to have a cursor  that can navigate around the viewer window even though that  viewer does not actually have a normal PC or caret.  The  user is actually navigating in the virtual buffer created by  JAWS and feels as if navigation were occurring within a word  processor type of application.  This makes reading in such  applications much easier.

How do we use these cursors?  It's generally impractical and  often impossible to go exploring around the screen with the  PC cursor.  Therefore, before we go to read some text from  the window, we usually first switch to one of the other two  cursors.  One can pretty much use either the JAWS or the  Invisible cursor to do the reading for you, so to some  extent it's just a matter of choice.  However, there are  some situations where you will definitely want to use one of  these two cursors in preference to the other.  If you plan  to click a button or take some other action with the mouse,  it will be necessary to use the JAWS cursor.  Since the  mouse pointer accompanies the JAWS cursor in its travels, it  is then a simple matter to click the mouse as it's already  in the correct position.  If you do not wish to move the  JAWS cursor from its current position, or there won't be any  need to click on anything, the Invisible cursor is the best  choice.  There is one particular situation which comes to  mind when you definitely won't want to move the JAWS cursor.  Sometimes in Windows, information text will appear somewhere  on the screen, usually the status line, when the mouse  pointer is at a particular location.  If you move the JAWS  cursor to read the text, the mouse pointer will move with  it, and the text will disappear.  In this situation, you  must use the Invisible cursor to read such text.  In  general, it's a good idea to use the Invisible cursor to  read text when no clicking is necessary, unless there is a  particular reason to use the JAWS cursor.

In some cases, it is necessary to move one of the cursors  such as the JAWS cursor, even though you want it to be back  in its original location when the script is finished.  For  this reason, there are the SaveCursor and RestoreCursor  functions that you can use to put everything back as it was  before your script was executed.

Here is a list of the functions that you can use to select and manipulate the cursors.

? PCCursor() - Activates the PC Cursor.
? JAWSCursor() - Activates the JAWS Cursor.
? InvisibleCursor() - Activates the Invisible Cursor.
? BrailleCursor () - Activates the Braille cursor.
? RouteBrailleToPc () - Moves the Braille cursor to the PC  cursor.
? RouteBrailleToJAWS () - Moves the Braille cursor to the  JAWS cursor.
? RouteJAWSToBraille () - Moves the JAWS cursor to the  Braille cursor.
? RoutePCToBraille () - Moves the PC cursor to the Braille  cursor, if possible.
? RoutePCToJAWS() - Moves the PC Cursor to the JAWS Cursor,  if possible.
? RouteJAWSToPC() - Moves the JAWS Cursor to the PC Cursor.
? RouteJAWSToInvisible() - Moves the JAWS Cursor to the  Invisible Cursor.
? RouteInvisibleToPC() - Moves the Invisible Cursor to the  PC Cursor.
? RouteInvisibleToJAWS() - Moves the Invisible Cursor to  the JAWS Cursor.
? SaveCursor() - Saves the active cursor and its position.
? RestoreCursor() - Reactivates the saved cursor and  restores it to its original position.

The next group of functions is used to move the cursor.

? PriorCharacter() - Moves the active cursor to the prior  character.
? PriorWord() - Moves the active cursor to the prior word.
? PriorLine() - Moves the active cursor to the prior line.
? NextCharacter() - Moves the active cursor to the next  character.
? NextWord() - Moves the active cursor to the next word.
? NextLine() - Moves the active cursor to the next line.
? NextParagraph - Moves the active cursor to the beginning  of the next paragraph.  If the PC cursor is active, and  the next paragraph is not already visible, then text in  the window will automatically scroll to bring it into  view.
? NextSentence - Moves the active cursor to the beginning  of the next sentence.  If the PC cursor is active, and  the next sentence is not already visible, then text in  the window will automatically scroll to bring it into  view.
? PriorParagraph - Moves the active cursor to the beginning  of the prior paragraph.  If the PC cursor is active, and  the prior paragraph is not already visible, then text in

the window will automatically scroll to bring it into  view.
?	PriorSentence - Moves the active cursor to the beginning  of the prior sentence.  If the PC cursor is active, and  the prior sentence is not already visible, then text in  the window will automatically scroll to bring it into  view.
?	JAWSHome() - Moves the active cursor to the beginning of  the line.
?	JAWSEnd() - Moves the active cursor to the end of the  line.
?	JAWSPageUp() - Moves the active cursor to the top of the  window.
?	JAWSPageDown() - Moves the active cursor to the bottom of  the window.
?	MoveTo - Moves the JAWS or Invisible cursor to the screen  coordinates specified by the user.
?	MoveToControl - moves the active cursor to a specific  control within a window.  If the PC cursor is on when  this function is called, the JAWS cursor is turned on  automatically.  Otherwise the active cursor is used.
?	MoveToFrame - Moves the active cursor to the top left  corner of the specified Frame.  If the PC cursor is  active when this function is used, then the JAWS cursor  is activated and it is moved to the new position,  otherwise the active cursor is moved.
?	MoveToGraphic - Moves the JAWS cursor, Invisible cursor,  or Braille cursor in a specific direction to find a  graphic symbol in the active window.
?	MoveToWindow - Moves the active cursor to the specified  window.  If the window contains text, then the cursor is  positioned on the first character.  Otherwise, it is  positioned at the window's center.  If the PC cursor is  active when this function is used, then the JAWS cursor is activated and it is moved to the new position.
?	PriorChunk - Moves the active cursor to the prior chunk  of text.  A chunk of text is a section or block of text  that is written to the screen at one time
?	NextChunk - Moves the active cursor to the next chunk of  text.  A chunk of text is a section or block of text that  is written to the screen at one time
?	DownCell - When inside a table or spreadsheet, moves the  active cursor to the cell in the same column but the next  Row.
?	UpCell - When inside a table or spreadsheet, moves the  active cursor to the cell in the same column but the  previous Row.
?	PriorCell - When inside a table or spreadsheet, moves the  active cursor to the cell in the same row but the  previous column.

? NextCell - When inside a table or spreadsheet, moves the active cursor to the cell in the same row but the next column.

Note that If your purpose is to read something located on the screen, you would usually follow a move function with one of the say functions. An example of this is shown below.

JAWSPageDown () ;moves to the bottom of the window SayLine () ;reads the line moved to

## 5.3.3 Creating Reading Scripts

Now, let's look at a practical use of what we have learned so far. Before we look at a real script, here are the steps again:

1. First, save the status of your current cursor, if necessary. Then activate the cursor of your choice.
2. Now move the active cursor to a specific position in the window.
3. Select the appropriate reading function.
4. Restore your original cursor if you used the SaveCursor function at the start of the script.

The following are two variations of a script which can be used to read the bottom line of a window. There is a comment at the end of each line that describes the function. (We previously mentioned comments briefly in the section on include statements, and will discuss them more fully below.) In the first script we will switch to the JAWS cursor, read the bottom line and then leave the JAWS cursor active at the bottom of the window.

```
Script ReadBottomLine() ; Begins the script and assigns it the name, ReadBottomLine
JAWSCursor(); Activate the JAWS cursor RouteJAWSToPC() ; Move the JAWS cursor to the PC
cursor to assure it is in the window we're working
                    ;in and not in some other window elsewhere on the screen.
JAWSPageDown() ; Move the JAWS cursor to the bottom of the window
JAWSHome() ; Move the JAWS cursor to the beginning of the bottom line
SayLine() ; Speak the bottom line of the window EndScript ; End of the script
```

The next variation adds the save and restore cursor functions and uses the Invisible cursor.  This script will  read the status line and then restore the original cursor.

Script ReadBottomLine() ; Begins the script and assigns it  the name, ReadBottomLine SaveCursor() ; Saves the current cursor and its position  InvisibleCursor() ; Activates the Invisible cursor  RouteInvisibleToPC() ; Move the Invisible cursor to the PC  cursor so it is in the correct window.  JAWSPageDown() ; Move the Invisible cursor to the bottom of  the window JAWSHome() ; Move the Invisible cursor to the beginning of  the bottom line SayLine() ; Speak the bottom line of the window  RestoreCursor() ; Reset the cursor and position we saved  EndScript ; End of the script

See how it works?  You could use this script to read the  status line in Microsoft Word.  The status line tells you  the page number, line number, column number and other  information like that.  If you only wanted the page number,  you would substitute a SayWord() function followed by a NextWord () and another SayWord () for the SayLine()  function.  Then you would just hear the words: page and the  number.

There is an important subtlety about the SaveCursor and  RestoreCursor functions which needs to be understood.  These  functions save and then restore the cursor type of whatever  cursor is active when the SaveCursor function is used.  In  addition, if the JAWS or Invisible cursor is the active  cursor when the SaveCursor function is used, the cursor  position is also saved and then restored by the  RestoreCursor function.  Cursors which are activated and  moved after the SaveCursor function is used will not be  restored to their original types or positions.  Thus, in the above example, the position of the Invisible cursor would  not be restored unless it happened to be the active cursor  at the time the SaveCursor function is called.  However, it  is possible to use multiple SaveCursor and RestoreCursor  statements in one script.  These statements will "stack" in  very much the same way that the default and application  scripts stack.  This means that, if you were to use a second  SaveCursor statement, and you were to have a different  cursor active than you did when you used the first  SaveCursor statement, it is the position of the second active cursor which would be saved when the second  SaveCursor is executed.  Furthermore, the RestoreCursor  statements will undo the SaveCursor statements in the  reverse order.  Thus, if you perform two SaveCursor statements and then perform a RestoreCursor, it is the  second SaveCursor that will be reversed. To undo the first  SaveCursor, you would have to perform a second  RestoreCursor.  The following example illustrates the  stacking of SaveCursor function calls:

Script ReadBottomLine () ; Begins the script and assigns it  the name, ReadBottomLine
SaveCursor () ; We don't know what cursor was active, but  Saves the current cursor and its position  InvisibleCursor () ; Activates the invisible cursor  SaveCursor () ; Saves the invisible cursor and its position  RouteInvisibleToPC () ; Move the Invisible cursor to the PC  cursor so it is in the correct window.  JAWSPageDown () ; Move the invisible cursor to the bottom of  the window
JAWSHome () ; Move the invisible cursor to the beginning of  the bottom line
SayLine () ; Speak the bottom line of the window  RestoreCursor () ; Reset the invisible cursor and position  we saved
RestoreCursor () ; Reset the users original cursor and  position we saved
EndScript ; End of the script

Finally, if you use a SaveCursor ()in a script, JAWS  automatically does a RestoreCursor () at the end of the  script, even if you forget to put one in.  JAWS assumes that  you want to restore the cursor if you've put a SaveCursor ()  in the script.  If you've used multiple SaveCursor statements, JAWS will automatically undo all of them in  reverse order when the script is finished, even if you don't  remember to put in all of the RestoreCursor statements.

## 5.4 Putting It All Together

Now let's look at the process of creating a script,  assigning it to a key, and compiling the script file so we  can use it.  Here's the plan.  We will create a script for  the Script Manager that reads the current line number.  We'll need the Script Manager for this task.  Remember that  we used this editor in an earlier section (See Chapter  3.)when we created our first scripts.  This time, we'll let  the Script Manager do more of the work for us.  Follow these  steps exactly:

1. Start the Script Manager by selecting it from the JAWS  Utilities menu.
2. Press INSERT+Q to verify the Script Manager settings are  loaded into the stack above the default script file and

that the program name for the script editor is JSCRIPT.EXE. This means, of course, that the script file currently controlling JAWS is JSCRIPT.JSS. To load this file so you can edit it, drop down the file menu with ALT+F and arrow down to Open and press ENTER. Type in the file name, JSCRIPT.JSS, and press ENTER. Finally, press CTRL+END to take the caret to the bottom of the script file, the best place to add a new script.

3. Before we actually write our script, let's explore the window to see what we want our script to read. Script Manager has a status line at the bottom that contains a help message, the current line number and the total number of lines in the file. We want to read only the current line number with our script. Just to make sure we know what we want our script to do, let's do it manually first.
4. Start by activating the JAWS cursor.
5. Then press PAGE DOWN to move to the bottom line followed by HOME to move to the beginning of the bottom line.
6. Now use INSERT+NUM PAD 5 to say the current word and you should hear the word "For." Use the next word key (INSERT+NUM PAD 6) to read across the line. You now know that the fifth word on this line is the current line number. We now know what our script has to do. It must activate a cursor, position it at the proper place, and then read the word. OK, reactivate the PC cursor and we'll get started.
7. Select New Script from the Script menu, and the New Script dialog will appear. Now we have to name our script and write its documentation.
8. When the New Script dialog appears, the cursor is in the Script Name field. Type in SayLineNumber. Be sure to cap the S, L, and N and don't put in any spaces.
9. Press TAB to move to the Can be attached to key checkbox and press SPACEBAR to check it. We want to attach our script to a keystroke later.
10. Press TAB to move to the Synopsis field and type, "Read current line number."
11. Press TAB to move to the Description field and type, "The line that contains the PC cursor."
12. Press TAB to move to the Category field and type, "Say," because this script will say something.
13. Press TAB to move to the Assign To field and press CTRL+SHIFT+L to assign this keystroke to SayLineNumber.
14. We won't need any other fields filled in for our script, so press ENTER to close the New Script dialog and insert our blank script into the editing area.
15. We now have a script begin statement with the script name and an EndScript statement. There are a few blank lines in between for us to put in our script statements and the

cursor is positioned on one of the blank lines. Use the UP ARROW key until you reach the first blank line of the script.
16. This time we will not type our functions as we did when creating our first script. We will use the Insert Function dialog of the Script Manager.
17. Choose Insert Function Call from the Script menu, or use the hot key combination, CTRL+I. The Insert Function dialog box appears and the cursor is placed in the function name edit field. There is also a list of functions displayed beneath the function name field. You could tab down to the list and start arrowing through the functions to find the one you need, but that would not be very efficient as there are more than two hundred of them. JAWS has a better way, but it does require that you have some idea of the function name.
18. Just like in our example earlier, we need to start with saving the cursor and position. The function is SaveCursor(), but we only need to press the S as the Insert Function tool starts searching as soon as you start typing. SaveCursor() is the first function that starts with an S. Notice that as focus moved to SaveCursor(), a help message was read that explained how this function is used.
19. Press enter and this function is placed into our script. Easy isn't it? Now let's activate the Invisible cursor. We use the Invisible cursor here as we just want to read the line number and return. No sense in dragging the mouse with us, so we won't use the JAWS cursor.
20. Press ENTER to insert a new line and then press CTRL + I again.
21. The function we need this time is InvisibleCursor(). We'll have to type two characters this time as InvisibleCursor() is not the first function that begins with an I. Type slowly so that you can hear each function as it is highlighted. Once you hear InvisibleCursor(), press ENTER to place it in our script.
22. Use the same procedure to add the rest of our functions in this order: RouteInvisibleToPC (), JAWSPageDown(), JAWSHome(), NextWord() five times, SayWord() and RestoreCursor().
23. Press CTRL + S to compile the script file. If you made no mistakes, you should hear the "compile complete" message. If not, repeat the previous steps until you can compile without an error. If you do get an error, the Script Manager will position the caret near where it thinks the error is so you will have a better chance of finding it.
24. Now test your work by pressing CTRL+SHIFT+L to hear the current line number.

Try a few variations like reading the whole line instead of the word. Then add a message that says "line number" before it actually speaks the number. When you feel comfortable with this process, either continue with the next part, "Creating Advanced Script Files, or jump in and write some scripts you been wanting for some time.

Homework Assignment #2

Here is your second homework assignment. As you probably know, most windows have three states, minimized, restored, and maximized. In the upper right hand corner of most windows, there are three buttons. The first is minimize, and the last is close. The middle one can either be restore or maximize, depending on the current window state. By moving to and reading the middle button, you can determine the window state. If it says "restore symbol", then the window is maximized. If it says "maximize symbol", then the window is restored. Your assignment is to add a script to the default script file, DEFAULT.JSS, which will read this button. Obviously, to test your script properly, make sure you are in a window that has these buttons. The Script Manager window will serve quite nicely. After you have written this script, you will be able to determine the state of the current window with one keystroke. For this assignment, we want you to use the JAWS cursor to do the reading, and you should make sure that the JAWS cursor is returned to its original position by the time the script ends. Assume that the PC cursor is active when you start the script, and make sure it is active when the script is done. You can load the default script file by choosing the Script Manager from the JAWS window Utility menu and then loading DEFAULT.JSS. An alternative way of doing this is to press CTRL+SHIFT+0 on the number row. This will automatically open the Script Manager and load the DEFAULT.JSS file. Add your script to the bottom of this file. One possible answer can be seen in Appendix A (See Chapter 14.1.), but don't look at it until you've had a try at writing this script.

Extra credit - If you're really feeling adventuresome, try modifying the above script to be a little more sophisticated. Instead of just reading the Maximize or Restore symbols, use If-Then, ElIf, Else, GetWord, and Say functions to decide what the state of the window is and what message to speak. If the window is showing a Restore symbol, have the script say "Maximized." If a Maximize symbol is showing, have the script say "Restored." If it doesn't find either of these two symbols, have it say

"Couldn't find the symbol." Hint - a labeled graphic such as a Restore or Maximize symbol can be treated as text, and its label can be read and manipulated with the same functions that are used for pure text. One possible answer to this problem is also in Appendix A. (See Chapter 14.1.)

## Homework Assignment #3

And just for fun, here is your third homework assignment. Windows comes with a sound recording program called Sound Recorder. This program can usually be found in the Multimedia folder of the Accessories folder of the Start Menu. Open it, and explore around the Sound Recorder window with your JAWS cursor if you are not familiar with this program. You will find that there are several buttons for play, stop, record, etc. You will also find that the fourth line from the top has a counter which tells you how many seconds have played and how many seconds long the file is. Load any file on your computer with a .WAV extension using the Open Dialog of the File menu. (You can find many .WAV files in the C:\WINDOWS\MEDIA folder.) Click the play button to start playing the file, then click the stop button before the file finishes playing. (If you prefer to use the keyboard to do this, the hot keys for these two functions are CTRL+P and CTRL+S, respectively.) If you look at the elapsed time counter again, you will see that there are now numbers indicating how much time is elapsed out of the total time. We want you to write a script for this application's JSS file (SNDREC32.JSS) which will read this counter for you with one keystroke. The announcement should be in the form "Current time is X seconds out of Y seconds". The X and Y values are to be obtained from the Sound Recorder display, and the rest of the announcement should be created from Say statements. Use the Invisible cursor to do the reading, and return the starting cursor to the original state when done. One possible solution to this problem is shown in Appendix A. (See Chapter 14.2.)

# Part II: Creating Advanced Script Files

# 6 Windows Program Structure

You've probably heard people use terms like window class and window handle in discussions about how Windows programs are structured. What do terms like this mean? In this section, we'll explain window hierarchy, identification, and structure so you can make use of the script functions that query and compare these items.

The first question is what is a window? Well, that would seem to be simple. Programs run inside of windows. Even JAWS has a window. That's true. The JAWS window is a window, but it might surprise you to know that every entry field, button, and other control is also a window. At least it should be. Like all other rules in computer programs, it is not always followed. A programmer may choose to create controls, such as entry fields, without making them actual windows. Programs like this are the most difficult to make accessible because there's no simple way to identify and speak the various controls. These programs need script writing even more because the scripts in the default files will not speak these windows properly. For the moment, we'll assume that the programmer assigns each control to a window. Sounds complicated doesn't it? Let's see if we can sort it out.

## 6.1 Hierarchy

Most people who operate in the Windows environment never really understand what the word "window" truly means with respect to this environment. Of course they realize that the big rectangular thing that pops up on the screen when they open a new application is a window, and they probably also think of the text area of their word processor as a window, but there is much more to it than that. In most applications, every dialog box, menu, button, edit field, listbox, etc., is actually a separate window with one or more identification numbers that can be used to refer to it. These scores of windows that may be present on your screen at any one time are not randomly dropped there, they are interrelated with a special hierarchy that allows the Windows operating system and the programmers who work within it to keep track of them all. It will all make more sense if we look at the hierarchy of windows in the system. The metaphor most often used is one of parents and children. Here's how it works. Your computer desktop is the parent of all application windows. That makes the JAWS Window, your word processor's window and the Internet browser's window all children of the desktop. These windows are all one level down from the desktop, so they are all children of the desktop at the same logical level. This makes them all peers of each other, and, when you later encounter terms such as PriorWindow and NextWindow, the reference is to moving among peer windows at the same level. OK, now let's go to the next level. The word processor has several child windows: the text area, the menu bar, the toolbar and the status line.  These are all children of the word processor  window and are, themselves, all children at the same logical  level.  Are you beginning to get it?  When you ask to open a  file in the word processor, a child window appears so that  you can look for the file.  It's often referred to as a  dialog box, but in the grand scheme of things, it is a child  window to the word processor, its parent.  Now think of the  Open File dialog box as a parent and then you find that the  File Name field is a child window of the dialog.  So are all  the other fields within the dialog.  These are all children  of the parent dialog and are all at the same logical level.  Each window is a child to the window one level up that  spawned or generated it, and each window is a parent to the  windows one level down that it spawns.

Just in case you're having a little trouble visualizing all  of this, let's try using an analogy.  Most people are  familiar with directory tree structure, either from working  in DOS or Windows Explorer.  On your disk drive, say your C  drive, C:\ is the parent of all other directories on your drive.  All of the subdirectories one level down from C:\  are children of that parent and are at the same logical  level.  Thus, if you have subdirectories named Eudora,  JAWS37, MyDocuments, Recycled, and Windows on your system,  they will all be one level down from C:\.  They are all direct children of C:\ and are all one logical level down.  For the purposes of directory tree structure and window  hierarchy, you can assume that being at the same logical  level means that all members were spawned or generated by  the same parent.  Remember that we said that the PriorWindow  and NextWindow command could be used to move to windows at  the same logical level?  Well, in a directory tree, this  would be analogous to moving to the prior subdirectory and next subdirectory of a directory tree.  If we open the  JAWS37 subdirectory, we will see four new subdirectories,  Help, Manuals, Settings, and Tecnotes.  These four  subdirectories are all one level down from JAWS37, are  children of JAWS37, and are at the same logical level as  each other, two levels down from C:\.  This can, of course,  continue until we reach the lowest logical level of a  particular branch.  The parent/child structure of Windows  works much the same way as a directory structure.  Each  application is a branch of the tree, and child windows are spawned from it.  All windows one level down from the parent  application are children at the same logical level, one  level down from the parent.  Each of these children can  spawn children of their own, and these children are three  levels down from the parent application.  Try moving up and  down through the various logical levels of different  branches of your directory tree in Windows Explorer, and you should get a better feeling for this parent/child hierarchy.

## 6.2 Identifying Windows

The application programmer assigns a number of identifiers to each window. These identifiers allow the window in question to be referred to unambiguously. For instance, each has a Window Class, a Type, a Type Code, a Subtype Code, and a Control ID. The Window Class and Type Code tell us whether it is an edit field, a button, or some other type of window. The Subtype Code can give more specific information about the window, differentiating, for example, between a regular button and a radiobutton. Thus, the window class, Type, Type Code, and Subtype Code allow us to categorize the window. Listed below are more precise definitions of these four categories and the JAWS built-in functions that are used during script writing to obtain them.

### 6.2.1 Window Classes, Types, Type Codes, and Subtype Codes

? Window Class - This is a category which denotes some information about what a window is and does. For example, a Window Class can be an edit field, a listbox, or a button. This is already providing some valuable information about the window since we know that a listbox is something entirely different and will have quite distinct reading requirements from a button. The problem is that Window Class does not always provide information which is sufficiently detailed for our purposes. For example, checking the Window Class will return the same information for a button, radiobutton, and checkbox, all of which have the Window Class of button. Obviously, we are going to need more detailed information in some instances. The other window categories listed below can be used to provide this information. Furthermore, some programmers use custom window classes which have strange and unrecognizable names that are not understood by JAWS. We are going to need some method of telling JAWS how to deal with these custom window classes, and this is discussed below. The Window Class is obtained with the function, GetWindowClass.

? Window Type - As with Window Class, Window Type returns a string or name which is descriptive of the window. This category can sometimes give more specific information about certain windows. However, asking for the Window Type will Return the same information about a window as asking for the Window Class if no more specific information is available.  For example, the  Window Class and Window Type of an edit field will be  exactly the same.  However, if more specific  information is available, the Window Type can be used  to sort this out.  Thus, whereas the Window Class of a  button, radiobutton, and checkbox is always "button",  the Window Types of these three windows will be  reported as "button", "radiobutton", and "checkbox".  This more specific information will allow us to tell  JAWS what to do for each of these types.  If the  programmer has used custom window classes, then asking  for the Window Type will return a string that says that  the Window Type is unknown.  However, when a custom  window type is reclassified to a known window class  (see below), the Window Type will return the name of  the reclassified window and, thus, tell us what the  window does.  One problem with the Window Type is that  the string returned by this function is always in English and, thus, is not very useful for those using  JAWS in other languages.  For this reason, the Window  Type Code was developed.  The Window Type is obtained  with the function, GetWindowType.

?	Window Type Code - Asking for the Window Type Code  returns a number instead of a string.  This number is  translated into a recognizable string by constant  definitions which can be found in the file HJCONST.JSH.  For example, in this file you will see lines such as  WT_Button =1, WT_Edit = 3, WT_ListBox = 4, and  WT_ScrollBar = 5.  The WT prefix of the constant tells you that this is a constant that refers to a Window  Type.  While the Window Type Code does not end up  providing any more information than the Window Type, it  does have the advantage of being language independent.  Since the same numbers are always returned for a given  type of window, the numbers can be equated to a string  in any language desired.  The Window Type Code is  obtained with the function, GetWindowTypeCode.

?	Window Subtype Code - As with the Window Type Code, the  Window Subtype Code also returns an integer.  The  Window Subtype Codes are translated into strings using  the same constant definitions as for Window Type Code.  The only difference is that the Window Subtype Code  will provide more detailed information, if available.  If not, both functions will return the same  information.  For example, a RadioButton will return a  Type Code of 1 (WT_Button) but a Subtype Code of 19  (WT_RadioButton).  The Start button located on the task  bar will also return a Type Code of 1 (WT_Button) but  will return a Subtype Code of 32 (WT_StartButton).  Similarly, GetWindowTypeCode will return WT_TABCONTROL for the Task Bar whereas GetWindowSubtypeCode will return WT_TASKBAR. You will want to use GetWindowTypeCode when you don't wish to have the more specific information which is returned by GetWindowSubtypeCode. Window Subtype Codes are also language independent. They can be obtained with the function, GetWindowSubtypeCode.

The reason that these window categories are important to users of JAWS is that JAWS will behave differently, depending on what category of window has the focus. For example, if the focus is on an edit field, JAWS tracks the caret or insertion point. If you hit the DOWN ARROW key, JAWS will speak the line of text to which your caret moves. On the other hand, if you are in a list box or menu, JAWS tracks the highlight or lightbar as the arrow keys are used to move it up and down the list. JAWS knows to do these things because of the scripts that are associated with the up and down arrow keys in the DEFAULT.JSS script file. These scripts contain logic that decides what to do based upon the category of the window that has the focus when you use one of these keys. If you examine the SayPriorLine and SayNextLine scripts in the DEFAULT.JSS file, you will see that decisions about what to speak are made based upon the Type Codes and subtype Codes of the windows that are current when the script is invoked.

As with Window Types, Type Codes, and Subtype Codes, there is a set of standard window classes that is used by most programmers, and the JAWS default scripts contain the necessary logic to allow proper speaking of these standard classes most of the time. However, Window Classes are not always as standard as these other categories. If you have ever needed to use the Window Classes dialog in the Configuration Manager to reassign a custom or unknown Window Class, you know that sometimes a programmer will create custom window classes that we need to inform JAWS about. Since these custom classes are, by definition, non-standard, JAWS has no idea what to do with them. Therefore, we have to use the Configuration Manager to equate the unknown class to one of the standard classes so JAWS will know how to behave when it encounters it. This process is called reclassifying or reassigning the Window Class. If you have never used this procedure, use either the key combination INSERT+7 or the key combination INSERT+F2 followed by W and ENTER to invoke the Window Class reassignment dialog, and explore the various controls to become familiar with it. Once you have reclassified a window to equate it with a standard class, it should speak properly. Sometimes you may have to try more than one class to figure out which one works best.

## 6.2.2 Control ID's and Window Handles

The Control ID is an arbitrary number which serves as an identifier that the programmer uses to mark each window in his program. It is obtained with the function, GetControlID. The Control ID, unlike the Window Class, has no special meaning, but it can be used to refer to a particular window. Ideally, no two windows in a program should have the same Control ID, but they sometimes do anyway. If you know how to access the Control ID, and, if it is unique within the application you are writing scripts for, then you can tell JAWS to do certain things if it finds that you are focused on the window with the particular Control ID you have identified. One example where this is extremely useful is in applications where the buttons are labeled with bit map drawings instead of text labels. Normally, JAWS will speak the text label of a button when you TAB to it, but if the label is a drawing, there is no name to be spoken. By determining the Control ID numbers of the various bit map buttons, one can write a script that will speak the name of a button when the focus moves to a button with a particular Control ID.

Of course, there is no way to insure that two windows in different programs won't have the same Control ID. Therefore, to prevent confusion, the system assigns a unique identifier, called a window handle, to every window when it is accessed during a given session. Window handles are numbers, but they are not unchanging values like a Control ID. As long as the handle is actively assigned to a particular window, it can be used to refer to and identify that window. Once the handle is assigned, it remains active as long as the window exists. However, when a window is destroyed, it relinquishes its handle. If the window is reinvoked later, it will probably have a different handle. Since the system assigns these handles as windows are used, the handle assigned to a window on this day may not be the same as it is tomorrow. A good analogy to the window handle would be the numbered ticket dispenser used in many bakeries and delicatessens. When you walk in you take a number from the dispenser, and this number becomes your handle. Today your number might be 31, and when the server at the counter calls 31, you know it's your turn. If you go in again the next day, your number will probably be different. It's still your handle, but it's different from what it was the prior day. The point is, it's an identification system that tells the server who's next in line.

Use the window' handle to identify it among all of the other windows in the system at any given time. The window handle can be obtained using the built-in functions, GetCurrentWindow () and GetFocus (). One way to do this, for example, would be to use the GetCurrentWindow () function as a parameter for the SayInteger () function. Thus, SayInteger (GetCurrentWindow ()) would speak the window handle of the window containing the active cursor. By the way, this is another example of the use of nested functions which was mentioned earlier.

There are some built-in default scripts that you can use to query these and other types of information from an application. We'll teach you how to use them in "Exploring the Application With the Utility Functions". (See Chapter 8.1.) For now, try the following. Press CTRL+INSERT+F1. You should hear the Control ID, Window Class, and Window Handle announced for the window you are currently in. Move around to different windows and different applications to get some idea of the kind of information this keystroke combination can supply.

# 7 Building Blocks of the Script Language

We construct script files from building blocks that make up the JAWS script language. Think of this part of the book as learning a language because you will be doing exactly that. Just as our written language has parts of speech that make sentences that make paragraphs, so also does the JAWS script language have statements that make up scripts that make up script files. First, we'll learn how an individual script is assembled, and then we'll learn all the different kinds of statements in this language. In the next part, "Script Writing Techniques," we'll learn how to put complete script files together and how to compile and use them.

## 7.1 The Script

As we've said before, a script is a small computer program that controls JAWS activities. Scripts can be attached to a keystroke. They do not have to be, but any script can be. A script must have the following pieces in this exact order:

1. First is the script begin keyword. This is simply the word "Script" followed by a space.
2. On the same line following the word "Script" is the script name. This is usually several words concatenated together that describe the action of this script, such as, CloseDocumentWindow. Note that each of the concatenated words is initial capped, so JAWS can read

the name correctly. () follows the name to complete the script begin statement. All of this is automatically entered by Script Manager when you create a script.
3. Next are all of the local variable declarations followed by all of the functions, arithmetic operations, and flow control statements that you need in sequential order. This could be as simple as just saying a string as we did in Our First Script or a complex operation.
4. The script end statement, EndScript, is always the last line of a script. This is automatically entered by the Script Manager when you create a script.

The Script Manager will create the basic script structure correctly when you choose New Script from the Script menu or use the accelerator key combination, CTRL+E. You should always create a script in this fashion as the Script Manager will also prompt you for the proper script documentation and let you assign a keystroke.

## 7.2 The Function

The biggest difference between scripts and functions is that a function cannot be attached to a keystroke. It will either be called from a script or another function or be triggered by a system event. It is important to remind you about some jargon we discussed earlier. This word function has several slightly different meanings, and it's important to keep them in mind for the following discussion. We are about to talk about functions that can be created by you, and we mean by this a script you write that is not attached to a keystroke. This is a user-defined function, meaning it is a function written by you, the user. There are also functions already present in script files you may be modifying, and these are functions that were written by Henter-Joyce's script writers. These are also user-defined functions, but they were defined by other users, not you. Finally, there are the built-in functions that are the building blocks of the JAWS language. You, the user, cannot modify the built-in functions as you can your own functions or the ones written by other people. You can use any of these three types of functions as building blocks when writing scripts or new functions, but the built-in functions are hard coded into JAWS, and so the following discussion does not apply to them. (Built-in functions will be discussed later. See Chapter 7.3.5.) A non-built-in function must have the following pieces in this exact order:

1. The return type, either handle, Int, Object, String, or Void.
2. The function begin keyword. This is simply the word

"Function" followed by a space.

3. On the same line following the word "Function", the function name. This is usually several words concatenated together that describe the action of this function, such as, SayFocusedWindow. Note that each of the concatenated words is initial capped. That way JAWS can speak the name as if it were separate words. The name is followed by () to complete the function begin statement. Note that all functions should have return type designations such as those listed under item 1 before the function begin keyword. These return types were mentioned earlier (See Chapter 4.4.1.) and describe what type of information is returned to the calling script when the function finishes running. Of course, a function can be designed simply to be called and complete its task without sending any specific information back to the calling script or function. In this case, the return is designated as Void. On the other hand, in addition to completing any other tasks, the function can also send an integer, string, object, or handle value back to the calling script. If it does this, you should see the word Int, String, Object, or Handle before the word Function on the first line. Special function event names such as AutoStartEvent or NewTextEvent tell you that a function runs automatically when certain things happen in an application. These functions are not normally called by other scripts or functions. There can also be parameters in between the ( and ) which are used to pass information to the function which it will need to perform its purpose. We'll discuss all of these in more detail later. (See Chapter 7.5.)

4. Then you have all of the local variable declarations followed by all of the functions, arithmetic operations, and flow control statements that are needed in sequential order. This could be as simple as just saying a string with one function as we did in Our First Script or a complicated operation.

5. The function end statement, EndFunction, always the last line of a function.

The Script Manager will create the basic function structure correctly when you choose New Script from the Script menu. You should always create a function in this fashion as Script Manager will also prompt you for the proper function documentation and let you assign parameters and returns.

Let's also remember one other piece of jargon. The term "calling a function" means placing that function's name into a script or function to be executed. You can call either a user-defined function or a built-in function in this manner.

In many cases, rules and procedures that apply to functions  also apply to scripts.  For example, variables are declared  the same way in both and functions are called the same way  in both.  There are also many differences.  If we are  describing a design facet that applies only to a script or  only to a function, we will be careful to point that out.  Otherwise, the rule applies equally to both.

## 7.3 Types of Statements

The following is a list of the types of statements that make  up scripts.  Each statement is like a part of speech in the  JAWS script language, has a specific job to perform, and has  certain rules associated with its use.

### 7.3.1 Comments

Comments are lines of text in your script file that are not  executable statements.  They are usually used to inform  those reading the source file how the statements work.  Comments may be placed on a line by themselves or at the end  of a line.  Begin a comment with a semicolon.  Everything  else on the line after it is treated as a comment.  If you  need multiple lines for comments, start each line with a  semicolon.

### 7.3.2 Includes

The include statement tells the compiler to include the  contents of another file along with this file, just as if  that information had been typed into the main file at the  point of the include statement.  Hence, all of the  information in the included file is accessible by the main  file from the point of the include to the end of the main  file.  This means that all included files will be compiled  along with the main file into one, large JSB file.  Include  files are separate files with common items that may be  usable by a number of script files or which you may wish to  keep separate for reasons of clarity or convenience.  There  are two types of include files:

? Header Files are designated by the extension JSH and  contain either variable or constant declarations.
? Message Files are designated with the extension JSM and  contain message statements with their assigned numbers.

In versions of JAWS before 3.0, it was sometimes necessary  to break macro files into pieces to avoid exceeding the  maximum size allowed.  That is not a concern with script files as they can be as large as you like.

The include statement has the name of the included file in quotes as follows:
Include "HJGLOBAL.JSH"
If you leave the quotation marks out, the syntax will be incorrect and you will get an error message when you attempt to compile. It is also assumed that the file to be included is contained within the language (e.g., ENU) subdirectory of the \SETTINGS subdirectory of your JAWS directory.

The following is a list of the standard include files provided with JAWS and included in DEFAULT.JSS:

? HJGLOBAL.JSH - Contains the default JAWS global variables.
? HJCONST.JSH - Contains the default JAWS constants.
? DEFAULT.JSM - Contains the standard message statements referred to by DEFAULT.JSS.

As we covered previously, remember that a message file is a special include file, which assigns message numbers to common messages. This way the script writer can use the message number in place of the message and JAWS will look up the actual text for the message in the message file. Actually, the message number is just a constant which has the actual text message assigned to it. All of the script files shipped with JAWS have their own separate message files. Use the same name as your script file followed by .JSM as an extension when you create your own message file. Then include it in your script file. Whereas the DEFAULT.JSM file contains messages useful only to DEFAULT.JSS, HJGLOBAL.JSH and HJCONST.JSH are used in any script file which refers to the JAWS global variables and constants.

Homework Assignment #4 -

Pretend you are starting to build a script file for an application called JAWSWINS.EXE. You want to use include statements for a header file of constants and variables, a message file for messages, and the two JAWS files, HJCONST.JSH and HJGLOBAL.JSH. Write the include statements required to do this. The answer can be found in Appendix A. (See Chapter 14.3.)

## 7.3.3 Variables

A variable, as we discussed earlier (See Chapter 1.3.) and as its name implies, is an entity that holds a value that can change during code execution.  (The term execution is  used by programmers to describe the running of code, i.e., a  computer program such as a script.  We use it here just to  introduce you to the term.)  Each variable is of a specific  type.  The contents of a variable are stored in memory by  the JAWS program and can be used by your script code  whenever you need it.  Variables must be "declared" or  defined to have a name and a type before they can be used.  Each variable must have a distinct name to distinguish it  from other functions and names used in the code.  Be careful  not to use names that are already assigned to functions,  global variables and constants as this will cause compile or  execution problems.

The following is a description of the types of variables  allowed in the JAWS script language:

?	Integer - A variable designed to hold an integer value.  (An integer is a whole number without a decimal point.)  It is declared like this:

Int MyIntegerVariable

Note that the type (Int) is followed by the distinct  name assigned to this variable (MyIntegerVariable).

An integer variable contains a whole number.  We use a  variable so that its value can be changed by actions that  take place while our script file is active and so that we  can refer to the number by name in our scripts.

The following rules apply to the use of an integer variable:

1.	An integer variable must be declared before it can be  used.
2.	Its default value, that is to say, the value it has  before any value is assigned to it, is always 0.

?	String - Holds a string of characters.  A string is a  group of characters including letters, numbers,  punctuation marks, and spaces.  It is declared like this:

String MyStringVariable

We use a variable to represent our string because the string  it contains may be changed by actions that take place while  our script file is active and so that we can refer to the  current string by name in our scripts.

The following rules apply to the use of a string variable:

1. A string variable must be declared before it can be used.
2. Its default is always "null" or no characters.

When you are assigning a value to a string variable, you must put the string between quotes. For example, the following syntax is correct: Let StringVariable = "Hello". Note that in versions of JAWS prior to version 3, you had to insert a space in between the quote marks when you were checking for a null value. That has been changed. Null is indicated by to quotation marks with no space between them.

? Handle - Holds a window handle. A window handle is a unique number assigned by the system to each window in a currently running program. It is declared like this:

Handle MyHandleVariable

You probably remember from our earlier discussion of window identifiers (See Chapter 6.) that the window handle is assigned by the system and is unique to a particular window. While a handle is also a whole number and can, therefore, be manipulated like other integer variables, the use of a handle variable is reserved solely for the identification of a window handle. Most of the time we use it to identify a particular window from which we want information.

The following rules apply to the use of a handle variable:

1. A handle variable must be declared before it can be used.
2. Its default value is always 0.

? Object - A variable designed to hold an object. An object, for the purposes of the JAWS script language, refers to the types of objects present in certain Microsoft applications such as the Office 97 suite. (See Chapter 7.3.5.9.) It is declared like this:

Object MyObjectVariable

An object variable is used to hold an object which has been returned to a JAWS script or function. It is employed to obtain certain types of information from some Microsoft applications which are normally very difficult or impossible to obtain simply by reading the screen. The following rules apply to the use of an object variable:

1. An object variable must be declared before it can be used.
2. It's default value is always null or empty.

Variables can be local which means they can only be used in the script where they are declared or global which means that they can be used in any script within the script file where they are defined. Global variables can be declared at the beginning of a script file with the group heading, Globals. Place this group right after the include statements. Global variables can also be declared in an include file as is done with the default script file and its include header file HJGLOBAL.JSH. Once it is included, the global variables declared in this file can be used in any of the default scripts. You should create a header file for variables that you intend to use in more than one script file. If you create a global variable that you only intend to use with one application, it's a good idea to declare that variable directly in the script file rather than the header file and to include the app name in the variable name. This will serve to tell the user that this is a global variable meant for use by this one application. The user should also understand that once a global variable has been created and assigned a value, that value remains in memory, even after the application that uses it has been closed. If the application is reopened during the same computer session, the global variables from that application's script file will have the same values they had just before the application was closed. The only way to clear a global variable is to unload and then reload JAWS.

Local variables are declared within a script in the first statement after the Script start statement. They have the heading, Var.

Here's how each class of variable looks in an example from a standard script file:

The following declaration statements are taken from WINWORD.JSS, the script file for Microsoft Word for Windows. Notice that most global variable names start with the word, global, so that they can easily be distinguished from a local variable. We'll go over naming conventions in more detail later in "Script Writing Techniques". (See Chapter 8.)

```
Globals
String GlobalWinwordVersion,
Int GlobalCurrentControl,
Handle GlobalRealWindow,
String GlobalRealWindowName,
Int WinWordFirstTime, ; to say auto start message only first time
```

Handle WinWordContextHandle, ; read word in context  Int WinwordFontCode ; when font buttons are pushed

There are two things to note about this list of global  variables.  First, each line except the last one has a comma  after the variable name.  Second, some of the lines have  comments after the variable name.  The comment is everything  which appears after the semicolon and can serve as a  reminder of the purpose of the variable.  Any commas must go  before the comment.

The following local variable declaration statements are  taken from the HotKeyHelp97 () function in WINWORD.JSS, the  script file for Microsoft Word for Windows.  As shown, the  variable declarations appear just after the function begin  statement.

```
Function HotKeyHelp97 ()  var
        handle WinHandle,
        Int verbosity
```

Here again, all lines except the last have a comma at the  end.

## 7.3.4 Constants

As we discussed earlier (See Chapter 1.3.), constants are a  way of using easily-remembered names to store hard-to-  remember strings of letters or numbers.  Think of a constant  as a number or group of letters or words that has been given  or assigned a new name.  This is done purely for mnemonic  reasons since it is easier to remember a name than a number  or long group of letters or words.  There are no  restrictions to how many constant names you can assign to a  number.  For example, in the standard JAWS constants file,  HJCONST.JSH, both of the constants True and On are assigned  the value one.  What that means is that you can write a script statement to check if a condition equals True, on, or  1.  They are all considered to be the same value once the  constant declarations have been made.  We strongly recommend  using constants because its more difficult to remember that  when something is true, its state is 1, but easy to  understand the difference between true and false.  Moreover,  when someone else is reading your script file, it's much  easier for them to understand what you were trying to do if  they see well-named constants instead of numbers.  Finally,  if you need to change the value of a number that is used in  several places, you can change all of them at once if you  have used a constant to represent that number by simply changing the value of the constant declaration.

Constants are declared at the beginning of a script file with the group heading, Const, as follows:

```
Const
        True = 1,
        False = 0,
        On = 1,
        Off = 0
```

Please note that all of these entries except the last has a comma at the end of the line. Script writers will sometimes define their own constants for a script file, but they often also use those that are already declared in HJCONST.JSH. If so, they must remember to include this header file. We'll go over the meanings of the constants in this file in the next section, "Script Writing Techniques."

## 7.3.5 Built-in Functions and Operators

There are several types of statements in the JAWS scripting language that can be used to construct scripts and user- defined functions. These are built-in functions, arithmetic operators, logical operators, and bitwise operators. There is also a special type of built-in function called a hook function which will be discussed separately.

### 7.3.5.1 Built-In Functions

The JAWS language provides over 350 pre-defined functions for use in scripts. Each function performs a specific series of activities. These are the building blocks or instructions from which we create our scripts. For instance, RouteJAWSToPC checks the position of the JAWS and PC cursors, makes the necessary calculations and moves the JAWS cursor to the same point as the PC cursor. All of that activity is transparent to you as all you need to know is that it moves the JAWS cursor to the PC cursor.

It would not be practical to discuss all the predefined functions here, so we'll be content to discuss some of the most important ones. Appendix C lists many of the most important functions and gives a brief description of each one. These descriptions are based on information that can be found in a file called BUILTIN.JSD which can be found in the \settings\enu subdirectory of your main JAWS directory, assuming you are using the U. S. English version of JAWS. These BUILTIN.JSD descriptions are similar to the descriptions you hear when viewing the functions list of the

Script Manager, but many of them have been enhanced with additional information. While all the JAWS functions are important, the ones shown in Appendix C were deemed to be most important and worthy of inclusion here. Obtaining a thorough understanding of these will put you well on your way to becoming a master script writer. These descriptions and the ones for the remaining functions can be reviewed at any time by invoking the insert function dialog in the Script Manager or by viewing the BUILTIN.JSD file.

We have also grouped many of the built-in functions by the type of task they perform. These groupings can be viewed in Appendix D. If you want to do a particular type of task but don't know the name of the function, you might try looking in the Appendix D categories instead of looking through the entire alphabetical list of Appendix C or BUILTIN.JSD.

Before studying Appendix C, there are a couple of concepts about some of the functions that we should review. The syntax of these built-in functions is very similar to that already discussed for user-defined functions, but let's take a moment to remind ourselves.

The syntax of a function is as follows:
FunctionName ([Parameters])

Some built-in functions require parameters. This is indicated by the word "parameter" in brackets above. A parameter is data that the function needs to have in order to do its job. If there is more than one parameter, a comma is used to separate each parameter within the parentheses. The format is exactly the same as was previously discussed for user-defined functions. When a function which requires more than one parameter is called, the data must be supplied in the same order stipulated by the function definition.

Some of the built-in functions provide "returns". Recall that a return is data that the function returns back to the script for further processing. If a function provides a return, that information is given in the BUILTIN.JSD file and in the function list documentation of the Script Manager.

7.3.5.2 The GetCurrentWindow and GetFocus Functions

There are two built-in functions that are so very important to script writing that they deserve separate attention here. These two functions are named GetCurrentWindow and GetFocus. What in the world do these do? We have talked quite a bit about window handles, but never have we explicitly told you how to get one or what to do with it if you do get it.  This  is where you're going to start learning how to do these two  things.  This will allow you to obtain detailed information  about a window (which, as we've already said, also means any  control) that is present in your application.

Shown below are the definitions for these two functions  which are taken from BUILTIN.JSD.

?  GetCurrentWindow - Determines the window handle for the  window that contains the active cursor.  In contrast, the  GetFocus function uses an analytic process to find the  window that currently has the focus regardless of which  cursor is active.
?  GetFocus - Obtains the window handle for the window that  has "the focus".  It always seeks to find the PC cursor  or highlighted item that has the focus.  It does not take  into account which cursor is active.  In contrast, the  GetCurrentWindow function is less sophisticated.  It simply obtains the handle for the window in which the  active cursor is located.

So what do these definitions mean?  First of all, they both  supply a window handle, so using them allows us to obtain  this information about a window in our application.  For  example, the statements Let MyHandle = GetCurrentWindow ()  and Let MyHandle = GetFocus () would set the handle variable  equal to the value of the handle of the window in question.  If you followed these by a statement such as SayInteger  (MyHandle), JAWS would speak the number or window handle of  the window of interest.  The difference between these two  functions is that GetFocus will obtain the window handle of  whatever window JAWS believes to be the focus.  This might be, for example, the currently highlighted button or edit  field in a dialog box.  GetCurrentWindow will check what  cursor is active and then get the handle of the window  containing that cursor.  So, if you activate the JAWS  cursor, for example, and move it away from the focus to  another control, GetCurrentWindow will get the window handle  of that new control while GetFocus will still get the handle  of the window that has focus.

Okay, so now you know how to obtain a window handle.  What  are you going to do with it?  It's not much use by itself  since it's just a number.  However, the GetFocus and GetCurrentWindow functions can be used as parameters by  other functions which do provide much more useful  information about the window of interest.  Using these other  functions along with GetFocus and GetCurrentWindow to tell  the other functions which window to obtain information about

, can provide very useful information.  Some examples are  shown below.

?          GetControlID (GetCurrentWindow ()) - Will provide the  Control ID of the window containing the active cursor.
?          GetFirstChild (GetFocus ()) - Obtains the handle of the  first child window of the window which has focus.  This  child handle could then be used to obtain information  about the child window.
?          GetParent (GetCurrentWindow ()) - Obtains the handle of  the parent that spawned the current child window.
?          SayControl (GetFocus ()) - Will speak the contents and  prompt of the control that currently has focus.
?          SayWindowTypeAndText (GetCurrentWindow ()) - Will speak a  variety of information about the window containing the  active cursor such as the window title, if any, the  window contents, the window type, and other information.

Thus, you now have an understanding of how to use some JAWS  functions to provide handles that can be used by other JAWS  functions to obtain information which will be needed by your scripts, either to speak that information or to make  decisions about how to continue the script processing.  Again, you will find a list of the most important built-in  functions in Appendix C and a complete list of these  functions in BUILTIN.JSD.

7.3.5.3  The Pause and Delay Functions

There are two functions which will cause the processing of a  script to be suspended temporarily, Pause () and Delay ().  This suspension of script processing is often necessary to  prevent a script from getting too far ahead of the  application it is working with.  For example, if you use a function that moves the JAWS cursor to a different part of  the screen or which drops down a menu, you may have to  suspend script processing to allow the action to complete  before the script continues.  The Pause function simply  stops script processing and gives other applications an  opportunity to complete their processing.  Then the script  resumes.  The Delay function stops for a specified length of  time which is indicated by an integer between the  parentheses.  The larger the integer, the longer the  suspension.  However, if the suspension time chosen is not adequate for other applications to complete their  processing, the script will resume anyway.  Sometimes a  trial and error process is necessary to determine the proper  amount of delay.  In general, the Delay function is more  useful when you want to force the script to stop,  irrespective of whether the application is finished or not.

Sometimes a Pause function simply does not seem to make the script suspend for a long enough period, and a Delay is called for to achieve the desired affect. In general, it's worthwhile to try the Pause first. If more suspension time is required, a Delay can then be used.

### 7.3.5.4 A Word About SDM Windows

No sooner do you learn something about Windows when you have to learn about an exception. We talked earlier (See Chapter 6.) about the interrelationship of parent and child windows and described how every control (i.e., button, edit field, checkbox, etc.) is really a window in its own right and is the child window of the parent which spawned it, usually a dialog box. Well, it turns out that there is one time when this isn't true. Windows has a type of window which it uses in some of its applications called an SDM window which stands for standard dialog manager. In an SDM window, the controls are not child windows of the parent dialog and, in fact, are not separate windows at all. Thus, they do not have their own handles. This makes it impossible to write scripts to get information about them using the standard techniques used for other controls. For example, in a conventional dialog, if one wanted to get information about the control currently focused on, one could use a nested function statement such as GetWindowSubtypeCode (GetFocus ()). GetFocus () will feed the window handle of the currently focused control to the GetWindowSubtypeCode () function, and this function will then return the SubTypeCode of the currently focused control. In this way you could learn whether the current control was a button, radiobutton, slider, checkbox, or other type of control. This simply will not work for an SDM window since its controls do not have unique handles. Therefore, JAWS incorporates some special functions which you will see in Appendix C which are used to get information about controls in an SDM window.

So how do you know if you are dealing with an SDM window? Well, that part, at least, is easy. You can always use the JAWS hot key combination CTRL+INSERT+F1 to get some information about the window you're focused on, and one of the items you'll get is the window class. SDM windows always have the letters SDM in their name. For example, if you bring up the Open Dialog in Microsoft Word, and use CTRL+INSERT+F1 on the controls in that dialog, you will find that most of them have the window class name bosa_sdm_Microsoft Word 8.0. You will also note that all of the controls with that name also have the same window handle. This is the handle of the Open Dialog itself, and, thus, will not be useful, all by itself, in providing information about the individual controls.  So how is one to  get this information?  The special functions in JAWS which  are designed to do this rely on the fact that the controls  in an SDM dialog do have a unique control ID even if they do  not have a unique window handle.  By using this unique  control ID in combination with the window handle of the  dialog, one can learn information about the SDM dialog's  control.  For example, two of the special SDM functions shown in Appendix C are SDMGetCurrentControl and  SDMGetNextControl.  The first of these will return the  control ID of the control on which the active cursor is  focused, and the second will get the control ID of the next  control in the SDM dialog.  Thus, if one wanted to obtain  more detailed information about the current or next control  in the SDM dialog, he could use the Control ID information  from either of these two functions with one of the other  special SDM statements to report the advanced information  about that control.  For example, such a statement is shown below.

```
SDMSayControl (GetFocus (), SDMGetCurrentControl ())
```

The function SDMSayControl will speak a control in an SDM  dialog, but we have to tell it which one.  Two parameters  are required to do this.  The first one, GetFocus, tells the  function to get the window handle of the currently focused  SDM dialog.  The second parameter, SDMGetCurrentControl,  tells the function we want the control ID of the SDM control  we are focused on right now.  Thus, this function will speak  the name and other information about the SDM control we are  focused on right now in an SDM dialog.  In other words, we  can specify a unique control by telling JAWS both the handle  of the SDM dialog we are referring to and the control ID of  the control we are interested in.  These two pieces of  information, taken together, are sufficient to tell JAWS  which SDM control we want to know about.  The other SDM  functions listed in Appendix C provide other options for  control identification in SDM dialogs.

#### 7.3.5.5  Arithmetic Operators

Arithmetic operators are used to perform classic arithmetic  operations within scripts.  The four operators are a minus  sign or dash (-) for subtraction, a plus sign (+) for  addition, a slash (/) for division, and an asterisk (*) for  multiplication.  These operators are normally used on  integer variables and integers.  Statements such as the  examples shown below illustrate how these operators can be  used

Let IntVar = IntVar + 1  (This will increment the  integer variable called IntVar by 1.)
Let IntVar = FirstVar * SecondVar  (This sets IntVar  equal to FirstVar times SecondVar.)
Say (MyConstant + YourConstant, OT_NO_DISABLE)  (This  will speak the contents of the two constants, MyConstant and  YourConstant, as if they were one message. OT_NO_DISABLE is  an output type constant that tells JAWS when to speak the  message with respect to a user's custom verbosity settings.  This topic will be covered in more detail later.  (See Chapter 8.9.)

### 7.3.5.6  Logical Operators

Logical operators are used to make comparisons of variables  with other variables or constants. We frequently need to  know if one item has the same value, a lesser value, or a  greater value than another item.  Sometimes several items or  groups of items must be compared with other items.  The  logical operators are usually used within If-Then and While  loops to check whether conditions required for logical  decisions are true or false.  A list of the JAWS logical  operators is shown below.

? == - Two equals signs together in this manner ask whether  the first condition is "equal to" the second condition.  That is, is the expression to the left of the two equals  signs equivalent to the expression on the right side.  For example, the expression (A == B) asks whether A  equals B.
? != - An exclamation point and an equals sign together in  this manner ask whether the first condition is "not equal  to" the second condition.  That is, is the expression to  the left of the exclamation point and equals sign  different from the expression on the right side.  For  example, the expression (A != b) is true if A does not  equal B.
? < - A < sign asks whether the first condition is "less  than" the second condition.  That is, is the expression  to the left of the < sign less than the expression to the  right side.  Thus, the expression (A < B) is true if A is  less than B.
? <= - A < sign followed by an equals sign asks whether the  first condition is "less than or equal to" the second  condition.  That is, is the expression to the left of the  <= signs less than or equal to the expression to the  right side.  Thus, the expression (A <= B) is true if A  is less than or equal to B.
? > - A > sign asks whether the first condition is "greater  than" the second condition.  That is, is the expression

to the left of the > sign greater than the expression to  the right side.  Thus, the expression (A > B) is true if  A is greater than B.
?          >= - A > sign followed by an equals sign asks whether the  first condition is "greater than or equal to" the second  condition.  That is, is the expression to the left of the  >= signs greater than or equal to the expression to the  right side.  Thus, the expression (A >= B) is true if A  is greater than or equal to B.
?          && - This operator is placed between two logical  comparisons made using logical operators such as the ones  described above.  It asks whether the first condition is  true "and additionally" is the second condition true.  The total expression is evaluated as true only if the comparisons on both sides of the && operator are true.  Thus, in the expression (A == B) && (C != D), a value of  true is only returned if A does equal B and C does not  equal D.
?          || - This operator is placed between two logical  comparisons made using logical operators such as the ones  described above.  It asks whether either the first  condition is true or whether the second condition is  true.  The total expression is evaluated as true if  either of the comparisons on either side of the ||  operator is true.  Thus, in the example (A == B) || (C ==  D), a value of true is returned if either A equals B or C  equals D.  Of course, a value of true would also be  returned if both expressions were true.

This is a good place to introduce you to a convention some  programmers employ when using logical operators.  Sometimes  you will see a statement such as the one shown below.

```
!Expression == True
```

where Expression can be a variable or function.  Please note  that there is an exclamation point at the beginning of the  word, Expression.  This exclamation point means, in essence,  that we want to use the opposite of Expression and,  therefore, Thus, the above statement would be equivalent to  saying Expression == False.  Thus, an exclamation point has  the effect of inverting the logic of the statement.

7.3.5.7  Bitwise operators

Bitwise operators are a tool that allows us to compare two  pieces of data in a rather unusual way.  What they allow us  to do is compare two pieces of information bit by bit  instead of as a whole value.  We all know that computer  information is stored in digital form as a series of ones and zeros.  This is, of course, true for variables and  constants.  For example, the JAWS constant for highlighted  or selected text is called Attrib_highlight.  If you look in  the HJCONST.JSH file, you will find that this constant has a  value of 64.  64, what does that mean?  Well, 64 is obviously a base 10 or decimal number, not a binary number.  We can tell this because binary numbers only use ones and  zeros, so 64 cannot be a binary number.  When we translate  64 into binary we get the number 01000000.  (A discussion of  binary arithmetic is beyond the scope of this manual, so if  you don't understand how this conversion was made, you'll  just have to accept this as the truth.)  HJCONST.JSH also  tells us that the bold attribute, Attrib_Bold is 2 or 00000010 in binary, the italics attribute, Attrib_Italic, is  4 or 00000100 in binary, and the underline attribute,  Attrib_Underline, is 8 or 00001000 in binary.  Remember how  we said early on in this manual that constants were used  because they're easier to remember than numbers?  Well, here's an excellent example of that situation.  Names such  as ATTRIB_HIGHLIGHT are much easier to remember than long  binary numbers.

So what can we do with these numbers, and what do they have  to do with bitwise operators?  Let's say we want to know if  a certain piece of text is highlighted.  We will use one of  the bitwise operators which is designated by a single & sign  to figure this out.  This bitwise operator allows us to take  a variable and compare it, bit by bit, with another variable  or a constant such as ATTRIB_HIGHLIGHT.  There is an  essential difference between a standard comparison and a bitwise comparison of two pieces of information using the &  operator.  In the standard comparison, a value of true or  one will only be returned if both pieces of information are  identical in every respect.  In a bitwise comparison using  the & operator, a value of true or one will be returned for  every pair of bit positions in the two pieces of information  which are identical.  Let's say we want to do a comparison  of two binary numbers, 0101 and 0001.  In a standard comparison, these two numbers are not the same.  In a  bitwise comparison, the rightmost digit is the same for both  numbers, so a value of 0001 would be returned for the  bitwise comparison.  This return means that the comparison  was true, but only in the first or rightmost position.  If  we were to compare the numbers 0111 and 0110, we would see  that the second and third positions from the right are both  ones in both numbers.  Therefore, the bitwise comparison  would return a value of 0110.  Similarly, 0011 and 0110  would return a value of 0010.  Only the second digit from the right is a one in both numbers.  We can do similar  comparisons when one of the numbers is a variable like, for example, the variable that reports the attributes that are associated with a piece of text in the NewTextEvent script function. (Recall that the NewTextEvent function runs automatically every time new text is written to the screen.) The following line is taken from that function.

If (nAttributes & ATTRIB_HIGHLIGHT) Then

This is a pretty strange statement at first glance. The variable nAttributes is the parameter that brings attribute information into the NewTextEvent function when new text is written to the screen. So, if this text is highlighted, nAttributes will have the value 01000000. If the text is bold, it will have the value 00000010. If its italicized and bold, nAttributes will have the value 00000110. If it is highlighted, bold, and italicized, it will have the value 01000110. You can think of each position in nAttributes as a toggle switch that turns one of the possible attributes on or off. Well, we already know that ATTRIB_HIGHLIGHT has the value 01000000, so the above line of code says take nAttributes and compare it using the bitwise & operator to the constant Attrib_highlight. If there is a match, report back a value of one for each bit position that matches. If we compare, for example, text which is highlighted, bold, and underlined (nAttributes = 01000110) in a bitwise fashion with ATTRIB_HIGHLIGHT (01000000), we will return a value of 01000000. This means that the If-Then statement is returning a value of true or 1 for the seventh position from the right, the one which indicates whether or not text is highlighted, and the NewTextEvent has figured out that the newly-written text is, indeed, highlighted. Therefore, the text will be handled as highlighted text. The fact that this text is also bold and italicized is true, but irrelevant to this test. The above If-Then statement is only checking to see if the highlight bit is set to true or 1, and that is the information being sent back to the NewTextEvent script.

There is also another bitwise operator which performs an or operation instead of an and operation. This operator is designated by a single | to differentiate it from the standard or operator which is designated by two of these symbols, ||. When you use the bitwise or operator, a value of true is returned if either or both of the bits in the comparison is equal to 1. Thus, a comparison of the two binary numbers 0101 and 0010 would return a value of 0111. Similarly, comparison of the numbers 0100 and 0001 would return a value of 0101. Thus, the bitwise or operator is satisfied if either of the numbers has a 1 in a particular position.

## 7.3.5.8 Hook Functions

Sometimes it is necessary to put JAWS into an altered state where some or all of the keys on the keyboard do different things from their normal functions. We are already familiar with this concept to some extent. You can toggle such settings as screen echo, keyboard echo, and the home row state, the last being a feature which will be described later. (See Chapter 8.1.) These types of toggles are available because JAWS has built-in functions that can be called by scripts to toggle these states in predefined ways. However, JAWS also contains two functions that allow you to alter the state of the system to suit your own needs. These are the AddHook () and RemoveHook () functions. In a nutshell, what happens is this. If you install a hook by calling the AddHook () function from a script, we say that a hook is in place. When you use the AddHook function, you give it the name of a hook. This hook is a special user- defined function that you must create which is to be run instead of the code normally run by JAWS scripts. When a hook is in place, it is called right before every script is run, and passed the names of the current script and frame as its two parameters. Since the hook function is actually called by JAWS once it is in place and not by a script, these parameters are supplied by JAWS internal code and not by a function call. If the hook returns TRUE, the script is allowed to execute. If the hook returns FALSE, the script will not be allowed to run. Thus, if a hook is in place and we have designed the hook to return a value of false while it is in place, the script assigned to the key we pressed will not run, and we can have the hook function do something else instead. You could, if you wanted, design a hook that would make every key which is assigned to a JAWS script recite "Mary had a little lamb" instead of performing its normal JAWS function. Now this wouldn't be of much use, but it would be pretty funny if you had such a hook you could turn on every time you let a friend use your computer! Let's look at an example of a more useful hook function, the KeyboardHelpHook.

As you probably know by now, you can turn on keyboard help by pressing INSERT+1. When you do this, all the keys on the keyboard start reciting their functions if they are attached to a JAWS script instead of actually performing those functions. When you press INSERT+1 again, the keyboard returns to performing the functions instead of reciting the descriptions. Did you ever wonder how this is done? Well, guess what? It's done with hook functions. Let's look at the KeyboardHelp script and the KeyboardHelpHook function to see how these tasks are accomplished.

```
Script KeyboardHelp()
SayMessage (ot_status, msg401_L, msg401_S) ;"Keyboard Help  on" ;spoken first time Insert+1
is pressed.  let nKeyboardHelpSavedTypingEcho =  GetJCFOption(OPT_TYPING_ECHO) ;saves
your current keyboard  echo setting.  SetJcfOption(OPT_TYPING_ECHO,1) ; echo characters
;Keyboard  help needs typing echo to be in the characters state.  AddHook (HK_SCRIPT,
"KeyboardHelpHook") ;Adds the hook named  KeyboardHelpHook.
TrapKeys(TRUE) ;When True, this function makes JAWS ignore  any keys not attached to
;scripts
EndScript
```

What's happening here is that a message is spoken telling  you keyboard help is on, your current typing echo is saved,  typing echo is set to characters so the help can function  properly, and the KeyboardHelpHook is installed.  Additionally, TrapKeys is set to true so keys not bound to  JAWS scripts or frames will be ignored.  (You can't give a  help message for a key that has no JAWS script or frame  associated with it.)  Please also note that the AddHook  function requires two parameters.  The first, HK_SCRIPT, is  a constant defined in HJCONST.JSH you must use for every  AddHook.  The second is the name of the hook you are  installing.  So now, any key that is tied to a JAWS function  will not execute its function but will call the  KeyboardHelpHook function shown below.

```
Void Function KeyboardHelpHook (string ScriptName, string  FrameName)  if (ScriptName ==
"KeyboardHelp") then ;If we press INSERT+1  while the hook is installed,
                    ;it means we want to turn Keyboard Help off.  SayMessage (ot_status,
msg402_L, msg402_S) ; "Keyboard  Help off"
          RemoveHook (HK_SCRIPT, "KeyboardHelpHook") ;This  removes the hook.
          TrapKeys(FALSE) ;Now all keys will be recognized again.
SetJcfOption(OPT_TYPING_ECHO,nKeyboardHelpSavedTypingEc  ho) ;Restore our
;original keyboard echo.  return FALSE ;Even though we're turning the hook off,  we'd better
return false one
                    ;more time, or the INSERT+1 script will run again  and reinstall the hook.
EndIf
```

```
;The rest of the function, as shown below, decides what to  speak depending upon whether the
key is tied to
                    ;a key or a script and whether the key was pressed  once or twice.  Study this
                    ;code until you understand it thoroughly as an  exercise.  If you have trouble
                    ;understanding all of the If-Then-Else statements,  read the next section on
                    ;Controlling flow, then come back and study this  script again.  if IsSameScript ()
then
          if ((FrameName != "") && (GetFrameDescription  (FrameName) != "")) then
                    SayMessage (ot_help, msg420_L, msgSilent) ; "This  frame "
                    Say (GetFrameDescription (FrameName), OT_HELP)  EndIf
          if (FrameName == msgNull_L) then  Say (GetScriptDescription(ScriptName), OT_HELP)
          EndIf  else
          Say (GetCurrentScriptKeyName (), OT_HELP) ; says key  name, "INSERT+UP ARROW"
          if (FrameName != msgNull_L) then  SayMessage (ot_help, msg421_L + FrameName) ;
"says  the content of the frame
                    Say (GetFrameSynopsis (FrameName), OT_HELP)  else
                    Say (ScriptName, OT_HELP) ; will say the name of  the script, "SayLine"
                    Say (GetScriptSynopsis(ScriptName), OT_HELP)  EndIf
EndIf  return FALSE ;Prevent the script from running since we only  want the help message.
EndFunction
```

## 7.3.5.9  Introduction to Script Writing With Microsoft  Objects

If you happen to have perused some of the scripts in  Microsoft PowerPoint or Excel, you may have come across some  pretty strange code that doesn't resemble any of the script  building blocks we've talked about so far.  This code  includes words like Selection, Range, Text, and Collapse  separated from one another by periods.  Unless you're  already familiar with object-oriented programming using  languages such as Visual Basic and C Plus+, this type of  code will be completely incomprehensible.  So what is this  stuff, and what's it doing in the middle of JAWS scripts?

To explain this, we need to introduce three new concepts, objects, methods, and properties.

An object is a self-contained construct which possesses within itself a number of characteristics or attributes which are defined as the properties of the object. The object also contains a number of techniques or functions which can cause something to happen to the object or which allow the object to perform some action, and these are called the methods of the object. The properties can be used to return information back to the outside world about the object, and they can also be used to alter the qualities or attributes of the object. This information can be passed to another method or property for further processing or back to the JAWS script language to be used for script processing. Thus, the object contains within itself all of the attributes or properties as well as all of the techniques or methods that can be used to cause something to happen to the object or return information from it. Certain properties are read-write which means they can be looked at as well as changed in certain predefined ways, certain properties are read only which means you can determine what the property is but not change it, and some properties are write only, which means you can change them but not look at them. The methods of an object are specific to that object and are used within the object. They are not used on other objects. That is what we mean when we say an object is self-contained. It possesses all the devices needed to change it in all of the ways it is allowed to be changed and to return information about itself to the outside world.

So why do we care about these objects? There are certain things which users like to be able to do such as read and navigate by sentence and paragraph instead of just by line. Unfortunately, it's rather difficult to do this in most word processors since they aren't set up to navigate in this fashion. Furthermore, we must remember that a screen reader, by definition, reads what's visible on the screen, and if part of the paragraph or sentence we're trying to read is off the screen, it would normally not be possible to access that material without scrolling the screen. The user might not wish to do this since, for example, he or she might be in the process of editing text and not want to move the cursor. Objects provide us with a way to accomplish these tasks since the object which represents a given document contains all of the methods and properties needed to report the information needed back to JAWS. Microsoft has chosen to expose these "automation objects" to the outside world in certain of its applications such as the Office 97 and 2000 suites, and JAWS, by examining and manipulating these objects, can accomplish the sorts of desirable tasks described above. In essence, JAWS temporarily becomes an object reader in addition to being a screen reader. The term "automation object" is used here to mean an object which is exposed so that other entities such as JAWS can access them. Applications which do not use the sort of object model described here for their coding or do not expose these objects to the outside world cannot be accessed in this fashion. Thus, the use of objects for enhancing screen reading is limited to certain Microsoft applications or any application that chooses to expose these automation objects.

Unfortunately, it is beyond the scope of this manual to provide all of the knowledge and teach all of the techniques necessary to use these objects in scripting. In fact, for those who are not familiar with object-oriented programming, that task would probably require a manual longer than this entire script manual. What we intend to do here is introduce the subject so the objects, methods, and properties which appear in JAWS scripts will not be completely mysterious and to provide some references so that those who wish to can continue on their own to learn this subject. This chapter is intended as an overview only.

Hopefully, it is beginning to become apparent that manipulating these types of objects is fundamental to the activities that occur in Microsoft Office applications, and, that by manipulating these objects, JAWS can access desirable information from the object model. In fact, if you take the time to learn how to record a macro in Word (which is simply done from the Macro submenu of the Tools menu), you will see something very interesting. (You can access information on macro recording from the Windows help system by pressing F1 while in Word. Go to the Index tab and type in the word macro. You will be presented with several topics, including macro creation.) Try recording a macro while doing some typical operation like opening a document, saving a document, or selecting and then deleting some text. Then, after the macro recording has been turned off, go back to the Macros submenu and press Enter on the first option which is Macros, ALT+F8. You will open a list of your existing macros and be given the option to edit them. Select the edit option, and you will be placed in the Visual Basic editor and shown a copy of your macro. You will note that the code looks just like the kind of object programming we have been discussing. This means that all of the things you do while working in this application are being done by accessing objects and working with their methods and properties. Thus, studying macros you have recorded is one way of learning how objects are manipulated in the Office applications. There is one other useful technique you can use while you are in the Visual Basic editor. If you press F1 while your insertion point is on any method or property of the programming code, a help screen will pop up which explains the term and how it is used. If you do not get this help, it may mean that Visual Basic help is not installed on your system. You can install it from the Microsoft Office installation disk by inserting the disk into your CD ROM drive and then bringing up the Run dialog from the Start menu and typing X:/setup where X is the letter of your CD-ROM drive. Click the add/remove button when the installation dialog appears. Then, when the options list appears, choose one of the office applications such as Word and tab over to the Change Options button. Click this, and you will be placed in another options list. Select the Help choice and tab to the Change Options button again. Click this, and you will be in a third list. Select the Visual Basic Reference option, and then tab over to the OK button and click it. Keep clicking the OK buttons until you return to the top level selection list. Repeat the entire procedure until you have selected the Visual Basic Reference selection for each application in which you are interested. Then click OK at the top level to finish the installation. You can return to the Visual Basic editor at any time by selecting it from the Macros submenu of the Tools menu or by pressing ALT+F11.

Now, let's take a look at one VERSION of a JAWS script which originally appeared in version 3.2 and which used objects to perform a scripting task. In version 3.2, pressing CTRL+NUM PAD PLUS while in Microsoft Word 8 read the current paragraph. The script which was executed is shown below.

```
Script SayParagraph ()
SaveCursor ()
PCCursor ()  if (GetLine () == "") then
        Say (msge18, OT_MESSAGE); "Blank"  else
        Say (GetParagraphContent (), OT_TEXT)
EndIf
EndScript
```

This looks like a perfectly ordinary script which calls a user-defined function named GetParagraphContent (). Let's take a look at that function.

```
String Function GetParagraphContent ()  var
```

```
            object o  let o = oWord.Selection.Paragraphs(1).Range;  return (o.Text);
EndFunction
```

We can see from the first line of this function that it returns a string and does not need any parameters. The second line starts our local variables declaration, and on the third line, we declare an object variable called o. Then the fun begins. The fourth line reads

```
let o = oWord.Selection.Paragraphs(1).Range;
```

Before we analyze this line, we want to mention the semicolons that appear at the ends of some of these lines. They mean the same thing they've always meant, that everything afterwards is a comment. Some programmers add these semicolons to indicate the end of the line's actual code, even if they don't actually put in a comment afterwards. Now, let's analyze this line from left to right. To the left of the equals sign, we are saying we are going to assign some value to the variable o. The first thing to the right of the equals sign is something called oWord. oWord is a global variable that has been declared in the WINWORD.JSH header file, and it is an object variable which holds the application object, the application being Microsoft Word. oWord is defined in the AutoStartEvent function at the beginning of the script file. Recall that this event function runs automatically every time Word is loaded or gets the system's focus. The code in the AutoStartEvent function which was written by the programmers at Henter-Joyce sets a pointer to Word so JAWS will know where to go looking for application objects. If there were nothing else on this line except Let o = oWord, we would be setting the o variable equal to the global object variable oWord, the whole application object. That wouldn't do us much good if what we wanted to do is read the current paragraph. So we're going to have to find some way to extract that paragraph's textual information from the whole application object.

The next item is a period. This is used to separate variables, methods and properties. Then comes Selection. This is the Selection property and, in essence, it returns the Selection object which contains information about what is currently selected in the document. If you have selected some text in the document, that text will be returned in the Selection object. If nothing is currently selected, the Selection object will be collapsed to a single point at the insertion point.

This all makes sense if we think about what we know about selecting text. We know we can only have one block of text selected at a time in a given document, so it makes sense that there can only be one selection object at a time. If nothing is selected, then the selection and the selection object have to be collapsed to a single point, the insertion point.

If a property returns an object, that object will have the same name as the property which returned it. Please keep this in mind because the alternative is total confusion. A property returns an object of the same name, but the starting property is not a property of the newly-returned object. This is true even though they have the same name. The property belongs to the previous object, the one just operated on, not the newly-created object. The new object has its own set of properties and methods, and they do not include the one that created it.

So let's recapitulate what we have so far. We have the whole word application object represented by the variable oWord, and the Selection property has returned a Selection object which contains information about what is currently selected in the document.

Next comes the Paragraph property. Using this property returns a paragraph object. This object contains information about the paragraph indicated by the index number in parentheses which is contained within the selection object that was returned by the Selection property to the left.

After the Paragraph property returns its information, we will have returned an object with the paragraph that contains the insertion point (assuming no text has been selected) or the first paragraph which is part of a block of selected text. (In the first case, there is only one paragraph, and in the second case, the (1) index of the Paragraph object returns the first paragraph which is part of the selection.)

Next comes the Range property. The Range property returns a Range object which represents the portion of a document that's contained in the specified object. In this case, the specified object is the Paragraph object that was returned by the Paragraph property to the left. The Range is similar to a pair of bookmarks. One represents the beginning of the range and the second the end of the range. The Range object which is returned by the Range property using the information from the Paragraph object will have the  beginning and ending points of the specified paragraph set  as the range.  This final object is now set up to allow us  to retrieve the information we need to speak the current  paragraph.

We now come to the next line in the GetParagraphContent  function, the one which returns the information to the  calling script.  That line reads as follows.

```
return (o.Text);
```

Here, o is the Range object that was returned at the end of  the previous line, and Text is the Text property which  returns the actual text in which we are interested from the  Range object.  (Note that in this case, the Text property is  returning actual text and not an object.)  As it is being  used here, the Text property returns the plain, unformatted  text of the range.  At last we've gotten to the information  we've been after.  The text which will be returned by the  Range property in this case will be the text of the current  paragraph.  That, in turn, will be returned to our calling  script by the Return statement of this line of code.  Since  the Text property returns a string, we will be returning a  string to our calling script, just as we specified in the  first line of our function.  The calling script can then  continue its processing and speak the text that has been  returned.

The key to all of these complicated machinations is that  each object can only use certain properties and methods (as  we said near the beginning of this section) and cannot use  the properties and methods of other objects.  To get the  information we want, we must use the correct method or  property for our current object, and we must use that method  or property in a way that will return a new object that is  properly set up to allow us to access the information we need from the next level down in the hierarchy.  If we do  this correctly, we will eventually get to the information we  need.  Let's perform an experiment to show you how the  properties used in the GetParagraphContent function extract  the information we need from the application object and how  altering the way we use those properties alters the  information we ultimately hear.  A cautionary note is  required before continuing.  The following experiment  involves modifying the WINWORD.JSS and WINWORD.JSB files.  It is strongly advised that you make backup copies of these  files for safety before making any of the changes.  This  will allow you to retrieve undamaged copies of these files  in case you inadvertently break something.

To perform this experiment, open the script file for Word 8 or 9, and move to the end of the file. Because JAWS versions after 3.2 have changed or eliminated the SayParagraph script and GetParagraphContent function described above, we are going to insert special versions of these that should allow someone using any version of JAWS to try the following experiments. Copy and paste the function and script shown below into your WINWORD.JSS file, making sure the function appears in the file before the script:

```
String Function TestGetParagraphContent ()  var
        object o  let o = oWord.Selection.Paragraphs(1).Range;  return (o.Text);
EndFunction

Script TestSayParagraph ()
SaveCursor ()
PCCursor ()  if (GetLine () == "") then
        Say ("blank", OT_MESSAGE)  else
        Say (TestGetParagraphContent (), OT_TEXT)  EndIf
EndScript
```

Next, press ALT+F to drop down the File menu and press the letter Y to synchronize your documentation. You should hear the message, "synchronization complete." Now, place your cursor within the TestSayParagraph script and press CTRL+D to open the Script Information dialog. It's not necessary to enter any Synopsis or Description since this script is temporary, and we'll be deleting it when we're done. So, tab over to the Assign Hot Key field and enter a convenient keystroke which will not interfere with other hot keys you may have defined or any Word accelerator keys. The author chose ALT+CTRL+F12 as being sufficiently unusual as to qualify for this purpose. Then press OK. You will be returned to the main Script Manager editing window.

There is one final step that must be taken before we can start our experiment. You must add a few lines of code to the script file so the test function and script will compile and run. Before the AutoStartEvent function, the first function in the script file, place the following two lines:

```
Globals
Object oWord
```

Finally, you must place the following five lines at the end of the AutoStartEvent function, just before the EndFunction line:

let oWord = MSOGetMenuBarObject () let oWord = oWord.Application; initializes a pointer to the Word application object
if !oWord () then let oWord = GetObject ("Word.Application"); EndIf

These additions define a global object variable called oWord and then initialize a pointer to the Word object. Now, press CTRL+S, and you should hear "compile complete." If you do not, and you get some errors, check your work and fix any errors until the file compiles successfully.

Now, place your cursor in the TestGetParagraphContent function and arrow down to the first line after the variable o is declared. We will be making some modifications to this line in a moment, but before we do that, let's return focus back to Word for a moment and explore what this function is doing in its unmodified state. You should return to Word and place the cursor on the first word of this paragraph. Use your TestSayParagraph keyboard command (ALT+CTRL+F12, in the author's example), and you should hear this entire paragraph spoken. Now, use the SHIFT+DOWN ARROW hot key to select the first line of this paragraph. Confirm that you have done this by reading the selected text with SHIFT+INSERT+DOWN ARROW, and you should hear the first line of the paragraph spoken. Now, use the TestSayParagraph hot key again. What you should hear is the entire paragraph spoken again. Okay, we've confirmed that the GetParagraphContent function is smart enough to read us the entire paragraph, even though we've selected a portion of it. That's good, because that's the way we want this function to behave. Now, go back to the Script Manager and remove the Paragraphs(1) property from the line that contains it. When you're done, the line should look like the line below.

```
let o = oWord.Selection.Range;
```

Save and recompile your script with CTRL+S. Return to Word, and place the insertion pointat the beginning of the same paragraph you used before. Make sure no text is selected, and then use the TestSayParagraph hot key again, and observe what happens. How interesting! Nothing at all is spoken! Why not? Well, since nothing is selected, the Selection property returns a Selection object which is equivalent to the insertion point. Since we took out the Paragraphs property, the Range property will operate directly on this Selection object and return a Range object which has a beginning and end point at the same location, the location of the insertion point. Obviously, a range which is zero characters wide cannot contain any text, so the Text property on the next line of the script returns a null or empty string, and we hear nothing at all. Now, perform one more experiment. Select the first line of the paragraph again with SHIFT+DOWN ARROW, and use the TestSayParagraph hot key again. This time you will hear the selected text spoken. This is because the Selection object which is returned by the Selection property includes all of the selected text, and the Range property will now set a range at the beginning and end of the selected text. The Text property in the next line will then return this text which will in turn be returned to the calling script for speaking. Make sure you remember to put the Paragraphs property back into the function just as it was before and to recompile the script file so that the function will work properly again. Now the Paragraphs property will, once again, return a Paragraph object from the Selection object and feed this information to the Range property. Once again, you will hear the entire paragraph spoken, whether or not any text is selected. Once you are done experimenting with the test function and script, you may wish to delete them from the script file.

## 7.4 Controlling Flow

Scripts are not always performed in a linear, top to bottom fashion, with each and every line being performed in order and only once. Sometimes some statements are performed many times, and sometimes certain statements are skipped. The types of statements which control the flow or sequence of events of script statements are discussed below. There are three types of flow in a script:

? Sequential - In this method a group of statements is run in order, starting at the beginning, and continuing to the end. This linear flow is the simplest format.
? Selection (Conditional) - In a selection, the flow of activity comes to a "fork in the road" and must choose one or the other of two or more paths. Which path is taken depends on comparisons or conditions which are made by the script statements. Depending on the values of certain variables, one path or another will be taken. This process is called branching.
? Iterative - An iterative section "loops" (i.e., performs

the same statement or group of statements over and over), either while a certain condition is true or until a certain condition is met.

A script can contain any or all of these techniques. Let's look at examples of each of these activities.

## 7.4.1 Sequential

All of the statements that we have worked with so far are sequential statements. That means that each of the statements acts in its turn starting at the top of the script and continuing to the EndScript statement. The following is an example of a script with only sequential statements:

```
Script SayTheAValue()
Var
        Int A
Let A = 1
SayInteger (A)
EndScript
```

## 7.4.2 Selection (Conditional)

The If-Then statement structure provides a means of selecting one path or another. Three statements are required for an If structure: If, Then and EndIf. Let's start with a very simple If-Then statement.

```
If (x == 1)
Then
        Say ("You are right", OT_JAWS_MESSAGE)  EndIf
```

If, and only if, the variable X is equal to 1, the script will execute the Say statement and say the words "You are right". Here's a slightly more complex example.

```
If GetWindowName (GetCurrentWindow ()) == "Main"  Then
Say ("In the main window.", OT_JAWS_MESSAGE)  EndIf
```

The function, GetWindowName, and its parameter, GetCurrentWindow, used in the If statement returns the title of the current window. So what happens is this. If and only if the name of the current window is Main, then the Say function is performed. Otherwise it is ignored. We only take the Then "path" if the condition of the If statement is true.

While the If and Then statements are actually separate statements, it is often conventional to put them on the same line. The above group of statements would then look as follows:

```
If GetWindowName (GetCurrentWindow ()) == "Main" Then  Say ("In the main window.",
OT_JAWS_MESSAGE)  EndIf
```

We will show our If-Then statements in this form from now on.

There are two optional statements in an If-Then structure, the Else and ElIf statements. Each of these statements provides an alternate path, but it does so in a somewhat different fashion. The Else statement tells the script to do certain things if, and only if, the preceding If statement is found to be false. For example,

```
If GetWindowName (GetCurrentWindow ()) == "Main" Then  Say ("In the main window.",
OT_JAWS_MESSAGE)  Else
Say ("Not in a known window.", OT_JAWS_MESSAGE)  EndIf
```

This sequence will say one thing if you are in the main window and something else if you are not. Here is an example of a three way branch:

```
If GetWindowName (GetCurrentWindow ()) == "Main" Then  Say ("In the main window.",
OT_JAWS_MESSAGE)  Else
If GetWindowName (GetCurrentWindow ()) == "Alternate" Then  Say ("In the alternate window.",
OT_JAWS_MESSAGE)  EndIf
Else
Say ("Not in a known window.", OT_JAWS_MESSAGE)  EndIf
```

Here we say one thing if we are in the main window, we say a second thing if we are in a window named "alternate", and we say a third thing if we are not in either of these two windows.

The ElIf statement provides a simpler way of formulating a branch which says "Do something else if a different condition is true". The above script sequence can be reformulated to do exactly the same things using an ElIf statement.  This is shown below:

```
If GetWindowName (GetCurrentWindow ()) == "Main" Then  Say ("In the main window.",
OT_JAWS_MESSAGE)  ElIf GetWindowName (GetCurrentWindow ()) == "Alternate" Then  Say
("In the alternate window.", OT_JAWS_MESSAGE)  Else
Say ("Not in a known window.", OT_JAWS_MESSAGE)  EndIf
```

Here again we have alternative paths to travel.  If the  window name is Main, we go down its path.  If the window  name is Alternate, we go down that path.  Otherwise we go  down the Else path.  The ElIf statement takes the place of  an Else statement followed by an If statement.  You can use  as many ElIf statements as you like.

If statements can be nested.  This means that an If  statement can be placed inside of another If statement.  This allows you to check for a particular condition only if  another condition exists.  The second If condition will only  be tested if the first one is true.  The sequence of  statements within the second If-Then test will, thus, only  be performed if both conditions are found to be true.  Don't  forget to use an EndIf statement for each and every If  statement you use.  Here is an example of a nested pair of  If-Then statements.

```
If GetWindowName (GetCurrentWindow ()) == "Main" Then  If GetWindowClass
(GetCurrentWindow ()) == "Edit" Then  Say ("In the main edit window.", OT_JAWS_MESSAGE)
Else
Say ("In the main window", OT_ JAWS_MESSAGE)  EndIf ;Is this an edit window?
Else
Say ("Not in a known window", OT_ JAWS_MESSAGE)  EndIf ;Is this the main window?
```

In this example, first we check to see if we're in the main  window.  If we are, we check to see if we're in an edit  window.  If so, we say that we're in the main edit window.  If it's not an edit window, we just say we're in the main  window.  The edit condition will only be checked if the main condition is true.  Note that we've used comments to  indicate which If test the EndIf statement is terminating.  This is very helpful in preventing confusion when using  nested If-Then statements.

### 7.4.3     Iterative

As mentioned above, an iteration section provides a looping function. A loop is used to perform a sequence of statements several times, thus shortening the number of statements required. Also, one does not always know in advance how many times the sequence will need to be repeated, so looping provides a way to determine the number of repetitions automatically, depending on conclusions derived from script operation. Looping is done with a While-EndWhile sequence.

A While Loop is a statement sequence that repeats or loops itself while a particular condition is true or until a particular condition becomes true. A While loop consists of two parts, the While statement which sets the condition to be tested for, and the EndWhile statement which terminates the loop. All of the statements within the boundaries of the While and EndWhile statements will be performed repeatedly until the conditions in the While statement are no longer satisfied. Here's an example:

Let's take a fairly common problem. You are working in an editor program that numbers the lines with five digit numbers. Therefore, line 1 is listed as 00001. JAWS spends a lot of time reading the zeros. It would be nice to start reading at the first non-zero wouldn't it. This is a good place to use a while loop. Consider the following script:

```
Script SayEditLine()
SaveCursor ()
JAWSCursor()
RouteJAWSToPC()
JAWSHome()
While (GetCharacter () == "0")
NextCharacter ()
EndWhile
SayFromCursor()
RestoreCursor ()
EndScript
```

Using this script to read the line would eliminate the zeros for us. Here's how it does this. The first four lines save the original cursor, activate the JAWS cursor, and route it to the beginning of the line we are on. Then the While loop looks at each character at the beginning of the line in turn and checks to see if it's a zero. If it is, it just moves on to the next character and checks it as well. When it finds one that is not a zero, the condition for continuing to loop (GetCharacter () == 0) is no longer true, and the While loop is terminated. JAWS then continues to process the remainder of the script after the While loop. This involves reading the rest of the line from the cursor to the  end of the line.  That leaves out all of the zeros.

There is a warning about the use of While loops which must  be mentioned here.  Since the loop continues until a  condition becomes true or false, one must be careful not to  set up a loop with a condition which will not become  satisfied.  To do so would set up a loop which will cycle  forever, an infinite loop.  The effect of this would be to  lock up the computer until the program is terminated  manually.  Think carefully about the condition meant to  terminate the loop to be sure it will become satisfied at  some time.  If you find that the computer seems to lock up  after you execute a new script with a While loop in it, this  is probably what is happening.  One way to avoid this  problem while designing While loops is to include some  statements designed to break the loop after a certain number  of repetitions.  An example of this is shown below:

```
While FirstVariable < SecondVariable
Let BreakOutVariable = BreakOutVariable+1
        (Rest of loop would be here.)
If (BreakOutVariable > 100) Then
Return
EndIf
EndWhile
```

This While loop is designed to run until FirstVariable  becomes greater than SecondVariable.  Let's say you made a  logic error, and FirstVariable never exceeds SecondVariable.  Without the extra statements designed to prevent this, the  loop would try to run forever.  However the second line adds  1 to the variable called BreakOutVariable each time the loop  cycles.  When the value of BreakOutVariable exceeds 100, the  If-Then statement at the end of the loop will execute a return and end the cycle.  Once you are sure the While loop  is running properly, you can remove the break out logic.

Homework Assignment #5

Let's use the Sound Recorder program again for this  assignment.  Last time we wrote a script for this program,  we noted that the counter display above the buttons shows  both the elapsed time and the total time of the sound file.  Write a script using If-Then statements and ElIf statements which checks to see if the elapsed time is zero.  If so,  have it say the message, "At the beginning of the  recording."  If the elapsed time is greater than zero but  less than the total time, have it say, "Playback only  partially complete."  If the elapsed time is equal to the total time, have it say, "Playback is complete," and have it click the "Seek to Start" or rewind button to return the counter to zero. Then, have the script pause to give the application time to rewind, and then check the counter to see if it's at zero. If so, have it say the message, "Rewind complete." If the counter is not zero, have it say the message, "Rewind failed. Please check your batteries." Also, use local variables to store the values of the elapsed time and total time. One possible answer can be found in Appendix A. (See Chapter 14.4.)

## 7.5 User-Defined Functions

You can use the script language to write or edit a user- defined function. The difference between a built-in function and a user-defined function, as was discussed above, is that user-defined functions can be modified while built-in functions, which are hard coded into JAWS, cannot. The difference between a user-defined function and a script is that a script can be attached to a keystroke and cannot provide a return while a user-defined function cannot be attached to a keystroke and does provide a return. User- defined functions run whenever they are called from within a script or another user-defined function.

### 7.5.1 Simple Functions

User-defined functions are usually created as a self- contained module of scripting code that has been designed to accomplish a specific purpose. This is particularly useful when the job which the function accomplishes is to be performed numerous times and by different scripts. This allows the function's code to be reused without having to rewrite it multiple times. All the user has to do is call the function from another script or function, and the job will be performed. Then the called function relinquishes control back to the calling script or function which, in turn, continues with its processing. The simplest type of function does not receive any data (parameters) from or return any data (returns) to the calling script. Lets look at an example of a user-defined function that comes from the script file for Microsoft Excel from an early version of JAWS, EXCEL.JSS. This script is called NextSheet, and its purpose is to move the focus to the next spread sheet. Here is the code from the script:

```
Script NextSheet()
;switches to the next sheet and reads the number  {CONTROL+PPAGE DOWN}
Delay (1)
```

```
SayWindowPromptAndText()
EndScript
```

Note that the script calls a function called SayWindowPromptAndText. You will not find that function among the built-in JAWS functions. It is a user-defined function that was written by programmers at Henter-Joyce and is included in EXCEL.JSS. Here is the code for SayWindowPromptAndText from that same version of JAWS:

```
Function SayWindowPromptAndText () SaveCursor ()
InvisibleCursor ()
RouteInvisibleToPC ()
If (FindLastAttribute (ATTRIB_BOLD)) Then SayChunk ()
Else
Say (msg121, OT_MESSAGE) ;"bolded sheet tab not found" EndIf
EndFunction
```

Let's now go through the procedure of creating a simple, user-defined function. We'll have as our goal the creation of a function which speaks our name, just as we did in our first script. As before, we'll place this function in the Notepad script file. Follow the steps as outlined below.

1. Open Notepad from the Start Menu/Programs/Accessories group.
2. Open the Script Manager by pressing INSERT+0 or by pressing INSERT+F2 followed by S and ENTER. JAWS will tell you that you are in NOTEPAD.JSS, the script source file for Notepad.
3. Choose New Script from the Script menu or use the accelerator key combination, CTRL+E.
4. The New Script dialog appears, and we can now name our function and write its documentation. When the New Script dialog appears, the cursor is in the Script Name field. Type in SayName. Be sure to capitalize both the S and N, and don't put in a space.
5. Press TAB twice to move to the Synopsis field. Type in the words, "Say our Name." (By bypassing the Can be attached to key checkbox without checking it, we have told the Script Manager that we are creating a function, not a script.)
6. Press TAB to move to the Description field, and type in the words, "The name of the person who wrote this function."
7. Press TAB to move to the Category field, and type in the

word "Test." The Category field is an edit combo, so you can pick a category by typing one in or by arrowing down through the list to find one that is appropriate.
8. Press TAB to move to the Function Returns field, and arrow down to Void. We choose Void because our function isn't returning any information to the calling script.
9. We won't need any other fields filled in for our script, so TAB over to OK and press ENTER to close the New Script dialog and insert our blank function into the editing area.
10. We are now back in the main text area of the Script Manager, and we have a blank function where the first line says "Void Function SayName ()" and the last line says "EndFunction". We have been placed between these two lines. Arrow up to the first blank line after the first line, and you are ready to start writing the body of the function.
11. You will only need one line for this function. We will use the SayMessage function. This is similar to the Saystring function we used for our first script, but it is preferred because it allows the user to specify an output type constant which is used to tell JAWS how and when to speak the message. This line will be SayMessage (OT_JAWS_MESSAGE, "My name is XXX") where XXX is your name. To get this line into your function, press CTRL+I to open the Insert Function dialog. Then type S, A, Y, AND m. You should hear JAWS say SayMessage, the name of the built-in function that has been selected. Press Enter to accept this function.
12. You will next be asked to supply the output type constant. A list of possibilities is given in the file HJCONST.JSH. We will choose and type in the most appropriate one, OT_JAWS_MESSAGE. After typing in the constant and pressing the ENTER key, you will now be asked to supply the string you wish spoken as a long message. Type "My name is XXX" and don't forget to include the quotation marks. The next parameter is the short message required by the SayMessage function. We will not be using a short message here, so just press ENTER again. Long and short messages will be discussed in a later section. (See Chapter 8.9.) After you press ENTER again, you will be returned to the editing area where your function is now complete.
13. You should now have the following three lines on your page exactly as they appear below. Remember that close is not good enough as computers expect us to be precise and perfect. Also, a few extra blank lines are not important. JAWS ignores blank lines in functions.

```
Void Function SayName()
SayMessage (OT_JAWS_MESSAGE , "My name is XXX" )
```

```
        EndFunction
```

14. Now we need to save and compile the Notepad script file. Press CTRL+S and you should hear, "Compile Complete." If not, go back and retrace the steps and try again.

We have now created a function that will perform the same job as our first script. This function could be called any time we want our name spoken. To call the function, we would use a script formulated as shown below.

```
Script SpeakName ()
SayName ()
EndScript
```

This simple script will call the function which will say our name. Now it might seem a little silly to go to all of this trouble when we could accomplish the same task with our first script. Of course, for a simple case like this that would be true. However, functions can be long and complicated. In the case of such complex functions, it should be obvious that it is much simpler to create the code once and then call it any time we need it. Otherwise, we would have to recreate all of the code for every single script where it was needed.

The user-defined function we discussed a few paragraphs above, SayWindowPromptAndText, is the simplest kind of user- defined function because it does not accept any parameters from and does not return any values to the calling script. However, user-defined functions can do both of these things. Lets look at examples of those.

## 7.5.2 Functions That Require Parameters

The first example is a function that accepts parameters. Remember a parameter is nothing more than a piece of information you send over to the function for it to use when it runs. The only reason that we send it at run time rather than include it as part of the code is because the information may be different each time we run the function. Note the parameter is declared like a variable but is placed between the parentheses following the function name instead of after the word "Var" near the beginning of the script.

```
Void Function ScriptAndAppNames (string sFileName)  var  string theString

let theString = sFileName
If (GetVerbosity() == beginner) Then
```

```
		let theString = theString + msg374 ;" settings are loaded"  else
		let theString = theString + msg374b ; " settings"  endif
Say (theString, ot_help)
If(GetVerbosity() == beginner) Then
		Say (msg376, ot_help) ;"The application currently being  used is the "
EndIf
Say (GetAppFileName (), ot_help)
SpellString (GetAppFileName ())
EndFunction
```

You might think of the string parameter, sFileName, as a  doorway through which we can pass the name of the current  script file into the function when it is called.  The  parameter'sFileName is not equal to the filename, it is more  like a variable which will hold the filename temporarily so  the filename can be passed to the function which will use  it.  This calling statement would be formulated as follows:

```
ScriptAndAppNames ("ApplicationName")
```

Where ApplicationName represents the name of the application  you wish to pass to the function.  This function call is  saying the string, "ApplicationName", is to be passed into  the function, ScriptAndAppNames, via the parameter,  sFileName.  The function then "knows" this information.  Therefore, it can be used in the spoken messages which are  contained within the Say statements.  Here the parameter's  information is combined with fixed text to complete the messages which will be spoken during execution.

Let's take the function we wrote before, SayName, and  reformulate it to accept our name as a parameter instead of  having our name contained in the function's code.  Follow  the steps outlined below.

1. Open Notepad from the Start Menu/Programs/Accessories  group.
2. Open the Script Manager by pressing INSERT+0 or by  pressing INSERT+F2 followed by S and ENTER.  JAWS will  tell you you are in NOTEPAD.JSS, the script source file  for Notepad.  Choose New Script from the Script menu or  use the accelerator key combination, CTRL+E.
3. The New Script dialog appears, and we can now name our  function and write its documentation.  When the New  Script dialog appears, the cursor is in the Script Name  field.  Type in SayName.  Be sure to capitalize both the  S and N, and don't put in a space.

4. Press TAB twice to move to the Synopsis field. Type in the words, "Say our Name." (By bypassing the Can be attached to key checkbox without checking it, we have told the Script Manager that we are creating a function, not a script.)

5. Press TAB to move to the Description field, and type in the words, "The name of the person who wrote this function."

6. Press TAB to move to the Category field, and type in the word "Test." The Category field is an edit combo, so you can pick a category by typing one in or by arrowing down through the list to find one that is appropriate.

7. Press TAB to move to the Function Returns field, and arrow down to Void. We choose Void because our function isn't returning any information to the calling script.

8. Press CTRL+TAB to go to the Parameters tab. You will be placed in the existing parameters field. This field is empty since no parameters exist as yet.

9. TAB over to the New Parameter edit field and type in "UserName."

10. TAB twice until you get to the Description edit field and type in "The name of the user."

11. TAB over to the Available Types listbox and arrow down to String.

12. TAB over to the Add button and press the Spacebar to add the parameter. You will be returned to the New Parameter edit field, but we won't use this again since we're only adding the one parameter. TAB over to the OK button, and press ENTER.

13. We are now back in the main text area of the Script Manager, and we have a blank function where the first line says "Void Function SayName (String UserName)" and the last line says "EndFunction." We have been placed between these two lines. Arrow up to the first blank line after the first line, and you are ready to start writing the body of the function.

14. You will only need one line for this function, but it will be slightly different from the one we used in the SayName function we wrote previously. This line will be SayMessage (OT_JAWS_MESSAGE, "My name is " + UserName ) where UserName is the parameter which passes your name into the function. To get this line into your function, press CTRL+I to open the Insert Function dialog. Then type S, A, Y, and M. You should hear JAWS say SayMessage, the name of the built-in function that has been selected. Press ENTER to accept this function.

15. You will next be asked to supply the output type constant. A list of possibilities is given in the file HJCONST.JSH. We will choose and type in the most appropriate one, OT_JAWS_MESSAGE. After typing in the

constant and pressing the ENTER key, you will now be  asked to supply the string you wish spoken as a long  message.  Type "My name is" + UserName, and don't forget  to include the quotation marks.  The next parameter is  the short message required by the SayMessage function.  We will not be using a short message here, so just press  ENTER again.  Long and short messages will be discussed  in a later section.  (See Chapter 8.9.)  After you press  ENTER again, you will be returned to the editing area  where your function is now complete.

16. You should now have the following three lines on your  page exactly as they appear below.  Remember that close  is not good enough as computers expect us to be precise  and perfect.  Also, a few extra blank lines are not  important.  JAWS ignores blank lines in functions.

```
	Void Function SayName(String UserName)  SayMessage (OT_JAWS_MESSAGE, "My
name is " + UserName )
	EndFunction
```

17. Now we need to save and compile the Notepad script file.  Press CTRL+S and you should hear, "Compile Complete." If  not, go back and retrace the steps and try again.

We have now created a function that will perform the same  job as our previous function. However, the name to be  spoken isn't coded into the function, it is passed in as a  parameter. This function could be called any time we want  our name spoken.  To call the function, we would use a  script formulated as shown below.

```
Script SpeakName ()
SayName ("XXX")
EndScript
```

When the function SayName is called, the string "XXX" which  represents our name is passed into the function via the  parameter, UserName.  The function then has this data and  can speak it as part of the Say message.

### 7.5.3 Functions That Provide Returns

Now let's look at a function that provides a return.  A  return is information that is sent back from the function to  the calling script or function when it runs.  This returned  information can then be used by the calling script or  function.  Note the type of the returned information (i.e.,  string, Int, Object, handle, or Void) is declared before the  function name.  The following function comes from EXCEL.JSS,  the script file for Microsoft Excel.

```
Int Function GetExcelVersion()
```

```
var
          handle WinHandle  let WinHandle = GetAppMainWindow (GetFocus())  let WinHandle =
GetFirstChild (WinHandle)  if (GetWindowClass (WinHandle) == "Excel4") then
          return TRUE ;This is Excel 97  else
          return FALSE ; this is Excel95 or earlier  EndIf
EndFunction
```

You might have figured out that the purpose of this function is to determine whether we are running Excel 97 or not. What happens is that the function performs a test to find out if we are in Excel 97. If so, the function returns or sends a value of true back to the calling script. If not, a value of false is returned. By returning this value, we merely mean that we are informing the calling script of the result of the determination. The calling script can then go on and do other things, depending upon whether the value is true or false, i.e., whether we are really in Excel 97 or not. Here is an example of how this function would be called in a way that the return information can be used to make a decision:

```
If GetExcelVersion== true Then
(Block of statements we want to execute if we're in  Excel 97)
Else
          (Block of statements we want to execute if we're not in  Excel 97)
EndIf
```

In other words, if GetExcelVersionis true (that is to say, returns a value of true after being called), then we'll execute one set of statements. If it's not true (that is, it has returned a value of false after being called), then we'll execute the other set.

By the way, it's time for you to learn a shortcut which script writers often use in situations such as these. The following statement is a little shorter than If GetExcelVersion== true Then, but it means the same thing.

```
If GetExcelVersion Then
```

When you see statements like this, it is understood that we imply "== true". It's left out for convenience, but everyone should understand that "== true" or "== 1" (which, as we now know is an equivalent statement) is always implied in a statement like this.

Homework Assignment #6

JAWS announces capital letters by speaking them in a different pitch. Some users would like to hear words that are all capitalized preceded by the words "all caps" when they are spelled. Write a function called SayAllCaps which does the following things:

1. It should accept as a parameter the word the cursor is on, which is the word to be tested. Use CurrentWord as the string parameter that accepts the data from the calling script.
2. It should check the word to see if it contains any letters which are not capitalized. As soon as it finds one that isn't, it should return a null string, that is, two quotation marks with nothing between them. If it finds that all of the letters are capitalized, it should return an "all caps" string.

Hint - This is a fairly complicated job, so we are going to give you a start. You should use the built-in function, StringContains, to determine if the word contains any non- capitalized letters. (This is the only JAWS built-in function which is case sensitive.) As soon as the test determines that a letter is present which is not capitalized, a return of a null string should be executed since no further testing is required. If no lower case letters are found, it should return the "all caps" string. The function should also return a null string if any numbers are present to prevent it from saying "all caps" when testing a number. Finally, the function should contain code to prevent it from saying "all caps" if the cursor is on a blank or a punctuation mark or any other non-alphabetic character. To accomplish this, you need to know that all such non-alphabetic characters have values which are less than the letter, a or greater than the letter, z. You should test to see if your current word meets this criterion and return a null string if it does.

Finally, to utilize this function, the final part of the problem is to modify the script called SayWord which can be found in DEFAULT.JSS. You must modify this script to call the SayAllCaps function at the appropriate time and perform the test. SayWord is activated when you press the key combination, INSERT+NUM PAD 5. The SayWord script has an initial section which uses the built-in function IsSameScript to check if you've pressed INSERT+NUM PAD 5 twice quickly. Pressing this keystroke combination twice tells JAWS to spell the word instead of saying it. Place your function call to the SayAllCaps function at the appropriate place and call it in such a way that it will speak the string returned by the function just before it spells the word. If the function has found the word is not all upper case, it will return a null string and, thus, not say anything. If the function has found the word to be all upper case, it will return the string, "all caps" which you will then hear before the word is spelled.

Final Hint - You'll have to call the function within a Say statement to hear your returned string spoken.

One possible answer can be found in Appendix A. (See Chapter 14.5.)

## 7.5.4 Event Functions

There is also a special category of user-defined functions whose names end with the word, event. These are called event functions, and they run automatically whenever their associated events take place. For example, a function called MouseMovedEvent runs whenever the mouse cursor changes position. The purpose of these event functions is to allow JAWS to perform specific tasks automatically without waiting for the user to initiate the action with a keystroke. Some clarification is required here. We call these event functions user-defined functions because some user writes or modifies the statements they contain. However, they are different from other user-defined functions because users can't invent new event functions. The ones that exist do so because they have been coded into JAWS to exist. Users can change the code they use but cannot create new ones. There are several of these specific system events that can trigger a function. When these events occur, they often pass parameters to the functions used by the event function's code. Here is an example of this type of function, The AutoStartEvent, that you will find in most script files.

```
Function AutoStartEvent()
If (GetWindowName (GetFocus ())== wn292)
 && SDMGetFocus (GetFocus ())== 18 Then ; on check box  SDMSayControl (GetFocus (), 17) ;
say the tip of the  day
SDMSayControl (GetFocus (), 18) ; say check box  EndIf ; tip of the day
If (WinWordFirstTime == 0) Then  let WinWordFirstTime = 1  let GlobalWinwordVersion = FALSE
```

```
if GetVerbosity() == beginner Then
Say (msg293, OT_MESSAGE) ;"Use Insert plus the  letter H for help in various dialog boxes"
EndIf ; verbosity beginner
EndIf ; first time
EndFunction
```

The AutoStartEvent occurs whenever a script file loads. This one is from one of the versions of the Microsoft Word script file. It speaks the tip of the day, if it is visible, sets up a few variables, and then speaks a JAWS help message. It also checks to see if this is the first time the AutoStartEvent has run during the current JAWS session. If so, it changes the value of the WinWordFirstTime variable from 0 to 1. Then, the next time AutoStartEvent runs, it will skip the help message. This is a useful technique that can be imitated whenever you want a script to do something the first time an application loads but not on subsequent loads. If you want something to happen every time an application loads, just put the code into the AutoStartEvent function without any conditional statements surrounding it. This function does not get any parameters or return any values. Now, lets look at a more complex function that gets parameters passed to it. This is the NewTextEvent function which handles all text newly written to the screen.

```
Void Function NewTextEvent (handle hwnd, string buffer, Int  nAttributes,
Int nTextColor, Int nBackgroundColor, Int nEcho, string  sFrameName)
; Handles all newly written text.  If the text is contained  in a
; frame, then the frame name is passed as a parameter  if (ProcessSelectText(nAttributes, buffer))
then
        return
EndIf  if (sFrameName == "") then
        ; this text is not associated with a frame  if (nAttributes & ATTRIB_HIGHLIGHT) then
                SayHighlightedText(hwnd, buffer)  else
                SayNonHighlightedText(hwnd, buffer)  EndIf  else
        ; this text is associated with a frame  if (nEcho == ECHO_NONE) then
                return; frame is being silenced  EndIf
        if (nEcho == ECHO_ALL ||
           (nAttributes & ATTRIB_HIGHLIGHT)) then
                 ; if Frame echo is set to all or the text being  written is highlighted
                 Say(buffer, OT_BUFFER)
        EndIf
EndIf
EndFunction
```

This is the NewTextEvent function from one version of the  default script file.  Note that we can tell this function  has no returns because the word Void is at the beginning of  the function begin statement.  All parameters for this event  are declared in between the left and right parentheses in the function begin statement.  (Remember that parameters  such as these are originally defined in the parameters tab  of the New Script Dialog which is used to define and  document a new script or function.) When JAWS calls this  event function, it automatically passes a value for each of  the prescribed parameters in between the parentheses of the  function call statement.  The user may not add parameters to  or remove parameters from those which have been predefined for each event function.  However, since values for each of  these parameters are passed when the function is called, the  user may use the data provided by any of them while  customizing the event function.  The following describes  each of the parameters for NewTextEvent.

Handle hwnd is the window handle of the window getting the  text.
String buffer is the actual text being written to the  window.
Int nAttributes is the attribute of the text being written.  Int nTextColor is the color of the text being written.  Int nBackgroundColor is the background color of the text  being written.
Int nEcho is the level of echo set for the frame, if any,  that is getting the text.
String sFrameName is the name of the frame, if any, that is  getting the text.

As we said earlier, this function is called automatically  every time new text is written to the screen.  JAWS  automatically passes the values of these seven parameters to  the function when it is called.  By using logic and  performing comparisons on this data, the function can  determine what type of text has been written, whether or not  it is associated with a frame, and what, if anything, JAWS  should say.  If this were not an event function, the user  would have to call it from within another function or script  and would, in that function call, have to specify the variables being used to pass the required information.  Because this is an event function, JAWS does that job  automatically whenever new text is written.

It is worthwhile to study this event function to learn how  JAWS does these sorts of comparisons and how functions are  called within other functions.

The first line after the initial comments in the function is  as follows:

```
if (ProcessSelectText(nAttributes, buffer)) then
```

This line is an If-Then statement which calls a function  named ProcessSelectText.  This function is involved in  speaking text as it is selected and unselected.  The syntax  of this statement is exactly the same as if it had been  written:

```
if (ProcessSelectText(nAttributes, buffer) == True) then
```

The part that says "== true" is always implied if it is left  out.  So if this function call returns a value of true, then  the next line, Return, will be executed.  What's happening  here is that the NewTextEvent function hands off the task of  determining whether the text is being selected or unselected  to the ProcessSelectText function.  If this function decides  that the text is being selected or unselected, it does the  necessary speaking, and it is not necessary for the NewTextEvent to do anything further.  Thus, when control is  returned to the NewTextEvent function with a return of true,  the NewTextEvent function will then terminate via its own  return statement, and the job is complete.  If the  ProcessSelectText function returns false, then the text is  not being selected or unselected, the If-Then statement is  false, and the NewTextEvent function will continue.  The  next line in this function is:

```
if (sFrameName == "") then
```

Since the parameter sFrameName passes the name of any frame  containing newly-written text to the NewTextEvent function,  this parameter will only have a null or empty value if the  text is not associated with a frame.  A null value for the  frame name is indicated by the two quotation marks with no  text between them.  Thus, if there is no frame associated  with the text, there will be no frame name, a null value,  and the If-Then statement will be true.  Then the next  statement in the sequence will be executed.  If the If-Then  statement is false, meaning there really is a frame, then execution will jump to a different point in the script which  handles frame-associated text. Assuming no frame, the next  statements to be processed are:

```
        if (nAttributes & ATTRIB_HIGHLIGHT) then  SayHighlightedText(hwnd, buffer)
```

The first of these two lines is the famous bitwise If-Then  statement we talked about in the section on bitwise  operators.  If this statement is found to be true, we know  the text is highlighted, and processing is passed off to the  next line which is a function call to the SayHighlightedText function.  This function handles the speaking of all newly-  highlighted text.  If the bitwise If-Then statement is  false, the SayHighlightedText function is not called, and,  instead, the next lines are executed.  These are as follows:

```
        else
                SayNonHighlightedText(hwnd, buffer)
```

In other words, if the text is highlighted, call the  SayHighlightedText function.  Else, call the SayNonHighlightedText function.  You've probably already  figured out that the SayNonHighlightedText function is  called to handle the speaking of non-highlighted text.

So what happens if the script had decided early on that the  new text was, indeed, associated with a frame?  In that  case, the function processing would have jumped down to the  following line:

```
        if (nEcho == ECHO_NONE) then
```

This line compares the value of the parameter nEcho with the  constant ECHO_NONE.  If they are found to be equal, this  means that the frame echo is set to none, and nothing should  be spoken.  In that case, a simple return is executed,  nothing is spoken, and we are done.  If this If-Then  statement returns false, the echo is not set to none, and  more work needs to be done.  Then the next line to be  processed is:

```
        if (nEcho == ECHO_ALL ||  (nAttributes & ATTRIB_HIGHLIGHT)) then
```

The first part of this If-Then statement asks if nEcho is  equal to ECHO_ALL.  In other words, is frame echo set to  echo all text?  The second part of the If-Then statement is  our old, familiar bitwise operation which determines if the  text is highlighted.  If either of these statements turns out to be true, (remember that the || operator is an or  statement), then the next line of the function is executed.  This line is as follows:

```
Say (buffer, OT_BUFFER)
```

This statement simply says the contents of the buffer  variable which is the text that has been written to the  screen.  If the frame is not set to echo all or the text is  not highlighted, the function will terminate without having  spoken anything at all.

Wow, what a complicated job!  All of this processing is done  automatically by JAWS in the blink of an eye, every time new  text is written to the screen.  This is a perfect example of  how JAWS is continually monitoring the screen writes and  deciding what you need to hear.  This function embodies many  of the aspects of the JAWS script language which we have  been talking about, and you should study it until you  understand it thoroughly.  When you do, you will be well on  the way to an in-depth understanding of how the script  language works and how to create your own advanced scripts.

7.5.4.1  List of Event Functions

The following is a list of the special functions triggered  by system events.  If you wish to use these functions in  your script writing, You must use the name exactly as listed  here and assign all the prescribed parameters in the  function call in the prescribed order.  For more detailed information on any of these event functions, see the script  documentation for each one by calling up the script  documentation dialog in the default script file.  Note that  this list has been updated to include event functions from  JAWS 3.7.  If you are using an earlier version, you will not  have all of the items shown below.

ActiveItemChangedEvent - Triggered when the active element  in an object changes.
Parameters:
A handle containing the handle of the window in which  the active object or child has changed.
An integer containing the ID of the current object.  An integer containing the ID of the current object  child, such as the new item in a list or tree control.  A handle containing the handle of the previous window  in which the object changed.
An integer containing the ID of the previous object.  An Integer containing the ID of the previous object  child, such as an item in a list or tree control.

AppWillNotSpeakEvent - This event function is only used  during JAWS installation and is not meant for general  purpose scripting.
AutoStartEvent - Runs when the script file containing this  function loads or gets focus.
Parameters: None
AutoFinishEvent - Runs when the script file containing this  function is closed or loses focus.
Parameters: None
BottomEdgeEvent - Runs when the active cursor encounters the  bottom edge of the window.
    Parameters:
        A handle containing the window handle of the window  whose edge is reached.
TopEdgeEvent - Runs when the active cursor encounters the  top edge of the window.
    Parameters:
        A handle containing the window handle of the window  whose edge is reached.
ClipboardChangedEvent - This event is called whenever the  contents of the clipboard is replaced.
    Parameters
        None
CursorShapeChangedEvent - This event occurs when the mouse  pointer shape changes.
    Parameters:
        A string containing the current cursor type.  DocumentLoadedEvent - Performs a SayAll when a new web page  or html document is loaded and the virtual cursor is  active.
    Parameters: None  FocusChangedEvent - Runs when the visual focus changes.
    Parameters:
        A handle containing the window handle of the current  window.
        A handle containing the window handle of the prior  window.
FocusPointMovedEvent - Called whenever the PC cursor moves  to a different location on the screen.
    Parameters:
An integer specifying the current horizontal position  of the PC cursor.
        An integer specifying the current vertical position of  the PC cursor.
An integer specifying the previous horizontal position  of the PC cursor.
An integer specifying the previous vertical position of  the PC cursor.
An integer containing the unit of movement as defined  in HJConst.jsh.

An integer specifying the direction of movement as  defined in HJConst.jsh.
An integer containing the amount of time that has  elapsed since the  movement occurred.
ForegroundWindowChangedEvent - This event is triggered  every time a real window gets the focus.

Parameters

A handle containing the window handle of the real  window with focus.

FormsModeEvent - Runs when forms mode is turned on or off.  Parameters:

An integer containing the on or off status of forms  mode.

ItemNotFoundEvent - Runs when a requested item is not found  in the off-screen model.

Parameters:

An integer containing the window handle of the current  window.

KeyPressedEvent - triggered by any keystroke attached to a  script before that script is run.
Parameters:

An integer containing the numeric key code.  A string containing the key name as it would appear in  a key map.

An integer defining whether the key is attached to a  Braille display.

An integer defining whether the key is attached to a  script.

MenuModeEvent - Runs when the focus enters a menu.  Parameters:

A handle containing the window handle of the current  window.

An integer containing the menu mode.  MouseMovedEvent - Triggered when the mouse is moved.

Parameters:

An integer specifying the X coordinate.  An integer specifying the Y coordinate.

NewTextEvent - Runs when new text is written anywhere on the  screen.

Parameters:

A handle containing the window handle of the current  window.

A string containing the text written.  An integer containing the attribute of the text.

An integer containing the text color.  An integer containing the background color.

An integer containing the echo setting.  A string containing the frame name of the frame where  text is written.

SayAllStoppedEvent - This event is triggered when SayAll is terminated.
Parameters
None
TextSelectedEvent - Runs when text is selected or unselected in virtual mode.
Parameters:
A string containing the text which is being selected or unselected.
An integer describing whether the text is being selected or unselected.
TooltipEvent - This event is called if the jcf option OPT_PROCESSTOOLTIPEVENT is set to one or on. It is also called by the autographics labeler. This event runs each time a tool tip appears. It currently is used only to store the text of the last tool tip in the global string variable strLastTooltip.
Parameters:
A handle containing the window handle of the tool tip. A string containing the tool tip text. ValueChangedEvent - This function is triggered when a change of value of an MSAA object occurs. The MSAA flag must be turned on in your application-specific jcf file to enable this function.
Parameters:
A handle containing the handle of the window in which the object value has changed.
An integer containing the ID of the object whose value has changed.
An integer containing the ID of the object child whose value has changed.
An integer containing the object typecode for the object whose value has changed.
A string containing the name of the object whose value has changed.
A string containing the value of the object whose value has changed.
WindowCreatedEvent - Runs when a new window appears. Parameters:
A handle containing the window handle of the new window.
An integer containing the coordinate of the left side of the window.
An integer containing the coordinate of the top of the window.
An integer containing the coordinate of the right side of the window.
An integer containing the coordinate of the bottom of the window.
WindowDestroyedEvent - Runs when a window disappears.

Parameters:
A handle containing the window handle of the window  being destroyed.
WindowMinMaxEvent - Triggered when the Minimize/Maximize  status of a window changes.
Parameters:
A handle containing the window handle of the window.  An integer describing the general action taking place  in the window.
An integer describing the specific action taking place  in the window.
WindowResizedEvent - Triggered when a window is about to be  resized.
Parameters:
A handle containing the handle of the window being  resized.
An integer containing the left edge of the window being  resized.
An integer containing the top edge of the window being  resized.
An integer containing the right side of the window  being resized.
An integer containing the bottom edge of the window  being resized.

The following event functions pertain specifically to MAGic:

ActiveItemChangedMagEvent - This event function is called  when the selected item changes in a list view or a tree  view to handle all MAGic specific functions that must  occur when the selected item changes.  Parameters: None
FocusChangedMagEvent - This function is triggered when focus  moves from one control to another. Parameters:
A handle for the window which has the focus.  A handle for the previous window which had the focus.  FocusPointMovedMagEvent - This function is called by FocusPointMovedEvent to handle all MAGic specific  functions that must occur when the focus point moves.
Parameters:
An integer containing the X coordinate of the cursor  after the change in focus point.
An integer containing the Y coordinate of the cursor  after the change in focus point.
An integer containing the X coordinate of the cursor  before the change in focus point.
An integer containing the Y coordinate of the cursor  before the change in focus point.
An integer containing the unit of movement as defined  in HJConst.JSH.

An integer containing the direction of movement as  defined in HJConst.JSH.
An integer containing the amount of time which has  elapsed since the movement occurred.  MenuModeMagEvent - This event is triggered when the status  of a menu changes.
Parameters:
A handle containing the handle of the menu.  An integer containing the mode of the menu which has  changed.
NewTextMagEvent - A Function to perform processing that is  specific to MAGic Tracking when new text is written to  the screen.
Parameters:
A handle containing the window handle of the current  window.
A string containing the text written.  An integer containing the attribute of the text.
An integer containing the text color.  An integer containing the background color.
An integer containing the echo setting.  A string containing the frame name of the frame where  text is written.
ScreenMagnifiedEvent - An event function that runs every  time magnification is activated, deactivated, or the  magnification level changes.
Parameters:
An integer which indicates whether or not magnification  is on.
An integer which indicates the level of magnification.

# 8 Script Writing Techniques

## 8.1 Exploring the Application With the Utility Functions

Before we write a script file, we must have some idea of  what we want to accomplish with it.  We also often need to  explore the windows in our application to find out what  types of things are present and how they should be spoken.  We may also need to have detailed information on the windows  such as class, control ID, and handle as well as parent-  child relationships.  There is a set of scripts in the  default script file called the Utility Scripts that may be  of some benefit here.  These utilities aid in the writing of  scripts by allowing the user to explore the parent-child interrelationships of the various windows in the application  and by presenting detailed information about each window  such as class, control ID, etc.  Each of the utilities is  described below along with its assigned keystroke.  All of these utilities are used in the Home Row mode  (INSERT+SPACEBAR).  The Home Row mode is a shifted state of  the keyboard, somewhat analogous to the shifted states  obtained with the Capslock and numlock keys.  This means  that you access each function by using the indicated keystroke after you have turned the home row mode on by  simultaneously pressing INSERT+SPACEBAR.  To leave this  mode, press INSERT+SPACEBAR a second time.

The utility functions described below have been updated to  contain information about the new capabilities added in JAWS  3.7.  Those using older versions of JAWS will not have all  of the keystrokes described in this section.  In addition, a  new feature of the home row functions was added in JAWS 3.7.  The user will hear a spoken announcement when entering and  leaving the window tree for the currently active  application.  Since it is possible to navigate the child and parent windows not only of the active application but of  other applications, JAWS will alert the user when navigation  into and out of the active application occurs.

?	INSERT+H - This is the hot key help and lists all of the  utilities along with their keystrokes.
?	F1 - This key is used to speak information about the  current window or control.  What information is spoken  depends on the output mode you have selected.  This  Output Mode is defined by using the F3 key (see below).  The F3 key is like a selector switch that determines what  data you will get, and the F1 key actually gives you the  data.  When F1 is pressed twice in quick succession, the  requested information is spelled.  Spell Mode does not  apply to integers.
?	INSERT+F1 - This key posts the information requested by  F1 at the selected Output Mode to a message box.  This is  not available on Output Modes that return integers.
?	CTRL+F1 - This key posts the information requested by F1  at the selected output mode to the clipboard.
?	F3 - This key sets the Output Mode which will be spoken  by F1.  When F3 is pressed repeatedly, the Output Mode is  cycled in the following order: SayTypeAndText, window  focus or handle, control ID, window class, window type,  and first real window name.  This means that this is the  data you will hear when you press the F1 key.
?	SHIFT+F3 - This key cycles through the same settings as  F3, but in reverse order.
?	INSERT+NUM PAD HOME - This key sets the output mode back  to SayTypeAndText and can be used to return to this  setting instead of toggling through all of the modes.
?	TAB - This key moves the JAWS Cursor to the next window  of the same logical level as the one you are currently on

and speaks the required information according to the  Output Mode you have set with F3.
?	SHIFT+TAB - This key moves the JAWS Cursor to the prior  window of the same logical level as the one you are  currently on in the tab list and speaks the required  information according to the Output Mode you have set  with F3.
?	F2 - This key moves the JAWS Cursor to the first child of  the current window and speaks the required information  according to the Output Mode you have set with F3.
?	SHIFT+F2 - Moves the JAWS Cursor to the Parent of the  current window and speaks the required information  according to the Output Mode you have set with F3.
?	F4 - This key is a selector which sets the Attribute  search Mode.  The attribute search mode is the attribute  style that will be searched for using the four hot keys  described below.  When repeatedly pressing F4, the  attributes are cycled in the following order: bold,  italic, underline, highlight, and strikeout.
?	GRAVE ACCENT - This key moves the JAWS Cursor to the next  requested attribute.  Attribute search modes are defined  using the F4 key as described above.
?	SHIFT+GRAVE ACCENT - This key moves the JAWS Cursor to  the prior requested attribute.  Attribute search modes  are defined using the F4 key as described above.
?	CTRL+GRAVE ACCENT - This key moves the JAWS Cursor to the  first requested attribute.  Attribute search modes are  defined using the F4 key as described above.
?	SHIFT+CTRL+GRAVE ACCENT - This key moves the JAWS Cursor  to the last requested attribute.  Attribute search modes  are defined using the F4 key as described above.
?	F5 - This key routes the home-row focus to the currently-  active window.
?	F6 - This key toggles the automatic announcement of  window visibility.  When turned on, this feature will  automatically tell the user whether the new window just  moved to with other utility keys is visible or not.
?	F7 - This key announces the visibility state of the  window currently referenced by the home row.
?	F8 - This key will read the entire contents of the window  referenced by the home row.
?	F9 - This key is similar to F1 except that instead of  returning information about the currently-referenced  window, it returns information about the object which  currently has the focus of the PC or JAWS cursor.
?	INSERT+F9 - This key posts the information requested by  F9 at the selected Output Mode to a message box.
?	CTRL+F9 - This key posts the information requested by F9  at the selected output mode to the clipboard.
?	F10 - This key sets the Output Mode which will be spoken

by F9.  When F10 is pressed repeatedly, the Output Mode  is cycled in the following order: Name, Type, and  Subtype.  This means that this is the data you will hear  when you press the F9 key.
?	SHIFT+F10 - This key cycles through the same settings as  F10, but in reverse order.
?	F11 - This key speaks the current setting for the F3 key.  This allows the user to check the setting without  actually changing it by using F3.
?	F12 - This key speaks the current setting for the F10  key.  This allows the user to check the setting without  actually changing it by using F10.
?	INSERT+7 - This key opens the window class reassign  dialog and places within it the class for the window  currently referenced by the home row utilities.  This  behavior is identical to that obtained by INSERT+7  without home row turned on, but when home row is on, the  window class displayed in the reassign dialog is the one  referenced by the home row rather than just the window  class at the active cursor.

Using these utilities helps you to get all the information  you need about controls, parent/child relationships, and  text attributes to use in your own scripts.

Homework Assignment #7

Open the WordPad application.  Then, turn on the Home Row  mode.  Use F3 and F1 to obtain all of the information about  the parameters of the main edit window and the filename edit  field of the Open Dialog.  The answers can be found in  Appendix A.  (See Chapter 14.6.)

## 8.2	Obtaining Window Information With the  ScreenSensitiveHelpTechnical Script

If you need a quick readout on a window's Control ID, Class,  or Handle, this is easily obtained via a script called  ScreenSensitiveHelpTechnical in the default script file.  This script can be accessed with the CTRL+INSERT+F1 key  combination and will speak the Control ID, Class, and Handle  of the window containing your active cursor.  This is a  quick way of getting this information if you don't need all  of the other data provided by the Utility Scripts.

## 8.3	Script File Types

As was discussed earlier, there are several types of files  associated with script writing.  You can tell what the file  type is by its extension.

.JSS - Source script files
.JSH - Header files (included in JSS files) .JSM - Message files (included in JSS files) .JSB - Compiled script files
.JKM - Key map files which contain the keystrokes assigned to various scripts.
.JSD - Script documentation files which contain the script names with their associated synopses and descriptions.

## 8.3.1 Source Files

Source code for script files is very free-form. You can place spaces or blank lines anywhere you like. Capitalization is not required and is ignored by the compiler except for certain string comparison functions such as StringContains (). Having said that there are a number of formatting considerations. While your code may be very easy for you to understand, others may not be able to make sense of it because they don't know what's in your mind, and they may not organize their work in the same way. Therefore, we would like to suggest a few rules of thumb that will make it easier for others to understand your code and will help you to understand scripts written by others.

1. Individual scripts should always be separated by a blank line.
2. Indent any section of code that is subservient to that above and below it. This helps the user to sort out the logic of the script. Consider the following script:

```
Script CheckVerbosity()
If GetVerbosity() == BEGINNER Then  SayMessage (OT_JAWS_MESSAGE, "Beginning
verbosity is  active." )
Else
SayMessage (OT_JAWS_MESSAGE, "Beginning verbosity is  not active." )
EndIf
EndScript
```

Note that If, Else and EndIf are all at the same indent level and the statements within each section are indented farther. How much you indent is up to you, but we suggest about five characters. Using tabs to do indents works better than spaces because indents will be consistent even if you are working in a proportional font.
3. Each function should be on a separate line.
4. Here are a few capitalization suggestions:

A. Begin each statement with a capitalized letter.
B. Many JAWS standard function names are the combination of two or more words that describe the function. The first character of each word in a function name is capitalized. In the previous example, we used GetVerbosity. This helps to have them spoken correctly. When you create variables and functions, you should use the same convention.
5. The structure of the script file is also important.
A. Start each file with a comment that describes what the file contains and how it is intended to be used. List any specific information that someone reading this file needs to know. You might also include a list of variables and how each is used. It is also important to include the date and nature of any changes and improvements that have been made since the file was first written. Now someone can read the first few lines of your file and know all about it.
B. The next section should be any include statements.
C. Place any global variable declarations in the next section.
D. Place any constant declarations in the next section.
E. The next item in your script file should be the AutoStartEvent function if you intend to use one. Any code associated with this function automatically runs when your script file is loaded. This function is normally used to speak any introductory messages or initialize any global variables to values other than zero. For instance, in the default script file, the JAWS start up messages about how to access help are in this function.
F. Your next script should be one entitled ScriptFileName, the script attached to the INSERT+Q keystroke. This script will announce the name of the currently active script file and application. You can find an example of how this script should be formulated by looking in DEFAULT.JSS. You should substitute the name of your application for the data being passed to the ScriptAndAppNames function.
G. Next come the scripts and user-defined functions. Try to choose a logical order so that anyone reading through your script file will be able to make sense of it. For instance, if you have one script that will perform an action and another to perform the opposite action, place them one after the other. Place each function above the first script or other function that calls it. Not doing this could result in an error condition.
1. Use comments, comments, and more comments! Comments provide a road map for anyone trying to work with your

code.  Place a comment line just after the script begin  line of each Script or Function that describes the  purpose of the script or function.  Place a comment in  the code describing any complex statement that may not be  intuitive.  Think in terms of someone trying to revise  your code and always give them the help they need.  You  will even be helping yourself as you may forget the  purpose of a particular section.

### 8.3.2 Compiled Files

When you have finished writing your code, it must be  compiled for use by the JAWS program.  Your file will be  given the same name as the application along with a .JSB  extension when you compile it so the JAWS program will know  when to load it.  The compilation process is performed automatically by the Script Manager when you choose Save  from the file menu.  The source or JSS file is saved, and at  the same time, a compiled JSB file is created and saved.  This compilation includes all header and message files that  have been included by the appropriate statements at the  beginning of the JSS file.

### 8.3.3 Include Files

As we have mentioned before, the JAWS script compiler  provides for including other files into our script file at  compile time.  Once the compilation is complete, everything  in the included files is part of the JSB file and is  available whenever the script file is active.  There are two  types of include files:

? Header Files are designated by the extension JSH and  contain either variable or constant statements.
? Message Files are designated with the extension JSM and  contain message statements with their assigned numbers.

Always group your include statements together near the  beginning of your script file.

Putting global variables and constants into a separate  header file and messages into a separate message file helps  to organize your work better.  Two files are provided with  JAWS that contain all of the standard global variables and  constants.  These are HJGLOBAL.JSH and HJCONST.JSH.  They  are provided this way so that script developers can use any  of the standard variables and constants by simply including  them in their own script files.  Message files are also set  up as includes.  The standard one is DEFAULT.JSM.  It is not  necessary to use a message file for your script file.

Henter-Joyce placed the default messages in a separate file for simpler translation to other languages. If you do use a message file, it should be named the same as your script file so that you can easily identify it.

## 8.4 Using Variables

Our earlier section on variables (See Chapter 7.3.3.) showed you what kind of variables the JAWS language allows and how to declare them. In this section we'll discuss when to use them, suggest some naming conventions and show you where to place them in your script files.

### 8.4.1 Naming Conventions

This is a touchy subject. Everyone has an opinion, and these opinions often conflict. Regardless of what naming convention you choose, using one will help prevent time consuming and embarrassing mistakes. By adopting standards and sticking to them, you're guaranteeing that you can come back to this code later and modify or debug with ease. We in the disability community spend a lot of time talking about adopting standards. Here's your chance to contribute. Let's go over a few suggestions on creating your standard. What's in a name? It can be a lot. Choosing a descriptive name makes a lot of sense. The easiest thing to do is to name your variable by putting descriptive words together. Try not to make it too long as you have to type the names, sometimes a lot. If you are naming a global variable, make its name begin with G_ or Global. You should also capitalize the beginning of each word as you join them together so JAWS can read the words correctly, assuming you have the Mixed Case option turned on in the Configuration Manager. You could carry this further and use Int, Str, Obj, and Hnd within the names to indicate integer, string, object, and handle variables. Thus, G_Int_VariableName would be a global integer variable and Str_VariableName would be a local string variable.

### 8.4.2 Declaration Placement

Global variables that will be used in more than one script in your file should be declared near the beginning of the file, right after the include statements. You indicate the start of the global variable declaration section by placing the word "Globals" on its own line. Then you place each declaration on a separate line. Place a comma immediately after each variable name, except for the last one in the list. While you're at it add a comment at the end of the declaration stating how this variable is used.  Later on you  can refer to this area of the file for information about any  variable that you encounter.

The only time you would declare a variable within a script  is if it is a local variable.  Recall that a local variable  is only used within the script where it is defined whereas a  global variable can be used within any script in the file.  A local variable should be declared on the first line of the  script immediately after the Script begin statement.

### 8.4.3 Using Global Variables

Remember that variables that you intend to use in more than  one script are declared as global.  In addition to the  global variables which you define for your own use, there is  a set of standard JAWS global variables that is very useful  for the creation of scripts.  These variables are contained  in the file HJGLOBAL.JSH, and you can use them by placing an  include statement for this file at the beginning of your own  script file.  Spend a little time going over the global variables defined in HJGLOBAL.JSH as they are used a lot in  the default scripts.  There are descriptions of how each is  used.  Remember the values contained in these variables are  often modified by the default scripts.  Therefore, if you  include HJGLOBAL.JSH in your script file, you can call upon  these variables to get specific information as needed.  For  instance, checking the contents of GlobalMenuMode will tell  you if you are on the menu bar or in a menu.  What we are saying here is that these HJGLOBAL.JSH global variables are  used constantly by the default scripts in JAWS, and if you  understand them and learn how to use them and include them  in your own application script files, you can call upon the  values they contain during the execution of your own  scripts.

It is also worth mentioning that, once you have defined and  assigned a value to a global variable in one of your  applications, that value is maintained in memory, even after  you close that application.  If you later reopen the  application, the global variables you defined will still  have their values.  The only way you can truly clear these  global variables is by closing and then reloading JAWS.

## 8.5 Using Constants

Remember that a constant is just a label which represents  a number or a string of letters so that the script code is  easier to read and understand.  You should include the  standard constants file, HJCONST.JSH, in each of your script files and use the constants in it whenever you can.  The  versions of these files contained in the latest releases of  JAWS contain many, many categories of constants for use in  different situations.  Some of these are quite specialized  and of minimal relevance to most users, so we'll concentrate  on the most important categories here.  The file contains  the following important groups, roughly in the order they  appear:

? Clipboard Constants - These constants all begin with  CLIPBOARD_ and are for use with the ClipboardChangedEvent  function.
? Keyboard Constants - These all start with Key_ and assign  easily-remembered names to keyboard keys.  They are for  use with the KeyPressedEvent function.
? Window Type Codes and Subtype Codes - These codes map to  the type of window.  Remember that even the lowliest  control is classified as a window.  Therefore, Every  window will return a type code if queried.  The push  button will return a window type of 1 or its constant name, WT_BUTTON.  There are two functions used to get  this information, GetWindowTypeCode and  GetWindowSubtypeCode.

Consider the following If-Then statement from the  SayPriorCharacter script OF ONE VERSION OF DEFAULT.JSS.  It  is the script that runs when the left arrow key is pushed.  The else leg of this If statement is run when there is no  visible PC cursor.  This branch is designed to check to see  if we are in a menu.  We've added comments after each  section of code describing its purpose.

```
    If CaretVisible() Then ; Is a PC cursor visible  SayCharacter() ; Say the character that we
land on
    Return ; Quit this script
    Else ; A PC cursor is not visible
        Let TheTypeCode = GetWindowSubtypeCode  (GetCurrentWindow())
            ; Set an integer variable to the current window  type code
        If (TheTypeCode == WT_MENU) Then ; Is the current  window a menu
        Say (msg3, OT_MESSAGE) ;"Menu"  SayWindow (GetCurrentWindow(),
READ_HIGHLIGHTED) ;  Speak the current item
            Return ; Quit this script
        EndIf
    EndIf
```

What's happening here is that the script is deciding what to speak depending on the SubTypeCode of the window we're in. It should be pretty obvious that this set of constants and associated numeric codes is very important to JAWS users as it is from these that we find out what type of a control we are dealing with. You may want to browse this group of constants in HJCONST.JSH. They all start with WT_ for window type.

? Text Attribute Identifiers - These constants are used for text attribute comparisons. They all start with ATTRIB_ for attribute and identify different text attribute types.
? Control Attribute Identifiers - These constants begin with CTRL_, and are used to identify control attributes such as checked, grayed, disabled, etc. These attributes are analogous to the text attributes discussed above but are used to describe control characteristics. They are used by a function called GetControlAttributes which returns a value for all of the control attributes. This value can be compared to the control attribute constants using bitwise operators to obtain the control's characteristics. Bitwise operators were discussed in an earlier section. (See Chapter 7.3.5.7.)
? Commonly Used Constants - These include string and graphic search types that start with S_, as well as True, False, On and Off.
? Output Modes - These are constants which begin with OT_ and which are used to determine when and how JAWS will speak different types of information. These output types are very important for customization of user verbosity levels and will be discussed more fully later. (See Chapter 8.9.)
? Item Types - These constants, beginning with IT_, are used to describe the items in the GetItemRect function.
? Modes for the Braille Display - These all start with BRL_ and are used to set different modes for Braille displays.
? Braille Marking Types - These also start with BRL_ and are used for setting up Braille attribute marking options.
? Verbosity Settings - These are BEGINNER, INTERMEDIATE AND ADVANCED levels and MESSAGE_LONG And MESSAGE_SHORT for long and short message type options.
? Button Control ID - These all begin with ID_ and are used with the built-in function called DLGSelectControls, a function used to display the controls of the active dialog in a list box. The DLGSelectControls function presents the user with several possible actions to perform on the control selected in the list box, such as left single click, right single click, etc. and then returns a Control ID for the action selected. These

constants represent the button Control ID numbers for the various possible actions.
? Button Types - These constants all begin with BT_ and are used with the function DLGSelectControls. They represent the actions, such as left single click, right single click, etc., to be included as choices in the dialog. In other words, these represent the buttons the user will see in the dialog for the various possible actions to be taken on the item selected in the list of controls.
? General - These include testing for active and inactive states, success or failure, cursor types, various reading and highlight settings and options, graphic search constants, and pointer attachment.
? Option Identifiers Used by SetJcfOption and GetJCFOption Functions - These constants, all beginning with OPT, are used to retrieve and set, on the fly, all of the various parameters that are set, permanently, by the Configuration Manager.
? Restriction Types - These constants, all beginning with RESTRICT, are used to describe the restriction state for the JAWS and Invisible cursors.
? Cursor Types - These are used to describe various shapes of the mouse pointer. They begin with CT_. For example, CT_Wait corresponds to the hour glass, and CT_ARROW represents the arrow pointer.
? Menu Modes - These constants describe the active versus inactive states of menus and menu bars.
? Default Files - These constants, beginning with DEFAULT_, describe the default keyboard layouts.
? Magic Focus types - These MF_ constants are for setting how Magic magnifies and displays the screen.
? Tracking Engines - These constants, beginning with TE_, refer to whether JAWS or Magic is being used to track the screen.
? Word delimiters - This single constant is used by the IsWordDelimiter function to determine if a given character is a word delimiter.
? for the Adjust JAWS verbosity dialog - These constants are used to describe various entries for creation of the Adjust JAWS Verbosity dialog.
? Format - These constants begin with FORMAT_ and relate to the various format options available under the settings menu of the Configuration Manager.
Capitalization Types - These constants, beginning with CAPITALIZATION_, are used by the SayFormatAndText function to describe capitalization state such as undefined, lower, cap, and all caps.
? Voice context names - These constants, beginning with VCTX_, represent the different JAWS voices such as Global, Message, JAWS, etc.

? Voice Parameters - These are used for various voice parameters such as volume, rate, pitch, etc. and start with V_.
? Movement Units - These constants, starting with UNIT_, represent cursor movement by various units (character, line, page, etc.).
? Movement Direction Values - These constants are used to represent movement in the forward and backward directions and start with MOVE_.
? The String to Use Between Items in DLGSelectItemInList - This constant represents the separator to be used between items when constructing the item list for DLGSelectItemInList.
? Constants for Use With the WindowsMinMaxEvent - These constants, starting with SW_, are solely for use with the WindowMinMaxEvent function.
? Message Box Type Specifiers - These determine the type of message box (e.g., OK, Yes/No, etc.)which is to be displayed by the ExMessageBox function. They begin with MB_.
? Message Box Return Values - These constants all start with ID and are used to represent the values (e.g., OK, Cancel, Retry, etc.) returned by the ExMessageBox function.

All of the constant categories described above refer, as stated, to the HJCONST.JSH header file that ships with JAWS. We would not want to give the impression that these are the only constants available to you. Of course, you can and should create your own constants, either by declaring them near the top of your script file or by placing them in an included header file. Using constants whenever possible makes comprehension of the script file easier and makes it possible to change the value of a constant everywhere in the file by just changing one declaration. This is much easier than having to change a value at every location where it appears in the file.
Homework Assignment #8

Let's say you are about to script an application called SPAMKILLER.EXE. You will be using a header file, a message file, and the two default HJ global variable and constant files. Also, you will need to define three additional global variables called GlobalSpamInteger, GlobalSpamString, and GlobalSpamHandle and three additional constants defined as SpammerOne =1, SpammerTwo = 2, and SpammerThree = 3 within the script file. Show the beginning of the script file with all of the statements that do these things. The answer can be found in Appendix A. (See Chapter 14.7.)

## 8.6 Using Multiple Functions

Sometimes you have to "string" a number of functions together to achieve your goal. Other situations require "nested" functions, where one function becomes the parameter for another, which is the parameter for yet another. This can look rather confusing in the code. The trick is to count up the parentheses. They will show you how the functions are nested within each other. First we'll look at a group of functions strung together for a single purpose and then we'll look at a nested function.

### 8.6.1 Using Functions Sequentially

It's pretty rare when you can accomplish a task with a single function. Several functions in the proper order are usually required. The following code is from a script file for Visual Basic 4.0. It's job is to check whether the Toolbox is the focus window and, if not, make it so and then position the JAWS cursor on the first tool icon. By the way, we used the home row utility scripts to figure out the window class of the tool box window, ToolsPalette, before we wrote this script.

```
Script GoToToolBox()
; This script focuses on the tool box and positions the JAWS  cursor
; at the first tool of the tool box
Say ("Tool Box", OT_JAWS_MESSAGE) ; Let the user know what's  happening
While GetScreenEcho() != 2 ; Loop through the screen echo  settings until we set screen
verbosity to all  ToggleScreenEcho () ; Increment screen echo  EndWhile
If GetWindowClass (GetFocus ()) != "ToolsPalette" Then If we  are not already in the tool box
PCCursor();
{ALT+V} ; View Menu
Pause() ; Time for the menu to appear  {x} ; Select Tool Box
EndIf
Delay(10) ; provide time for tool box to appear  RouteJAWSToPC()
JAWSCursor()
JAWSPageUp()
JAWSHome()
PerformScript ({DOWN ARROW}) ; Regular down arrow with  special stuff for the tool box
EndScript
```

? First we make sure we are in say all screen echo mode.
? Next we check to see if we are not in the toolbox by getting the window class of the current focus window and comparing it to the string, ToolsPalette, which is the known window class of the toolbox window. (This window class would have been defined in a constant include file for this script file.) If we are not in the toolbox, then we must make that the focus window by performing several functions that will guarantee the end result.
? We use the PCCursor function to force the PC cursor active.
? We use the keyboard simulation to send an ALT+V to the application. This pulls down the View menu. Remember that anything in curly brackets is passed to the application, just as if you'd typed it from the keyboard. Note that there is a comment there to let anyone reading this know that that's the purpose of {Alt+V}.
? The Pause function then gives the system time to pull down the menu before we continue.
? We use the keyboard simulation to send an X to the application which is the accelerator key for the toolbox menu item. This is also commented.
? Then the EndIf terminates our If-Then sequence.
? Now we are sure the toolbox is open and has the focus.
? Then we activate the JAWS cursor and position it on the first tool icon.

We've used a number of functions sequentially to accomplish our task.

## 8.6.2 Using Nested Functions

Here is an example of a script with a nested function wherein functions are used as parameters of other functions.

```
Script IsJAWSWindow()
If (GetWindowClass (GetAppMainWindow (GetFocus())) == JFWUI2) Then
Say (msg113, OT_JAWS_MESSAGE) ;"This is the JAWS application window"
EndIf
EndScript
```

We can find out that the window class of the JAWS window is JFWUI2 by going to the JAWS window and turning on the home row functions. Using SHIFT+F2 will take us to the parent window, and F3 will cycle through the output modes until we come to class. Then F1 will speak the class which is JFWUI2. We want to compare our window class to that, but first we must make sure that our script is looking at the correct place.  We use the function, GetWindowClass to query  a window class.  Its parameter is a window handle, so we  give it AppMainWindow which is the parent in the current  application. We use this because JAWS is an application,  and we want to check the window class of that application.  AppMainWindow requires a window handle as its parameter so  it knows which application to check.  In our case we use the  function, GetFocus, which returns the window handle of the  focus window.  Simply said it is: get the window class of  the application window that has the focus.

Homework Assignment #9

Let's go back to the WordPad Open Dialog once again.  Open  this dialog and look around to refresh your memory.  Write a  script that will determine, first of all, if you are in this  dialog.  If so, have your script say "In the Open Dialog."  Then, determine if the focus is on the filename edit window.  If so, have the script also say "and in the filename field."  If, however, focus is on another control, have the script  say, instead of the second message above, "but not in the filename field."  Finally, if the focus is not even in the  Open Dialog, have the script say "Not in the Open Dialog."  Hint - The Open Dialog is a real window.  Recall that this  means it is a window that has a title.  Use this fact to  determine if you are in the Open Dialog.  Use the window  class of your current control to determine if you are  focused on the filename field.  Second Hint - Look up the  function called GetRealWindow in Appendix C.  One possible  answer can be found in Appendix A.  (See Chapter 14.8.)

## 8.7 Selecting and Manipulating Cursors

Before you attempt to read something from the screen with a  script, it's worthwhile to review some information about  cursors which we covered earlier.  (See Chapter 5.3.2.)  There are four kinds of cursors, the PC cursor, the JAWS  cursor, the Invisible cursor, and the Virtual PC cursor.  If  you are editing or entering text, the PC cursor is where  characters would be placed when you type.  (This editing or  text insertion form of the PC cursor is also called an  insertion point or carat.)  However, when you are using a  menu or tabbing through some selections such as buttons in a  dialog box, the PC cursor is the focus.  If you wish to read  other areas of a window or read a part of the screen where  the PC cursor cannot go, you can switch to the JAWS cursor. The JAWS cursor can be moved anywhere within the boundaries  of the current real window, making it possible to explore  areas of the screen where the PC cursor cannot go.  Since  the JAWS cursor always moves the mouse pointer to its location, the mouse pointer is always positioned and ready to be clicked at the location of the JAWS cursor should you decide to do so. The third kind of cursor is called the Invisible cursor. In one way it is just like the JAWS cursor as its movement is also only restricted by the boundaries of the real window, but it has no visible entity on the screen since it does not bring the mouse pointer with it. The Virtual PC cursor is a special case of the PC cursor which is turned on automatically in certain applications such as Internet Explorer 5, Outlook/Outlook Express, and Eudora 4.X where a Microsoft browser or browser style viewer is being used. The Virtual PC cursor is created within a special JAWS buffer and is employed to allow the user to have a cursor that can navigate around the viewer window even though that viewer does not actually have a normal insertion point. The user is actually navigating in the virtual buffer created by JAWS and feels as if navigation were occurring within a word processor type of application. This makes reading in such applications much easier.

How do we use these cursors? It's generally impractical and often impossible to go exploring around the screen with the PC cursor. Therefore, before we go to read some text from the window, we usually first switch to one of the other two cursors. One can pretty much use either the JAWS or the Invisible cursor to do the reading for you, so to some extent it's just a matter of choice. However, if you plan to click a button or take some other action with the mouse, it will be necessary to use the JAWS cursor. Since the mouse pointer accompanies the JAWS cursor in its travels, it is then a simple matter to click the mouse as it's already in the correct position. If you do not wish to move the JAWS cursor from its current position, or there won't be any need to click on anything, the Invisible cursor is the best choice. There is one particular situation which comes to mind when you definitely won't want to move the JAWS cursor. Sometimes in a Windows application, informational text will appear somewhere on the screen, usually the status line, when the mouse pointer is at a particular location. If you move the JAWS cursor to read the text, the mouse pointer will move with it, and the text will disappear. In this situation, you must use the Invisible cursor to read such text.

In some cases, it is necessary to move one of the cursors such as the JAWS cursor, even though you want it to be back in its original location when the script is finished. For this reason, there are save and restore functions that you can use to put everything back as it was before your script  was executed.

Here is a list of the functions that you can use to select  and manipulate the cursors.

? PCCursor() - Activates the PC Cursor.
? JAWSCursor() - Activates the JAWS Cursor.
? InvisibleCursor() - Activates the Invisible Cursor.
? BrailleCursor () - Activates the Braille cursor.
? RouteBrailleToPc () - Moves the Braille cursor to the PC  cursor.
? RouteBrailleToJAWS () - Moves the Braille cursor to the  JAWS cursor.
? RouteJAWSToBraille () - Moves the JAWS cursor to the  Braille cursor.
? RoutePCToBraille () - Moves the PC cursor to the Braille  cursor, if possible.
? RoutePCToJAWS() - Moves the PC Cursor to the JAWS Cursor,  if possible.
? RouteJAWSToPC() - Moves the JAWS Cursor to the PC Cursor.
? RouteJAWSToInvisible() - Moves the JAWS Cursor to the  Invisible Cursor.
? RouteInvisibleToPC() - Moves the Invisible Cursor to the  PC Cursor.
? RouteInvisibleToJAWS() - Moves the Invisible Cursor to  the JAWS Cursor.
? SaveCursor() - Saves the active cursor and its position.
? RestoreCursor() - Reactivates the saved cursor and  restores it to its original position.

The next group of functions is used to move the cursor.

? PriorCharacter() - Moves the active cursor to the prior  character.
? PriorWord() - Moves the active cursor to the prior word.
? PriorLine() - Moves the active cursor to the prior line.
? NextCharacter() - Moves the active cursor to the next  character.
? NextWord() - Moves the active cursor to the next word.
? NextLine() - Moves the active cursor to the next line.
? JAWSHome() - Moves the active cursor to the beginning of  the line.
? JAWSEnd() - Moves the active cursor to the end of the  line.
? JAWSPageUp() - Moves the active cursor to the top of the  window.
? JAWSPageDown() - Moves the active cursor to the bottom of  the window.
? MoveTo - Moves the JAWS or Invisible cursor to the screen

coordinates specified by the user.
? MoveToControl - moves the active cursor to a specific control within a window. If the PC cursor is on when this function is called, the JAWS cursor is turned on automatically. Otherwise the active cursor is used.
? MoveToFrame - Moves the active cursor to the top left corner of the specified Frame. If the PC cursor is active when this function is used, then the JAWS cursor is activated and it is moved to the new position, otherwise the active cursor is moved.
? MoveToGraphic - Moves the JAWS cursor, Invisible cursor, or Braille cursor in a specific direction to find a graphic symbol in the active window.
? MoveToWindow - Moves the active cursor to the specified window. If the window contains text, then the cursor is positioned on the first character. Otherwise, it is positioned at the window's center. If the PC cursor is active when this function is used, then the JAWS cursor is activated and it is moved to the new position.
? PriorChunk - Moves the active cursor to the prior chunk of text. A chunk of text is a section or block of text that is written to the screen at one time.
? NextChunk - Moves the active cursor to the next chunk of text. A chunk of text is a section or block of text that is written to the screen at one time.

Note that If your purpose is to read something located on the screen, you would usually follow a move function with one of the say functions.

Here is an example of saving the current cursor, switching cursors, moving the cursor, reading some text and then returning to the original position with the original cursor:

```
Script ReadBottomLine()
Var ;Declare our local variables
Int TheTypeCode ;We have an integer variable called  TheTypeCode
Let TheTypeCode = GetWindowSubtypeCode (GetCurrentWindow())  ;Find out what type of
window
;we're in and assign it to TheTypeCode  variable.
SaveCursor() ; Save the current cursor and position  InvisibleCursor() ; Switch to Invisible cursor
RouteInvisibleToPC() ; Bring the Invisible cursor to the PC  cursor
If (TheTypeCode == WT_MENU) Then ; If we are in a menu  MoveToWindow
(GetAppMainWindow (GetCurrentWindow())) ;  Refocus on the main window
JAWSPageDown () ; Go to the bottom
```

```
JAWSHome() ; Go to the left most position  NextWord() ; Move one word to the right  SayChunk()
; Says the chunk of text that was written to  the screen together
Else
JAWSPageDown() ; Go to the bottom
SayLine() ; Say the line
EndIf
RestoreCursor() ; Go back to the original cursor and  position
EndScript
```

This is the script that reads the bottom line of a window.  Usually this is where the status bar is located.  The PC  cursor is not allowed to go to this location, so we must  switch to the Invisible cursor and use it to read this line.  Notice that there is a SaveCursor function before taking  this action and a RestoreCursor function after.  That way,  everything is the same as before and after reading the  status bar.  This script uses the subtype Code of the window  to determine what type of window we are in and, therefore,  what script statements need to be used to read it.

## 8.8 Sending a Keystroke

If you need to send a keystroke or keystroke combination  from a script (that is to say, if you want the script to  enter that keystroke just as if you were entering it from  the keyboard), place the keystroke name in braces.  For  example {CONTROL+H} sends a CTRL+H to the system.  This technique can also be used to send alphanumeric characters.  Thus, {h} will send the letter h to the application you are  using.  Each letter you need to send must be placed within  its own set of braces.  If you need to send more than a  couple of letters, you may wish to use the TypeString function which will send an entire string of characters to  the application at one time.

## 8.9 Making Your Scripts Compatible With Custom Verbosity  Levels

Starting with version 3.7, JAWS allows the user to customize  the beginner, intermediate, and advanced verbosity levels to  suit personal preferences.  This means that the user can  select what type of feedback will be heard from JAWS in a  variety of situations at each verbosity level.  These  choices are made in the Configuration Manager under the Set  Options menu by choosing the item titled "Verbosity."  This  opens the Verbosity Options dialog.  In this dialog, one  sees, in addition to the three radio buttons which allow the user to specify the preferred verbosity level, three buttons which allow for the customization of each level. By activating any of these buttons, the user is placed in a new dialog, either the beginner, intermediate, or advanced preferences dialog, which contains a list box wherein each item can be checked or unchecked with the SPACEBAR. Unchecking an item means it will not give feedback at that verbosity level. Then, one can tab over to a pair of radio buttons which select whether short or long messages will be heard at that verbosity level.

So, how does JAWS know when to speak a particular type of information and how to customize that speaking based upon the user preferences described above? To do this, script writers use a set of constants called output types. These are defined in the file, HJCONST.JSH. These constants all begin with the letters OT_. For example, one of the output type constants is OT_APP_START. If you look at the first entry in the Items to be Spoken list mentioned above, you will see that it is called "Application Start Message." This is the JAWS message you frequently hear when you first start an application. If this item in the list is checked, it means JAWS is being instructed to speak the messages contained in SayMessage functions that use the OT_APP_START constant. For example, a SayMessage function formulated as shown below will speak its message when the Application Start Message item is checked in the Items to be Spoken list.

```
SayMessage (OT_APP_START, MESSAGE_L, MESSAGE_S)
```

If the Application Start Message item is not checked, then SayMessage functions using this output type will not get spoken. MESSAGE_L AND MESSAGE_S represent constants defined by the script writer to represent long and short versions of the particular message and are placed in a JSM header file for the particular application. Whether the long or short version is spoken depends upon whether the user has selected the long or short JAWS message radio button in the preferences dialog.

Other items in the Items to be Spoken list and their corresponding output types are JAWS Message (OT_JAWS_MESSAGE), Screen Message (OT_SCREEN_MESSAGE), Control Name (OT_CONTROL_NAME), Control Type (OT_CONTROL_TYPE), Dialog Name (OT_DIALOG_NAME), Dialog Text (OT_DIALOG_TEXT), Document Name (OT_DOCUMENT_NAME), Selected Item (OT_SELECTED_ITEM), Item State (OT_ITEM_STATE), Position Information (OT_POSITION), Error Message (OT_ERROR), Item Number (OT_ITEM_NUMBER), Tool Tip

(OT_TOOL_TIP), Status Information (OT_STATUS), Control Group Name (OT_CONTROL_GROUP), Smart Help Messages (OT_SMART_HELP), and Select (OT_SELECT). When constructing scripts which give verbal feedback to the user, care should be taken to use the correct output type constant so that the user's verbosity preferences will be honored.

## 8.10 Synchronizing Documentation

After you do any editing of your script file, especially editing that includes adding or deleting scripts by pasting them in from elsewhere or by highlighting and deleting, it is wise to synchronize the documentation. Every script should have a corresponding documentation entry in the JSD file of the same name. You are, of course, prompted for this information when you create a script, but during the editing process, there might be scripts without documentation entries and documentation entries for scripts that no longer exist. That's why we have the Synchronize Documentation item in the Script Manager File menu. Here are the steps that Script Manager performs when you choose this menu item:

1. The Script Manager looks in the JSD file for a documentation entry for each script in the JSS file and adds a blank entry for each that is missing. (You should go through any scripts that you think might have blank entries and type in the proper Synopsis and Description after the synchronization process.) You will note, when you bring up the documentation dialog, that if you have a script in your application file with the same name as one in the default file, the documentation from the Default script has been placed in the fields for your script. You may need to modify this documentation to match how this script or function works with your application.
2. Script Manager checks the JSD file for any documentation entries that no longer have corresponding scripts in the JSS file and asks if you want to delete the documentation entry.

The synchronization command is also useful when pasting scripts into a script file from a different script file. The process will use the same techniques described above to place blank documentation into the JSD file for the pasted scripts. This should be filled in with the appropriate information.

Using this synchronization process ensures that your scripts and documentation are always current. Remember, JAWS uses the Synopsis and Description for Keyboard Help when the user presses the corresponding key for your script. Therefore, keeping your documentation current will help all users of your scripts.

## 9 Debugging

Volumes have been written about debugging programs. The most important thing to remember is that it takes a different mind set than normal diagnostics. Diagnosing a problem usually starts with the assumption that whatever you are working on was once working and is now broken. Therefore, if you simply follow the logical flow from start to finish, you will find the errant part. Debugging something that hasn't worked yet means that you cannot assume that even the flow is correct. How does this relate to debugging our script file? It's simple. Do not assume that just because you intended for the flow to be a certain way, that it will be. JAWS may not always think the same way you do or, heaven forbid, you may have left out some vital statements. One of the most helpful techniques for finding logical errors is to go through the script line by line, pretending you are the computer. Do exactly what the script says to do, not what you think you told the computer to do. You may find, to your surprise, that you actually gave an instruction that is quite different from what you intended. Correct these mistakes as you find them, and then continue on through the script, checking each statement to see if it's really doing what you thought. Here are some other tools that can help you in the debugging effort.

### 9.1 The JAWS Script Compiler

The first big help is the script compiler. It checks the syntax of your statements and also makes sure that you complete each sequence correctly. It displays an error message when you try to compile a script with a mistake in it. Those of you who have written macros for JAWS in the past will notice that the compiler errors for scripts are more descriptive. After you dismiss the error dialog, the cursor will be placed in the script near the error or what the compiler thinks is the error. You must understand that the compiler can not always know exactly where the mistake is. If you use several If-Then statements that are nested, for example, and you forget to use the correct number of EndIf statements, the compiler will flag an error, but the cursor may not be located where the missing EndIf statement should have been. The cursor may be several lines away from that location. The compiler can only tell you that your If- Then loops are not formulated properly and that an If statement must be followed by an EndIf statement. It's up to you to find out where you intended to put that missing  statement.

Include files are examined as they are included.  Always  place the include statement in your script file before you  use anything in the include file.  For instance, you must  include HJCONST.JSH before using any of the constants in it.  Of course, if you are laying your script file out properly,  all of your includes will be near the beginning of the  script file anyway.

## 9.2 Erroneous Activity

By erroneous activity, we mean the script is working (i.e.,  it has compiled properly) but may not be working as  intended.  Once your script file compiles without error,  it's time to test it in its intended environment.  It is not  necessary to restart JAWS before testing, unless you are modifying the default script file.  Just switch to the  application and test.  Here are a few tips to make the  testing process easier:

? Fragments - Whenever possible, compile fragments  separately and test to see if they work by themselves.

? Messages may be inserted at crucial points in your code  so JAWS tells you what it's doing.  You can put Say  messages into crucial parts of a script to see if they're  really executing when you think they should be.  A series  of these messages that say slightly different things can be very informative in helping you decipher if the code  is being executed in the order you intended and branching  is occurring correctly.  For instance, Say messages in  each leg of an If statement will inform you of which leg  was executed.  You can also use the SayInteger function to inform you of the values of integer variables and how  these values are changing during script execution.  You  may find that these variables do not have the values you  expected at certain points during script execution.  The  SayInteger function may also be useful in helping you  keep track of integer variable values during the  execution of a While loop.

? Many errors of operation can be traced to the failure to  save and restore states properly. This refers to cursor  states, screen echo states, verbosity states, etc.  Anytime you have to change a state for your purposes, be  sure to return to the original state.  For instance,  changing screen echo to All so that you can hear a help  bubble spoken is fine, but don't leave the system set that way as it will be very confusing to the user.

? Debug Scripts - You can create debug scripts that report all sorts of information like which cursor is active, what's contained in variables, and various window properties. Here are a couple of scripts you can copy and paste into your script file and modify to report information back to you.

```
          Script SayVars
          Say ( MyVariable, OT_DEBUG) ; Fill in the name of your  variable
          ; Copy the preceding line for each of your variables  EndScript

          Script WindowInfo
          Say ( GetWindowClass (GetFocus ()), OT_DEBUG) ; Window  class of focus control
          Say ( GetWindowClass (GetParent (GetFocus ())), OT_DEBUG)  ; Window class of the
parent
          UtilitySayInfoAccess() ; Control name  EndScript
```

? Interactions of Your Scripts with Event Functions - You must keep in mind that JAWS has the event functions that are present in the default script file which may trigger automatically as an application is running. These event functions can interact in unexpected ways with the scripts you are writing. For example, you may place reading functions in your script to read certain parts of the screen at certain times, only to find that the information is being spoken twice. This probably means that one of the event functions such as the NewTextEvent function or the Focused Changed Event function is being called to speak the text automatically, and your script is saying it a second time. You won't find the problem merely by analyzing your script since it isn't your script which is speaking the information twice. That's why it's important for you to have an understanding of what the JAWS event functions do and when they will be called by an application.

Homework Assignment #10

The following script contains an error on each line. Identify those errors and rewrite the script correctly. The answer can be found in Appendix A. (See Chapter 14.9.)

```
Script SayDefaultButton (Int Button )  vars
```

```
          string DefaultButtonName,  if (DialogActive ())
          let DefaultButtonName == GetDefaultButtonName  if (DefaultButtonName != " ") then
                    if (GetVerbosity () = 0) then
                              Say ("msg70", OT_JAWS_MESSAGE) ;"default  button is"
                    EndIf ()
                    Say (DefaultButtonName), OT_CONTROL_NAME)  Else ()
                    Say (msg444, OT_JAWS_MESSAGE) : "Can not determine  default button in this
dialog box "
          End If
Else ()
          Say (msg71, OT_JAWS_MESSAGE ;"not in a dialog box"  EndIf ()
End Script
```

## 10 Strategies for Attacking New Applications

When you first open a new application, especially one which isn't "speech friendly," it can be very difficult to decide how to go about learning the application layout and how to customize it for better access. This manual has introduced you to the scripting tools that will help you do this. One strategy for approaching these new applications is shown in Appendix E. Here you will find some ideas on how to approach the problem of analyzing and scripting a new application in combination with the use of other JAWS features such as the use of frames.

## 11 Guidelines for Creating Distribution Script Files

We have suggested numerous guidelines to observe when creating script files, such as using plentiful comments and filling out the script documentation forms properly. Of course, these are recommendations, and nobody can force you to follow these rules.

? Variable Names - Use meaningful variable names followed by a comment when the name is not a sufficient description.
? Comments - Place comments immediately after the beginning of each script or function describing its purpose. Comments should also be placed with in complex statements where the purpose is not obvious such as complex "If- Then-Else" statements".
? Messages - All spoken or Braille messages should be put into the message file for the current application where they can be referenced by the script. These messages

should never be placed with in quotes following the Say  or SayMessage in the JSS file itself. Obeying this rule  makes translation of the messages to other languages much  simpler.
?	Script Documentation - Fill in the synopsis and  description fields in the new script dialog. These  should be accurately and completely filled in as they are  used in the INSERT+1 keyboard help.  For example, in the  script SayTextAndAttributes, you will find the following  synopsis and description:
	Synopsis - Speaks changes in attributes as it reads the  text in the active window.
	Description Reads the visible text in the active window  without moving the cursor.  As it passes over changes  in attributes, they are announced.
?	Make sure your script file includes customized versions  of the HotKeyHelp, the ScriptFileName, and if necessary,  the ScreenSensitiveHelp scripts.

## 12	Converting Macro Files

You can use the Script Manager to convert macro files that  you created for versions of JAWS before 3.0.  The following  describes the steps that Script Manager uses to convert  macro files to script files.
1.	First, Script Manager requires some input from you.  When  you choose Import Macro File from the File menu, you must  enter the file name of the macro file.  Script Manager  uses this information to determine the path and file name  of the resulting script file.  You must also choose the  environment for which the macro file was created:  Desktop, Laptop, or Both.
2.	 Script Manager creates an empty script file based on the  target path and file name.
3.	Include files are processed as follows:
?	A message file is copied to the target directory,  renamed with a JSM extension and the include  statement is added to the JSS file.
?	Macros in include files are moved to the JSS file  and converted.  The conversion procedure is as  follows.
?	If there is a global variable or constant definition  section in an include file, a JSH file is created  and these are placed there.  An include statement  for this header file is added to the JSS file.
4.	Macros are converted as follows:  If a matching keystroke  is found in the default key map file based on the chosen  environment, the script is named appropriately.
?	If a matching keystroke is found in the "Old Keys"  section of the default key map file, the macro is

converted to a JAWS function with the appropriate  name.
? If no match is found, the script is named with the  file name followed by a number.
5. A key map file is created with assignments based on the  original macro keystrokes.  Macros that are converted to  user-defined functions are not given key assignments, but  a comment is placed after the function name with the  original macro keystroke.
6. Each PerformMacro is converted to a PerformScript, unless  the called macro was converted to a JAWS function.  In  that case the PerformMacro is converted to a function  call.
7. The resulting script file is opened in Script Manager.
8. The previous version of JAWS required a space between  quotes to denote a null value.  Nulls in the current  version are denoted by just two quotes with no space in  between.  If your macro file contained these tests for  nulls, you will need to remove the space between quotes manually.
9. The last step is for you to go through the scripts,  change the names, if necessary, and write documentation  for each.  If you had macros which served as functions,  convert them to user-defined functions.  Then compile the  new script file.
Note: If you are making extensive changes to the scripts and  functions, you may want to compile after each change.

## 13 Acknowledgments

I would like to express my sincere appreciation to Glen  Gordon, Mike Pedersen, Joseph Dunn, Sean Murphy, Glen Sepke,  James Puzzuoli, Les Kriegler, and Dusty Vorhees for their tremendously helpful technical information, ideas, and  suggestions.  I would also like to acknowledge Frank  Dipalermo who originally wrote the first macro and script  manuals which served as the starting point for this project.

## 14 Appendix A

Answers to Homework Assignments

Answer to Homework Assignment #1

```
Script LastFourFiles ()
SpeechOff ()
{Alt+F}
Pause ()
NextLine ()
NextLine ()
```

NextLine () NextLine () NextLine () NextLine () NextLine () SpeechOn () SayLine () NextLine () SayLine () NextLine () SayLine () NextLine () SayLine () SpeechOff () {escape} {escape} Pause () SpeechOn () EndScript

**********

14.1 Answer to Homework Assignment #2

Script SpeakWindowState () JAWSCursor () SaveCursor () RouteJAWSToPC () JAWSPageUp () JAWSEnd () PriorWord () SayWord () RestoreCursor () PCCursor () EndScript

Note that in this example we activated the JAWS cursor before using the SaveCursor function. Since the JAWS cursor is the active cursor when the SaveCursor function is performed, it is the JAWS cursor which will be returned to its original position by the RestoreCursor function. The PCCursor () function at the end will turn the PC cursor back on. This is necessary since the script turned on the JAWS cursor when it was run.

The extra credit answer is shown below:

Script SpeakWindowState () InvisibleCursor () SaveCursor ()

```
RouteInvisibleToPC ()  JAWSPageUp ()  JAWSEnd ()  PriorWord ()  If GetWord () == "Restore
Symbol" Then  Say ("Maximized", OT_STATUS)  ElIf GetWord () == "Maximize Symbol" Then
Say ("Restored", OT_STATUS)  Else  Say ("Couldn't find the symbol", OT_JAWS_MESSAGE)
EndIf  RestoreCursor ()  PCCursor ()  EndScript
```

******

## 14.2 Answer to Homework Assignment #3

```
Script PlaybackPosition ()  SaveCursor ()  RouteInvisibleToPC ()  InvisibleCursor ()
JAWSPageUp ()  NextLine ()  NextLine ()  NextLine ()  JAWSHome ()  SayMessage
(OT_JAWS_MESSAGE, "Current time is", "Time is")  SayWord ()  SayMessage
(OT_JAWS_MESSAGE, "seconds out of" )  NextWord ()  NextWord ()  SayWord ()  SayMessage
(OT_JAWS_MESSAGE, "seconds" )  RestoreCursor ()  EndScript
```

******

14.3 Answer to homework Assignment #4  Include "JAWSWINS.JSH"  Include
"JAWSWINS.JMH"  Include "HJCONST.JSH"  Include "HJGLOBAL.JSH"

******

## 14.4 Answer to Homework Assignment #5

```
  Script CheckAndReset ()  Var  String TotalTime ;Holds the value of the total time of the  sound
file  SaveCursor ()  RouteInvisibleToPC ()  InvisibleCursor ()  JAWSPageUp ()  NextLine ()
NextLine ()  NextLine ()  JAWSEnd ()  PriorWord ()  Let TotalTime = GetWord ()  JAWSHome ()  If
GetWord () == "0.00" Then  SayMessage (OT_JAWS_MESSAGE, "At the beginning of  the
recording", )  ElIf GetWord () != "0.00" && GetWord () < TotalTime  Then  SayMessage
(OT_JAWS_MESSAGE, "Playback only  partially complete" )  ElIf GetWord () == TotalTime Then
SayMessage (OT_JAWS_MESSAGE, "Playback is  complete." )  JAWSCursor ()
RouteJAWSToPC ()  JAWSHome ()  Pause ()  LeftMouseButton ()  Pause ()  InvisibleCursor ()  If
GetWord () == "0.00" Then  SayMessage (OT_JAWS_MESSAGE, "Rewind  complete" )  Else
SayMessage (OT_JAWS_MESSAGE, "Rewind  failed.  Please check your batteries." )  EndIf
EndIf  EndScript
```

Note - You may hear JAWS say some additional phrases like  "Seek to Start" or "Seek to End" when it performs the  rewind.  This is Okay.

******

## 14.5 Answer to Homework Assignment #6

```
  String Function SayAllCaps (string CurrentWord)  If CurrentWord < "a" || CurrentWord > "zzzzz"
Then  Return ""
EndIf
If (StringContains (CurrentWord, "a")) ||  (StringContains (CurrentWord, "b"))
 || (StringContains (CurrentWord, "c")) ||  (StringContains (CurrentWord, "d"))  || (StringContains
(CurrentWord, "e")) ||  (StringContains (CurrentWord, "f"))  || (StringContains (CurrentWord, "g")) ||
(StringContains (CurrentWord, "h"))  || (StringContains (CurrentWord, "i")) ||  (StringContains
(CurrentWord, "j"))  || (StringContains (CurrentWord, "k")) ||  (StringContains (CurrentWord, "l"))  ||
(StringContains (CurrentWord, "m")) ||  (StringContains (CurrentWord, "n"))  || (StringContains
(CurrentWord, "o")) ||  (StringContains (CurrentWord, "p"))  || (StringContains (CurrentWord, "q"))
||  (StringContains (CurrentWord, "r"))  || (StringContains (CurrentWord, "s")) ||  (StringContains
(CurrentWord, "t"))  || (StringContains (CurrentWord, "u")) ||  (StringContains (CurrentWord, "v"))
|| (StringContains (CurrentWord, "w")) ||  (StringContains (CurrentWord, "x"))  || (StringContains
(CurrentWord, "y")) ||  (StringContains (CurrentWord, "z"))  || (StringContains (CurrentWord, "1"))
||  (StringContains (CurrentWord, "2"))  || (StringContains (CurrentWord, "3")) ||  (StringContains
(CurrentWord, "4"))  || (StringContains (CurrentWord, "5")) ||  (StringContains (CurrentWord, "6"))
|| (StringContains (CurrentWord, "7")) ||  (StringContains (CurrentWord, "8"))  || (StringContains
(CurrentWord, "9")) ||  (StringContains (CurrentWord, "0")) Then  Return ""
EndIf
Return "All Caps"
EndFunction

Script SayWord()
If (IsSameScript ()) Then
Say (SayAllCaps (GetWord ()), OT_MESSAGE)  SpellWord()
```

AddHook (HK_SCRIPT, "SpellWordHook")  Else
SayWord ()
EndIf
EndScript

******

## 14.6 Answer to Homework Assignment #7

Edit Window:

SayTypeAndText - Edit
Focus - This is a window handle, so it is always changing  ControlID - 59648
Class - Edit
Type - RichEdit
SubTypeCode - 3
Real Name - Document WordPad (assuming you have not loaded a  file)

Filename Window:

SayTypeAndText - Filename Edit  Focus - This is a window handle, so it is always changing
ControlID - 1152
Class - Edit
Type - Edit
SubTypeCode - 3
Real Name Open

Note - If you look in HJCONST.JSH, you will find that a  SubTypeCode of 3 is equal to an edit window.  In these  cases, the SubTypeCode does not provide any more information  than the class.  Also, note that the two windows have  different ControlID numbers, even though they are both edit  windows.

******

## 14.7 Answer to Homework Assignment #8

;This is the script file for SPAMKILLER.EXE, created by  Michael Glen
;This file was last updated on July 4, 1776.

Include "SPAMKILLER.JSH" ;Header file  Include "SPAMKILLER.JSM" ;Message file  Include "HJCONST.JSH"  Include "HJGLOBAL.JSH"

Globals
Int GlobalSpamInteger,  String GlobalSpamString,  Handle GlobalSpamHandle

Const
SpammerOne = 1
SpammerTwo = 2
SpammerThree =3

******

14.8 Answer to Homework Assignment #9

Script IsThisOpenEdit ()
If GetWindowName (GetRealWindow (GetFocus ())) ==  "Open" Then
Say ("In the Open dialog,", OT_MESSAGE)  If GetWindowClass (GetFocus ()) == "Edit"  Then
Say ("and in the filename field.",  OT_MESSAGE)
Else
Say ("but not in the filename field.",  OT_MESSAGE)
EndIf ;Are we in the edit field?  Else
Say ("Not in the Open dialog", OT_MESSAGE)  EndIf ;Are we in the Open dialog?  EndScript

******

14.9 Answer to Homework Assignment #10

Each error is indicated below by a line starting with ***  after the line containing the error.

Script SayDefaultButton (Int Button )  ***Scripts do not use parameters, only functions use them.
vars
***It should be Var, not Vars  string DefaultButtonName,  ***The last or only variable should not have a comma after  it.  if (DialogActive ())  ***There should be a "Then" at the end of the line.
    let DefaultButtonName == GetDefaultButtonName  ***In this type of statement, where you are assigning a

value, there should only be one =.  if (DefaultButtonName != " ") then  ***There should not be a space between the quotation marks.

```
                    if (GetVerbosity () = 0) then  ***There should be two = (i.e., ==) in this statement.
                              Say ("msg70", OT_MESSAGE) ;"default button  is"
```

***There should not be any quotes inside the  parentheses.  This would say MSG70 instead of reciting the actual message.

```
                    EndIf ()
```

***No parentheses are used after an EndIf.  Say (DefaultButtonName, OT_CONTROL_NAME))
***There is an extra ) at the end.

```
          Else ()
```

***There should not be a () after an Else statement.  Say (msg444, OT_MESSAGE) : "Can not determine  default button in this dialog box "  ***The comment should begin with a semicolon, not a colon.

```
          End If
```

***There is a space in the EndIf that shouldn't be there.  Else ()
***No () after an Else statement.  Say (msg71, OT_MESSAGE ;"not in a dialog box"  ***There should be a ) after OT_MESSAGE.  EndIf ()
***No () after an EndIf statement.  End Script
***There should not be a space in EndScript.

The script as it should be and as it appears in DEFAULT.JSS  is shown below.

```
Script SayDefaultButton ()  var
          string DefaultButtonName  if (DialogActive ()) then
          let DefaultButtonName = GetDefaultButtonName ()  if (DefaultButtonName != "") then
                    if (GetVerbosity () == 0) then  Say (msg70), OT_MESSAGE ;"default button is"
                    EndIf
                    Say (DefaultButtonName, OT_CONTROL_NAME)  else
                    Say (msg444, OT_MESSAGE) ; "Can not determine  default button in this dialog
box ")
          EndIf  else
          Say (msg71, OT_MESSAGE) ;"not in a dialog box"  EndIf
```

EndScript

# 15 Appendix B

Description of the Script Manager Menus

Here is a brief description of the selections available under the various pull down menus of the menu bar of the Script Manager. Only items which are unique to the Script Manager or perform unusual functions are described. Others which are standard to Windows applications are simply noted as "standard function." Many of the functions discussed have accelerator keys which are listed in the menus and which you will probably want to learn.

## 15.1 File Menu

New: standard function
Open: standard function
Open Default File: opens the default JAWS script file, DEFAULT.JSS.
Close: standard function
Save: Saves the file, but also compiles and saves a binary JSB version if saving a JSS file.
Save As: standard function
Save Without Compile: saves any file and does not compile a JSS file.
Import Macro File: used to import a macro file from earlier versions of JAWS.
Synchronize Documentation: Used to make sure the script and documentation files concur. Also, when pasting in scripts from another JSS file, it will copy documentation into the related JSD file. For more information, see "Synchronizing Documentation" under Script Writing Techniques.
Print: standard function
Print Preview: standard function
Print Setup: standard function
Exit: standard function

## 15.2 Edit Menu

All functions of the edit menu are entirely standard, except the one listed below.

Select Script: Selects the entire script in which the caret is located.

## 15.3 Script Menu

New Script: Opens the New Script dialog for creating a new  script.
Delete Script: Deletes the script where the caret is  located.
Insert Function Call: Brings up the Insert Function dialog  for adding a function to a script.  Insert Perform Script: Brings up the Insert Perform  Script dialog for calling another script.  This is similar to calling a function, but it calls another  key-bound script instead.
Next Script: Moves the caret to the beginning of the next  script in the file.
Prior Script: Moves the caret to the beginning of the  current script or the prior script, depending on caret  location.
Go To Line: Moves the caret to a specific line  number.  Line numbers are shown at the page bottom.
Script List: Brings up the Script List dialog  which shows the scripts in the current file in alphabetical order.  Pressing ENTER on a script  name takes you to that script.

## 15.4 View Menu

Documentation: Brings up the Script Information  dialog which contains all of the documentation for  the current script.
Toolbar: standard function
Status Bar: standard function
Zoom: standard function

## 15.5 Window Menu

All items in the Window menu perform standard  functions.

## 15.6 Help Menu

Help Topics: standard function  keyword Help: Brings up the Keyword Help dialog  which contains help for the script or function on  which the caret is located.
About Script Manager: standard function

# 16 Appendix C

The Most Important Built-In Functions

Note - Most of the entries in the following list are in alphabetical order.  A few were moved slightly out of order,  however, to place them near functions with similar or  related functionality.  This list has been updated to  reflect new functions added in JAWS 3.7.  Those using  earlier versions will not have all of the functions shown  here.

? ActivateMenuBar - Activates/deactivates the menu bar for  the active program.  Same as pressing and releasing the  ALT key.
? ActivateStartMenu - Brings up the Start menu in Windows  95/98 or NT.  This is the same as pressing CTRL+ESC.
? AddHook - Installs a hook function.  When a hook is in  place, it is called right before every script is run, and  passed the name of the current script and frame as its  parameters.  If the hook returns TRUE, the script is  allowed to execute.  If the hook returns FALSE, the  script will not be allowed to run.  See the KeyboardHelp  script and the KeyboardHelpHook function in DEFAULT.JSS  for an example of a hook function in action.  This tool  is discussed more fully in the section on Hook Functions.
? AltLeftMouseClick - Unselects all previously selected  items.  Sends an ALT+LEFT MOUSE BUTTON to the system.
? ControlLeftMouseClick - Simulates a CTRL+Left click of  the mouse.  This script provides a method for selecting  items non-contiguously.  An example is selecting items  non-contiguously in a list view.
? ShiftLeftMouseClick - Simulates a SHIFT+Left click of the  mouse.  This function provides a method for selecting  items contiguously.  An example is selecting items  contiguously in a list view or an edit control.
? BackspaceKey - If the virtual cursor is not active, this  function simply passes the ENTER key through to the  application.  If the Virtual Cursor is active, any  special processing required by virtual mode is performed.
? BrailleAddFocusItem - Used from within the  BrailleBuildLine function to ad the contents of the  current control to the data to be shown on the Braille  display.
? BrailleAddFocusLine - Used from within the  BrailleBuildLine function to ad the contents of the  current line to the data to be shown on the Braille  display.
? BrailleAddFrame - Used from within the BrailleBuildLine  function to add the contents of a specific frame to the  data to be shown on the braille display.
? BrailleAddString - Used with in BrailleBuildLine to add  text to the Braille display
? BrailleCursor - Turns on the Braille cursor.  This cursor  is only used internally by the Braille scripts and should

never be left on after a script completes its work.
? BrailleG2StringLength - Determines the length of the translation to grade two Braille of a given string.
? BrailleNextLine - If Structured mode is active, The next string in the braille queue is displayed. If Navigation mode is active, JAWS activates the Invisible cursor and moves it to the next line. Otherwise, a normal next line function is performed.
? BraillePriorLine - If Structured mode is active, The prior string in the braille queue is displayed. If Navigation mode is active, JAWS activates the Invisible cursor and moves it to the prior line. Otherwise, a normal prior line function is performed.
? BrailleRefresh - Used to refresh the braille display and redisplay the appropriate text.
? BrailleSetStatusCells - Puts characters on the status cells of the Braille Display. This function should only be called from within a BrailleBuildStatus function.
? BrailleString - Sends the specified string to the braille display regardless of whether or not the line is visible.
? CaretVisible - Indicates whether an insertion point or a caret is visible in the active window. The function returns a constant value of "TRUE" to indicate that a caret is visible and a value of "FALSE" to indicate that it is not visible.
? ClipboardHasData - This function checks to see if any data at all is present on the Windows clipboard. It will return true if there is such data present.
? ColorToRGBString - Converts a color value to a string of the form "255255255" where the first three digits represent the proportion of red, the second three the proportion of green, and the third three the proportion of blue.
? ControlCanBeChecked - Determines whether or not the current control can be checked.
? ControlIsChecked - Determines whether or not the current control is checked.
? CopyToClipboard - Puts a string of text onto the Windows clipboard erasing any previous clipboard contents. To copy multiple lines of text, these must be concatenated together into a single string before calling CopyToClipboard. In the concatenated string, lines should be separated with \r\n.
? CreateObject - For certain applications (e.g., Internet Explorer and Microsoft Office ) it is possible for JAWS to obtain information directly from that application rather than relying upon what is displayed on the screen. To do this, JAWS needs a place to go to get that information and a sort of "road map" to tell it where to look. The road map is called an object pointer, and the

place it directs JAWS to is called an automation object  for that application.  There are several ways of  obtaining such an object pointer for this purpose, not  all of which work with all applications.  One such method  is CreateObject.  CreateObject launches an application  under the control of JAWS, with that application being  the automation object.  JAWS can then query that  automation object to get the information it needs.  The  CreateObject function is similar to the GetObject  function, except that GetObject creates a pointer to an  automation object which already exists whereas  CreateObject also creates the automation object for the  application.
?	Delay - Makes a script stand still for a specified period  of time.  It causes a script to stop, wait a period of  time, and then resume again.  It is different than the  Pause function which yields to the processing needs of  applications.  After the delay, the script will resume  regardless of whether an application is still processing.
?	DialogActive - Used to determine whether a dialog box is  currently active.  It returns a constant value to  indicate the status:  "ACTIVE" = a dialog box is active,  and "INACTIVE" = a dialog box is not active.  These  constants can be used in If- Then-Else statements.
?	DLGSelectItemInList - Displays a dialog that contains a  listbox of items.  The contents of the list are built by  the user who must supply a string containing each of the  list entries separated by a vertical bar (|).  This  function is normally used within another script or  function which specifies what actions are to follow when  an item is selected and the dialog is okayed.  When the  dialog is okayed, this function returns an index that  allows the calling script to decide what action to take  next.  See the function named ToolBar in BROWSEUI.JSS for  an example of how this function is used.
?	DLGSelectControls - Displays a list box containing  controls that can b activated.  The four possible actions  are right single click, left single click, left double  click, and move to.  The user can specify which of these  four buttons are present as well as which is the default.  The list of controls must be built by the user.  The  window title for the dialog can also be specified.  This function is similar to DLGSelectItemInList except that it  provides a wider selection of actions that can be  performed on the selected list item.
?	DLGSelectScriptToRun - Displays a dialog that contains a  set of scripts.  The list of scripts is built by the  user.  See the script titled AdjustJawsVerbosity for an  example of how this is done.  Scripts can be performed  from this dialog.

? DLGSysTray - Displays a list box containing the icons on  the system tray.
? DownCell - When inside a table or spreadsheet, moves the  active cursor to the cell in the same column but the next  Row.
? UpCell - When inside a table or spreadsheet, moves the  active cursor to the cell in the same column but the  previous Row.
? PriorCell - When inside a table or spreadsheet, moves the  active cursor to the cell in the same row but the  previous column.
? NextCell - When inside a table or spreadsheet, moves the  active cursor to the cell in the same row but the next  column.
? SayCell - When in a table or spreadsheet, speaks the  contents of the current cell.
? SayColumnHeader - When in a table or spreadsheet, speaks  the contents of the column header.
? SayRowHeader - When in a table or spreadsheet, speaks the  contents of the row header.
? DragItemWithMouse - The DragAndDrop function is used to  move the contents of one area of the screen to another  area of the screen.
? ElIf - Instead of using an Else operator in an If-Then-  Else statement, you may want to use an ElIf operator.  The statement could then be reconstructed as an If-Then-  ElIf-Then-Else statement.  Every statement must end with  an "EndIf" operator.  The script functions that appear between the ElIf and the Then are used to evaluate  whether a certain condition is present.
? Else - The Else operator is an optional part of an If-  Then- Else statement.  If-Then-Else statements always ask  a question about whether something is true or false,  i.e., is a condition present or not present.  When the  condition is not present (false), then the actions that  follow the "Else" are performed.  An If-Then-Else  statement must always include:  If, Then, and EndIf --  the Else is optional.
? EndIf - The EndIf operator marks the end of an If-Then-  Else statement.  A fully formulated If-Then-Else  statement includes "If", "Then", "Else", and "EndIf".  The "EndIf" is always required to terminate the If-Then-  Else statement.
? EndWhile - The EndWhile operator marks the end of a  "While Loop".  The script functions that are to be  repeated must be placed between "While" and "EndWhile".  (See description of the While function below.)
? EnterKey - If the virtual cursor is not active, this  function simply passes the ENTER key through to the  application.  If the Virtual Cursor is active and

positioned on a link or button, that control is activated. If the virtual cursor is on another form control, Forms Mode is activated.
? ExMessageBox - This function displays a Windows Standard Message Box wherein the user can specify the message, title, and buttons which can be activated. It is very much like the function MessageBox except it allows you to specify the Message Box title and the type of message box (i.e., the buttons that are used and the icon that is to be displayed). This function also returns a value that indicates which button was pressed on the Message Box.
? MessageBox - Used to Make a message box pop up on the desktop. It can be included within an If-Then-Else statement so that when a certain event occurs, the needed message pops up.
? InputBox - This function displays a simple dialog box containing four controls. One control is an edit box in which you can enter information. Another control is a Static Text window containing the prompt of the edit box. The other two controls are the OK and Cancel buttons. If you enter text into the Edit box and press the OK button, the text you typed in the Edit box is returned to the calling function by way of the third variable of the function, which is a string variable that is passed by reference to the calling function.
? FileExists - Checks whether the specified path exists.
? FindColors - Searches for the occurrence of a specific combination of foreground and background colors. If the search is successful, the JAWS cursor is placed at the beginning of the text with the desired combination of colors.
? FindDescendentWindow - A descendent window is any child window of some specified parent window. It can be a child at any level below the parent and in any branch. This function searches down the branches from the parent to find any child window (descendent window) with a specified control ID. FindFirstAttribute - Searches for the first occurrence of text with certain attributes. It begins the search at the upper left corner of the specified window and moves downward. If the search is successful, then the active cursor is placed on the first character that has the desired attributes. Generally, the PC cursor can be successfully moved to attributes within a text window.
? FindNextAttribute - Searches for the next occurrence of text with certain attributes. The search begins at the location of the active cursor and moves down through the remainder of the active window. If the search is successful, then the active cursor is placed on the first character that has the desired attribute.

? FindPriorAttribute - Searches for the prior occurrence of text with certain attributes. It begins the search at the location of the active cursor and moves up through the remainder of the active window. If the search is successful, then the active cursor is placed on the first character that has the desired attributes.

? FindLastAttribute - Searches for the last occurrence of text with certain attributes. It begins the search at the lower right corner of the active window and moves up to the upper left corner of the window. If the search is successful, then the active cursor is placed on the first character that has the desired attributes.

? FindGraphic - FindGraphic searches for a graphic in the specified window. If the graphic is found, then the JAWS cursor is placed on it. The graphic must have a text label associated with it, because the FindGraphic function searches for text labels. Text labels are assigned by the Graphics Labeler.

? FindString - FindString searches for a string of text in a specified window. If the text is found, then the JAWS cursor is placed at the beginning of the text string.

? FindTopLevelWindow - Find the top level window with the specified window class and/or window name. If you do not wish to search based on one of the arguments, use "" for that argument.

? FormatString - Formats a string supplied by the user with messages received as subsequent parameters. That is to say, a user's string can be modified to contain new strings at certain specified locations within the original string. The message to be formatted must contain parameter place holder delimiters such as %1, %2, %3, %4, %5, etc. The user must supply the various strings which are to be inserted in the original string at the locations of each place holder.

? GetActiveCursor - Determines which cursor is active.

? GetAppFileName - Determines the filename of the active application program or an active component of that program that was subsequently executed. These filenames are the actual program files that are executed by Windows. The application's filename is used when naming script files.

? GetAppFilePath - Used to get the fully qualified path name of the currently running application. When used in combination with GetFileDate, the results can be used to surmise an application version.

? GetAppMainWindow - Determines the window handle for the main window of the active application.

? GetAppTitle - Obtains the title of the active application program. To have the application title spoken, you must use this function as a parameter for the Say function.

? GetBrailleCellColumn - Retrieves the screen column at which the specified Braille cell is located.
? GetBrailleCellRow - Retrieves the screen row at which the specified Braille cell is located.
? GetCell - When in a table or spreadsheet, gets the contents of the current cell.
? GetColumnHeader - When in a table or spreadsheet, gets the contents of the column header.
? GetRowHeader - When in a table or spreadsheet, gets the contents of the row header.
? GetCharacter - Copies the character or graphic label where the active cursor is positioned. The copied text can then be used by other script functions.
? GetCharacterAttributes - Retrieves the text attributes of the character at the current cursor location. The returned value is a combination of the same bit fields used in FindFirstAttribute, FindNextAttribute, etc. To test for the presence of a particular attribute, use code of the form: if (GetAttributes() & ATTRIB_UNDERLINE) then ... (See the section on bitwise operators for a more thorough description.)
? GetCharacterFont - Retrieves the name of the font used for the character at the current cursor location.
? GetCharacterPoints - Retrieves the point size of the character at the current cursor location.
? GetCharacterWidth - Retrieves the width of the character or graphic at the active cursor location.
? GetChunk - Obtains the chunk of information to which the active cursor is pointing. A "chunk" is text and graphic information that was written to the screen in a single operation. GetChunk is similar to GetField, however, the GetField function uses logic to determine the text that is to be obtained, while GetChunk simply obtains the text that was stored in the Off Screen Model as a single unit.
? GetField - Obtains the information in the field where the active cursor is pointing. A "field is a section of information (usually text) that has a common attribute, i.e., bold, underlined, italics, or strikeout. The use of the attribute must be contiguous. GetField is similar to GetChunk, however, the GetField function uses logic to determine the text that is to be obtained, while GetChunk simply obtains the text that was stored in the Off Screen Model as a unit.
? GetColorBackground - Retrieves the background color of the character at the current cursor location. These colors are normally specified as 9 digit numbers where the first three specify the red contribution, the next three the green contribution, and the last three the blue contribution.
? GetColorText - Retrieves the text color of the character

at the current cursor location.
? GetColorField - Obtains the color field at the position  of the active cursor.  A "color field" is a section of  information (usually text) that has a common combination  of colors, i.e., white foreground on blue background.  The use of the color combination must be contiguous.  GetColorField is similar to GetField, however, the  function GetColorField is based on color changes, while  GetField is based on attribute changes.
? GetColorName - Color numbers are equated to color names  in a JAWS file called COLORS.INI.  This function  retrieves the textual name for the given color number  found in colors.ini.
? GetControlAttributes - Returns an integer value  indicating the attributes of the current control.
? GetControlID - Determines the Control ID for the  specified child window in a dialog box.  The window of  interest is specified by passing its window handle to  this function as a parameter.  Each list box, edit field,  radio button, etc. in a dialog has a unique control ID  number.  Child windows that contain static text all have  the same control ID.
? GetCurrentControlID - Determines the control ID of the  active child window in a dialog box.  Each list box, edit  field, radio button, etc. in a dialog has a unique  control ID number.  Child windows that contain static  text all have the same control ID.  This function  performs the same task as GetControlID but it does not  require a window handle as a parameter to tell it which  child window is of interest.
? GetControlName - gets the name of the current control.
? GetCurrentObject - Uses MSAA to obtain the object with  focus at the active cursor position.
? GetFocusObject - Uses MSAA to get the object with focus.
? GetObjectAtPoint - Uses MSAA to determine the object at a  set of X/Y coordinates.
? GetCurrentScriptKeyName - Retrieves the name of the key  used to invoke the currently active script.
? GetCurrentWindow - Determines the window handle for the  window that contains the active cursor.  In contrast, the  GetFocus function uses an analytic process to find the  window that currently has the focus regardless of which  cursor is active.
? GetFocus - Obtains the window handle for the window that  has "the focus".  It always seeks to find the PC cursor  or highlighted item that has the focus.  It does not take  into account which cursor is active.  In contrast, the  GetCurrentWindow function is less sophisticated.  It simply obtains the handle for the window in which the  active cursor is located.

? GetCursorCol - Determines the horizontal position or column where the active cursor is located. It returns an integer which can be spoken with the SayInteger script function.
? GetCursorRow - Determines the vertical position or row where the active cursor is located. It returns an integer which can be spoken by the SayInteger script function.
? GetCursorShape - Gets the current shape of the mouse cursor
? GetDefaultButtonName - Identifies the default button in a dialog box. This is the button that will be chosen when ENTER is pressed.
? GetDefaultJcfOption - Determines the value of a specified option in the default JAWS configuration file. These are values such as user verbosity or typing echo set in the Configuration Manager.
? GetJCFOption - Determines the value of a specified Configuration Manager option in the JAWS configuration file for the active application, if any.
? GetJFWSerialNumber - Retrieves the serial number for the currently-running copy of JAWS.
? GetDialogPageName - If the active cursor is inside a multi-page dialog, retrieves the name of the current page.
? GetDialogStaticText - Gets the static text in a dialog box.
? GetFileDate - Used to get the last modified date of a particular file.
? GetFirstChild - Determines the first child window that may have been created by a specific parent window. This function is useful when you wish to move down through the stack of window handles to find children of your starting window.
? GetParent - Moves up through the window stack to find the parent of the specified child window. It can, thus, be used to Determine which window created a specified child window. For example, when a dialog box pops up, it could be used to determine the window handle of the window that created the dialog box. It can be used to move up through a list of window handles in order to get to a specific window.
? GetFirstWindow - Provides the handle for the first window that is at the same logical (child) level as a specified window handle. It is used to get to the beginning point of a series of windows at the same logical level.
? GetNextWindow - Similar to GetFirstWindow, but it provides the next window handle in a series of window handles that are all at the same logical level. It is used to move across a list of window handles.

? GetPriorWindow - Similar to GetFirstWindow, but it provides the prior window handle in a series of window handles at the same logical level. It is used to move across a list of window handles.

? GetForegroundWindow - Retrieves the handle to the current Foreground window. This is generally the main window of the active application but can sometimes be the handle of a dialog box. This function is marginally faster than GetAppMainWindow.

? GetFocusRect - Gets all of the coordinates of a focus rectangle.

? GetFocusRectBottom - Gets the coordinates of the bottom edge of a focus rectangle.

? GetFocusRectLeft -Gets the coordinates of the left edge of a focused rectangle.

? GetFocusRectRight - Gets the coordinates of the right edge of a focus rectangle.

? GetFocusRectTop - Gets the coordinates of the top edge of a focus rectangle.

? GetFromStartOfLine - Retrieves text on the current line that is located to the left of the active cursor.

? GetToEndOfLine - Retrieves text on the current line that is located to the right of the current cursor.

? GetGraphicID - Gets the ID associated with the graphic under the current cursor. If this function fails, then the pointer is not positioned in a valid location on the graphic. Moving into the center of the graphic increases the success of this call.

? GetGroupBoxName - Gets the name of the current Group box.

? GetHTMLFrameCount - Gets the number of HTML frames present on the current web page.

? GetHotKey - Retrieves the first underlined character in the chunk of text on which the active cursor is positioned. This is especially useful in menus and dialog boxes where an accelerator key for a particular item has been defined and is displayed on the screen as an underlined letter in the name of the control.

? GetItemRect - Gets the bounding rectangle surrounding the specified item or items. Items can be combined using the bitwise (|) operator. When two or more items are combined, the resulting rectangle encloses all of the items.

? GetJAWSDirectory - Retrieves the full path to the directory in which JAWS is running.

? GetJAWSHelpDirectory - Retrieves the full path to the JAWS help directory currently in use. This depends on which language is being used.

? GetJAWSSettingsDirectory - Retrieves the full path to the JAWS settings directory currently in use. This depends on which language is being used.

? GetLineTop - Determines the vertical pixel location of the top of the tallest character of the line on which the current cursor is positioned. Since the cursor position reported by JAWS is based on the base line of characters, this function is the only way of determining how high up a line of text extends.
? GetLinkCount - Gets the number of links present on the current web page
? GetJAWSUserName - Retrieves the name of the user currently logged into JAWS. If this instance of JAWS is not network aware, then the function will return a nul string, "".
? GetJFWVersion - Used to get the version of the currently running JAWS. It can be used to insure that a function is only called on versions of JAWS that support it.
? GetLine - Copies the text from the line where the active cursor is positioned. The copied text can then be used by other script functions.
? GetObject - For certain applications (e.g., Internet Explorer and Microsoft Office ) it is possible for JAWS to obtain information directly from that application rather than relying upon what is displayed on the screen. To do this, JAWS needs a place to go to get that information and a sort of "road map" to tell it where to look. The road map is called an object pointer, and the place it directs JAWS to is called an automation object for that application. There are several ways of obtaining such an object pointer for this purpose, not all of which work with all applications. One such method is GetObject. GetObject sets a pointer to a preexisting automation object. JAWS can then query that automation object to get the information it needs. The GetObject function is similar to the CreateObject function, except that GetObject creates a pointer to an automation object which already exists whereas CreatObject also creates the automation object for the application.
? GetObjectName - Returns the name of the object at the position of the active cursor. If the Pc Cursor is active, the name of the object with focus is returned. Otherwise, the name of the object at the position of the active cursor is returned. The value is returned as a string.
? GetObjectRect - Retrieves the focus rectangle surrounding the object at the position of the active cursor. Will return true if the object has a focus rectangle, false otherwise. This function takes four integer parameters, Left, right, top and bottom. Declare, but do not initialize, the integer variables for these parameters, as they are passed by reference.
? GetObjectState - Obtains and returns the state of the

given object.  If the PC cursor is active, the state of  the object with focus is returned.  Otherwise, the state  of the object at the position of the active cursor is  returned
?	GetObjectType - Retrieves the type of the object located  at the current cursor's location.  Because the type is a  string that differs across languages, this function  should only be used when the objective is to speak the  type of the object.  GetObjectTypeCode should instead be used in all conditional statements because it provides a  language independent solution.
?	GetObjectTypeCode - Retrieves the numeric type code of  the object located at the current cursor's location.  Constants for these type codes all begin with WT_ and are  defined in HJCONST.JSH.  The numeric values are the same  for all languages of JAWS.  Using this function in all  conditional statements instead of using GetObjectType  insures that these statements will function without  change in multiple languages.
?	GetObjectValue - Returns the value of the object at the  position of the active cursor.  If the Pc Cursor is  active, the value of the object with focus is returned.  Otherwise, the value of the object at the position of the  active cursor is returned.  The value is returned as a  string.
?	GetProgramVersion - This function returns the major  version of an application.  For example, if the full  version was 3.00.62, GetProgramVersion would return 3.
?	GetRealWindow - Moves up through the list of window  handles in search of a "real window".  A real window is  one that has a title.
?	GetRestriction - Gets the restriction value for the  current cursor.
?	GetRestrictionName - Can be used to speak the current  level of restriction for the active cursor.  This  function must be provided with a parameter which is the  current level of restriction.  This value can be obtained  with the function, GetRestriction.
?	GetScreenEcho - Obtains the current screen echo setting.  The screen echo setting determines the amount of  information that is to be read as the information is  displayed on the desktop.  The NewTextEvent function  often checks the setting for screen echo before deciding how much information to speak.
?	GetScriptDescription - Retrieves the description of a  specified script.
?	GetScriptSynopsis - Retrieves the synopsis of a specified  script.
?	GetScriptFileName - Retrieves the name of a currently  active default or application script file.  For the

application script file, this is the same as the application's executable file name, except in those cases where the originally loaded application script file replaced itself with another one by means of SwitchToScriptFile. Similarly, for the active default script file, this is the name specified in JFW.INI, except in those cases where the original file replaced itself by means of SwitchToScriptFile.
? GetScriptKeyName - Retrieves the name of the key attached to the specified script. This function is mainly used to create JAWS hot key help files since they will then continue to report the correct key even if the hot key is changed.
? GetScriptKeyNames - Retrieves the names of all of the keys assigned to the specified script. These names are placed in a delimited list with the names being separated (delimited) by \r\n.
? GetSelectedText - Retrieves the currently-selected text as a string.
? GetSynthPitchRange - Retrieves the minimum and maximum pitch settings for the current synthesizer and assigns them to parameters taken by reference.
? GetSynthRateRange - Retrieves the minimum and maximum rate settings for the current synthesizer and assigns them to parameters taken by reference.
? GetSynthVolumeRange - Retrieves the minimum and maximum volume settings for the current synthesizer and assigns them to parameters taken by reference.
? GetTextBetween - gets the text between two horizontal points on the screen on the same line where the cursor is located. This function can be used to speak or capture part of a line.
? GetTextInFocusRects - Retrieves the contents of all focus rectangles into a single buffer.
? GetTextInFrame - Retrieves the text inside a specified frame.
? GetTextInRect - Gets the text contained within the specified rectangle
? GetTopLevelWindow - A top level window is the first or highest level parent window which spawned the child window from which a search is being started. This function finds the top level window with the specified window class and/or window name. If you do not wish to search based on one of the arguments, use "" for that argument.
? GetTreeViewLevel - Obtains the indent level of the item in a tree view that has focus. The root of the tree has the level 0.
? GetVerbosity - Used to obtain the current setting for verbosity. It returns a constant value to indicate the

setting:  "BEGINNER" means speak maximum information,  "INTERMEDIATE" means speak a moderate amount of  information, and "ADVANCED" means speak a minimum amount  of information.  These values can be used in If-Then-Else  statements.
?	GetVoiceParameters - Retrieves the parameters for the  specified voice context.  These include Volume, Rate,  Pitch, Punctuation and person.
?	SetVoiceParameters - Sets the parameters for the  specified voice context.  These include Volume, Rate,  Pitch, Punctuation and person.
?	GetVoicePerson - Retrieves the person name setting for  the specified voice context name.
?	SetVoicePerson - Sets the person name of a voice  specified by the voice context name to a specified person  setting.
?	GetVoicePitch - Retrieves the current pitch setting for  the specified voice context name.
?	SetVoicePitch - Sets the pitch of a voice specified by  the voice context name to a specified value.
?	GetVoicePunctuation - Retrieves the current punctuation  level setting for the specified voice context name.
?	SetVoicePunctuation - Sets the punctuation level of a  voice specified by the voice context name to a specified  value.
?	GetVoiceRate - Retrieves the current rate setting for the  specified voice context name.
?	SetVoiceRate - Sets the rate of a voice specified by the  voice context name to a specified value.
?	GetVoiceVolume - Retrieves the current volume setting for  the specified voice context name.
?	SetVoiceVolume - Sets the volume of a voice specified by  the voice context name to a specified value.
?	GetWindowAtPoint - Gets the handle of the window at the  specified point.  This can be used, for example, with  MouseMovedEvent to get the handle of the window  containing the Mouse pointer.
?	GetWindowClass - Obtains the window class name of a  window.  Most standard windows (non SDM windows) have a  window class name and a window type name.  Window class  names are generally unique to specific windows in  specific applications.
?	GetWindowBottom -Gets the screen coordinate for the  bottom of the window of interest.
?	GetWindowLeft - Gets the screen coordinate for the left  edge of the window of interest.
?	GetWindowRight - Gets the screen coordinate for the right  edge of the window of interest.
?	GetWindowTop - Gets the screen coordinate for the top of  the window of interest.

? GetWindowName - Obtains the title of the specified window, such as for a main Application window, dialog box, or document window. All windows do not have titles.
? GetWindowsOS - Provides information about which operating system is being used.
? GetWindowRect - Gets the bounding rectangle of the window whose handle is specified.
? GetWindowsSystemDirectory - Returns the path to the Windows System directory as a string.
? GetWindowText - Retrieves either all of the text in the specified window or just the highlighted text.
? GetWindowTextEx - As with GetWindowText, retrieves either all of the text in the specified window or just the highlighted text. However, it can also retrieve or ignore the text in child windows.
? GetWindowType - Obtains the window type name of a window. Most standard windows (non SDM windows) have a window type name and a window class name. Examples of window types are: static text, edit field, check box, radio button, etc. The same Window type names are commonly used in many different Windows applications.
? GetWindowTypeCode - Obtains the window type code number for the specified window. These numbers are the same for English and non English versions of JAWS. Many window type numbers have constant values assigned to them in the file HJCONST.JSH. The GetWindowType function returns these constants instead of the window type number.
? GetWindowSubtypeCode - Obtains the window subtype code number for the specified window. This function is quite similar to GetWindowTypeCode, but attempts to be even more specific. If there is no more specific information available, returns the same thing as GetWindowTypeCode. e.g. Calling GetWindowTypeCode using the window handle for the Taskbar would return WT_TABCONTROL, and calling GetWindowSubtypeCode would return WT_TASKBAR (a more specific type of tab control). (See the documentation for GetWindowTypeCode for more details.)
? GetWord - Gets the word where the active cursor is positioned. The text so obtained can then be used by other script functions.
? GraphicsEnumerate - Calls a user-specified function for every graphic contained within a specific window and then passes to this function the coordinates of each graphic. The function is called for each graphic until all have been enumerated or until the user-specified function returns that the enumeration should not continue.
? GraphicsListHelper - Assists the GraphicsList function in creating a string delimited list of graphics labels.
? HasFocusRect - Determines if the specified window has a focus rectangle. A TrackFocusRect=1 statement should be

added to the [OSM] section of the application's JCF file  for this function to work properly.
?	HasTitleBar - Determines whether a window has a title  bar.  If the window has a title bar, then the function  returns a constant value of TRUE.  If the window does not  have a title bar, then a constant value of FALSE is  returned.  Windows that typically have title bars are  dialog boxes and main application windows.
?	IE4GetCurrentDocument - Returns an object that represents  the HTML document in a given browser window.  The  document object can be used to retrieve information about  the document, to examine and modify the HTML elements and  text within the document, and to process events.  This  function obtains the document object of the document in  the current window.
?	If - The If operator marks the beginning of an If-Then-  Else statement.  A fully formulated statement includes:  "If", "Then", "Else", and "EndIf".  The script functions  that appear between the If and the Then are used to  evaluate whether a certain condition is present.  For  example, can a certain graphics character be found in the  active window?  Every statement must include "If",  "Then", and "EndIf".  The "Else" is optional.
?	IniReadInteger - Reads an integer value from an ini style  file.  An ini style file is a file containing sections of  keys with their values.  JCF files as used in JAWS are  ini files, and the features that are set in the various  sections of these files are the keys.  This function is  used to obtain the current setting for integer keys.
?	IniWriteInteger - Writes an integer value to an ini style  file.  An ini style file is a file containing sections of  keys with their values.  JCF files as used in JAWS are  ini files, and the features that are set in the various  sections of these files are the keys.  This function is  used to change the current setting for integer keys.
?	IniReadString - Reads a string value from an ini style  file.  An ini style file is a file containing sections of  keys with their values.  JCF files as used in JAWS are  ini files, and the features that are set in the various  sections of these files are the keys.  This function is  used to obtain the current setting for string keys.
?	IniWriteString - Writes a string value to an ini style  file.  An ini style file is a file containing sections of  keys with their values.  JCF files as used in JAWS are  ini files, and the features that are set in the various  sections of these files are the keys.  This function is  used to change the current setting for string keys.
?	IniRemoveKey- Removes a key and its value from an ini  style file.  An ini style file is a file containing  sections of keys with their values.  JCF files as used in

JAWS are ini files, and the features that are set in the various sections of these files are the keys. This function is used to delete a key and its setting from an ini file.

? IniRemoveSection - removes an entire section from an ini style file. An ini style file is a file containing sections of keys with their values. JCF files as used in JAWS are ini files, and the features that are set in the various sections of these files are the keys. This function is used to delete an entire section from an ini file.

? InHJDialog - Checks to see if one of the dialogs generated by JAWS is active. These include JAWSFind, GraphicsLabeler, DLGSelectItemInList, and DLGSysTray.

? InTable - Used to determine if the active cursor is currently in a table.

? IntToString - Converts an integer value to a string and returns the string value. This might be useful, for example, when one wishes to use an integer as part of a Say statement. Converting the integer to a string allows the value to be spoken as part of the Say statement and avoids the need for a separate SayInteger statement.

? InvisibleCursor - Activates the Invisible cursor and deactivates other cursors. The mouse pointer does not move along with the Invisible Cursor, and it can be used in situations where movements of the JAWS cursor and mouse pointer can cause the window display to change.

? IsBrailleCursor - Determines whether the Braille cursor is active. It can be used in an If-Then-Else statement to verify the active status of a cursor.

? IsInvisibleCursor - Used to determine whether the Invisible cursor is active. It can be used in an If- Then-Else statement to verify the active status of the invisible cursor.

? IsJAWSCursor - Determines whether the JAWS cursor is active. It can be used in an If-Then-Else statement to verify the active status of the JAWS cursor.

? IsPCCursor - Determines whether the PC cursor is active. It can be used in an If-Then-Else statement to verify the active status of the PC cursor.

? IsVirtualPCCursor - Tests to see if the virtual PC cursor is enabled and whether it is being used to navigate within the window with focus. It can be used in an If- Then-Else statement to verify the active status of the virtual cursor.

? IsJFWInstall - Checks to see if JAWS is running in Installation mode (with the /install flag).

? IsKeyWaiting - Indicates if there are other keys in the buffer that need to be processed.

? IsLeftButtonDown - Checks to see if the left mouse button

is locked down.
? IsRightButtonDown - Checks to see if the right mouse button is locked down.
? IsMultiPageDialog - Checks to see if the active cursor is positioned inside a multi-page dialog box.
? IsPointInWindow - Compares the specified pixel coordinates with the window boundaries of the window who's handle is indicated in the parameter hwnd to determine whether it is within the boundaries of that window.
? IsSameScript - Determines if the current script has been called two or more times in a row without any intervening scripts being called and with no more than 500 milliseconds between each call. Using this function allows a script to act differently depending upon the number of consecutive times it has been called. (A script is called whenever a key assigned to it has been pressed.)
? IsSpeechOff - Used to determine if the synthesizer is muted.
? IsWindowDisabled - Checks the status of the current window or control. Returns FALSE if the window is active; TRUE if the window is disabled.
? IsWindowObscured - Checks to see if this window is covered by any others and therefore cannot be entirely seen.
? IsWindowVisible - Checks the visual status of the window. This function will return true even if the window is completely covered by other windows. See function IsWindowObscured for a way to find out if a particular window is covered. Returns TRUE if the Window is visible on the screen, FALSE if the window is not visible on the screen.
? JAWSCursor - Activates the JAWS cursor and deactivates all other cursors.
? JAWSEnd - Moves the JAWS or Invisible cursor, if active, to the end of the line. If the PC cursor is active, JAWS sends an END to the system. The result is controlled by the active application.
? JAWSHome - Moves the JAWS cursor, if active, to the start of the line. If the PC cursor is active, JAWS sends a HOME to the system. The result is controlled by the active application.
? JAWSPageDown - Moves the JAWS cursor, if active, to the bottom of the window, but does not move it left or right. If the PC cursor is active, JAWS sends a PAGE DOWN to the system. The result is controlled by the active application.
? JAWSPageUp - Moves the JAWS cursor, if active, to the top of the window, but does not move it left or right. If

the PC cursor is active, JAWS sends a PAGE UP to the  system.  The result is controlled by the active  application.
?	JAWSFind - Invokes the JAWS Find dialog and then Searches  the visible screen to find the text or graphic you enter.  The text label of the graphic is used for searches.
?	JAWSFindNext - Searches the visible screen for the next  occurrence of the text or graphic last entered in the  JAWS Find Dialog.
?	 JAWSWindow - Invokes the JAWS window, from anywhere else  in Windows, so you can read the online help, change  synthesizers, or perform any other JAWS window function.
?	LeftMouseButton - This function will perform the same  action as clicking the left mouse button on the physical  mouse.  Invoking it twice will yield a double click.  The  position of the JAWS cursor is the same as the mouse  pointer.
?	LeftMouseButtonLock - Locks the left mouse button down to  drag items across the screen or perform any other  application-specific function.  Invoking it again will  unlock the left mouse button to drop the item.
?	Max - This function is used to compare to integers and  returns the greater of the two.
?	Min - This function is used to compare two integers and  returns the lesser of the two.
?	MenusActive - Determines whether a menu is currently  active.  It returns a constant value of "ACTIVE" to  indicate a menu is active and a value of "INACTIVE" to  indicate that a menu is not active.  These constants can  be used in If-Then-Else statements.
?	MinimizeAllApps - Minimizes all application windows, so  the Desktop gets the focus and is visually clear.
?	MouseDown - Moves the mouse downwards by the number of  pixels placed between the parentheses after the function  name.
?	MouseLeft - Moves the mouse left by the number of pixels  placed between the parentheses after the function name.
?	MouseRight - Moves the mouse right by the number of  pixels placed between the parentheses after the function  name.
?	MouseUp - Moves the mouse upwards by the number of pixels  placed between the parentheses after the function name.
?	MoveTo - Moves the active cursor to the specified X and Y  coordinates on the screen.
?	MoveToControl - moves the active cursor to a specific  control within a window.  Although primarily useful  inside dialog boxes, the function can be used in any  window where child controls have unique identifiers  obtained with GetControlID.  It can also be used in SDM  windows with the identifiers obtained using

SDMGetFirstControl, SDMGetLastControl, SDMGetFocus, etc.  If the PC cursor is on when this function is called, the  JAWS cursor is turned on automatically.  Otherwise the  active cursor is used.
?	MoveToFrame - Moves the active cursor to the top left  corner of the specified Frame.  If the PC cursor is  active when this function is used, then the JAWS cursor  is activated and it is moved to the new position,  otherwise the active cursor is moved.
?	MoveToGraphic - Moves the JAWS cursor, Invisible cursor,  or Braille cursor in a specific direction to find a  graphic symbol in the active window.  The graphic is  searched by its text name.
?	MoveToWindow - Moves the active cursor to the specified  window.  If the window contains text, then the cursor is  positioned on the first character.  Otherwise, it is  positioned at the window's center.  If the PC cursor is  active when this function is used, then the JAWS cursor is activated and it is moved to the new position.
?	MSOGetMenuBarObject - Gets the Menu bar object in MS  Office 97 applications.
?	NextCharacter - Performs a special version of` the {RIGHT  ARROW} or next character keyboard command.  When the PC  cursor is active, JAWS allows the application to move the cursor.  When other cursors are active, then JAWS tries  to move the cursor to the next character or graphic it  finds to the right of the cursor's current location.  To  speak the character at the new location, place a  SayCharacter function after the NextCharacter function.
?	NextChunk - Moves the active cursor to the next chunk of  text.  A chunk of text is a section or block of text that  is written to the screen at one time, i.e., with one  function call.  This would typically indicate a phrase or  description that should be spoken as one unit.  This is  useful in reading blocks of static text, control prompts,  and field names.
?	NextLine - Moves the active cursor down to the next line.  In many situations, Windows does not display information  in perfect horizontal rows, and the cursor may not move a  uniform distance each time this function is used.  To  speak the information immediately after the NextLine function is used, place a SayLine function after the  NextLine function.
?	NextNonLink - Positions the Virtual cursor at the next  large block of nonlink text.  This function is often  useful for moving passed advertisement text or repeated  links on a web page.
?	NextParagraph - Moves the active cursor to the beginning  of the next paragraph.  If the PC cursor is active, and  the next paragraph is not already visible, then text in

the window will automatically scroll to bring it into  view.
?	NextSentence - Moves the active cursor to the beginning  of the next sentence.  If the PC cursor is active, and  the next sentence is not already visible, then text in  the window will automatically scroll to bring it into  view.
?	NextWord - Performs a special version of the  {CONTROL+RIGHT ARROW} or next word keyboard command.  When the PC cursor is active, JAWS allows the application  to move the cursor.  When other cursors are active, then  JAWS tries to move the cursor to the next word or graphic  it finds to the right of the cursor's current location.  To speak the word at the new cursor location, place a  SayWord function after the NextWord function.
?	Not - The Not operator reverses the question asked by an  If-Then-Else statement.  It is placed immediately after  the "If" in the If-Then-Else statement.  It says, "If the  specified condition is not present, then perform the  following actions."
?	PassKeyThrough - If this function is invoked, JAWS will  not use or process the following key.  It will be sent  directly to the application, as if JAWS were not loaded.  This is very useful if you have a key conflict.
?	Pause - Stops the processing of a script so that other  applications can complete tasks.  When a Pause function  is placed in a script, JAWS yields to the time needs of  other applications.  Once other applications have been  given the opportunity to use processing time, then JAWS  resumes the script.  The Pause function performs  differently from the Delay Function which stops script  processing for a specified amount of time, irrespective  of whether the application is done processing or not.
?	PCCursor - Activates the PC cursor and deactivates all  other cursors.
?	PlaySound - Plays a wave file through the computer sound  system.  Requires one parameter that specifies the wave  filename.  If a complete path is not specified, JAWS will search the path for the file.
?	PriorCharacter - Performs a special version of the {LEFT  ARROW} or prior character keyboard command.  When the PC  cursor is active, JAWS allows the application to move the cursor.  When other cursors are active, JAWS tries to  move the cursor to the prior character or graphic it  finds to the left of the cursor's current location.  To  speak the information at the new cursor location, place a  SayCharacter function after the PriorCharacter function.
?	PriorChunk - Moves the active cursor to the prior chunk.  A chunk of text is a section or block of text that is  written to the screen at one time, i.e., with one

function call.  This would typically indicate a phrase or  description that should be spoken as one unit.  This is  useful in reading blocks of static text, control prompts,  and field names.
?	PriorLine - Performs a special version of the {UP ARROW}  keyboard command.  When the PC cursor is active, JAWS  allows the application to move the cursor.  When other  cursors are active, then JAWS tries to move the cursor up  to the line above its current position.  To speak the  line of information at the new location, place a SayLine  function after the PriorLine function.
?	PriorParagraph - Moves the active cursor to the beginning  of the prior paragraph.  If the PC cursor is active, and  the prior paragraph is not already visible, then text in  the window will automatically scroll to bring it into  view.
?	PriorSentence - Moves the active cursor to the beginning  of the prior sentence.  If the PC cursor is active, and  the prior sentence is not already visible, then text in  the window will automatically scroll to bring it into  view.
?	PriorWord - Performs a special version of the  {CONTROL+LEFT ARROW} or prior word keyboard command.  When the PC cursor is active, JAWS allows the application  to move the cursor.  When other cursors are active, JAWS  tries to move the cursor to the prior word or graphic it  finds to the left of the cursor's current location.  To  speak the word at the new location, place a SayWord  function after the PriorWord function.
?	ProcessNewText - This function is used to force a  NewTextEvent call.  The NewTextEventFunction is normally  called only when new text is written to the screen.  ProcessNewText can be useful if you need to call  NewTextEvent before FocusChangedEvent.
?	Refresh - Refreshes the screen.  This rewrites the  information to the JAWS off screen model and can be used  to clear leftover data from the screen.
?	RefreshWindow - Refreshes the contents of a window and  all windows contained within it.  This is much faster  than refreshing the entire screen.
?	RemoveHook - Removes a hook function put in place by the  AddHook function.  See the section on Hook Functions for  more information.
?	ResetSynth - Reinitializes the synthesizer with the  proper volume, rate, and pitch settings.  Used to bring  the synthesizer back to normal if it has gotten out of  sync with JAWS.
?	RestoreCursor - Reactivates the cursor that was saved  when the SaveCursor function was last used.  If the  cursor being restored is the JAWS cursor, Invisible

cursor, or Braille cursor, then the cursor is also  returned to its previous position on the desktop. If the  cursor currently in use is different from the cursor that  was in use when the SaveCursor function was used, then it  is deactivated and the previous cursor is reactivated.
?	Return - The return operator terminates execution of the  function in which it appears and returns control (and the  value of expression if given) to the calling function.  To define return values, the name of the function is  preceded by the return type.  Since a Script cannot  return a value, a Return operator that is used from  within a script should never be followed by a value.
?	RGBStringToColor - Converts a 9-digit String of the form  "255255255" to a color value.  The 9-digit string must be  enclosed in quotation marks.  (in this case, the color  would be white.)  Color equivalents are found in the file  COLORS.INI.
?	RightMouseButton - This function will perform the same  action as clicking the right mouse button on the physical  mouse.  Invoking it twice will yield a double click.  The  position of the JAWS cursor is the same as the mouse  pointer.
?	RightMouseButtonLock - This function will lock the right  mouse button and allow the user to perform any such  function required by an application.  Invoking it again  will unlock the button.
?	RouteBrailleToJAWS - Repositions the Braille cursor to  the same position as the JAWS Cursor.
?	RouteBrailleToPC - Repositions the Braille cursor to the  same position as the PC cursor.
?	RouteInvisibleToJAWS - Moves the Invisible cursor to the  location of the JAWS cursor and activates the Invisible  cursor.
?	RouteInvisibleToPC - Moves the Invisible cursor to the  location of the PC cursor and activates the Invisible  cursor.
?	RouteJAWSToBraille - Repositions the JAWS cursor so that  it is in the same position as the Braille cursor.
?	RouteJAWSToInvisible - Moves the JAWS cursor to the  location of the Invisible cursor and activates the JAWS  cursor.
?	RouteJAWSToPC - Repositions the JAWS cursor so that it is  in the same position as the PC cursor.  The JAWS cursor  is usually connected to the mouse pointer, thus, when  this function is used, it places the mouse pointer on top  of the caret or the highlighted selection cursor, or on whatever other type of pointer that is currently being  used.
?	RoutePCToBraille - Repositions the PC cursor so that it  is in the same position as the Braille cursor.

? RoutePcToInvisible - Repositions the PC cursor so that it is in the same position as the Invisible cursor.
? RoutePCToJAWS - Attempts to move the PC cursor to the position of the JAWS cursor. This function is the same as a single click of the left mouse button, which instructs Windows to move its insertion point to the position of the mouse pointer. The successful use of this function is related to the Windows operating system; there are often situations where Windows cannot move the PC cursor to the mouse pointer.
? SaveCursor - Saves the name of the active cursor. If the JAWS cursor, Invisible cursor, or Braille cursor is being used, then the position of the cursor is also saved. Scripts that use the SaveCursor function usually use the RestoreCursor function later in the script. If a RestoreCursor is not used, JAWS will execute one automatically when the script terminates.
? Say - This function speaks a string of text, much like the SayString function, but it has a second parameter called an output mode. These output modes allow the message to be spoken with a particular set of speech characteristics. It is possible to use separate output modes to speak title lines, dialog controls, menu items, etc, with different voices or at different verbosity levels. By using the SayMessage function instead, it is possible to assign short and long messages to many output types for JAWS Help and other information.
? SayActiveCursor - Says the name of the active cursor and its position by pixel location.
? SayAll - Says all readable information from the point of the active cursor to the bottom of the window. If the PC cursor is active, JAWS scrolls the screen by moving the PC cursor down. If the JAWS cursor is active, the rest of the window is read by moving the JAWS cursor down a line at a time.
? SayCharacter - Reads the character or graphic symbol at the active cursor. If the PC cursor is active, JAWS looks for the visible caret or the light bar. If the JAWS cursor is active, it speaks the character or graphic at the mouse pointer.
? SayCharacterPhonetic - Uses special pronunciation rules to read the character located at the position of the active cursor. Thus, A is pronounced alpha, B bravo, etc. The associations between characters and their phonetic pronunciations are made in the [PhoneticSpell] section of .JCF files. Which words are used can be changed by the user, if desired.
? SayChunk - Speaks the chunk of information to which the active cursor is pointing. A "chunk" is text and graphic information that was written to the screen in a single

operation.  SayChunk is similar to SayField, however, the  SayField function uses logic to determine the text that  is to be spoken, while SayChunk simply reads the text  that was stored in the Off Screen Model as a single unit.
?	SayColor - Says the color at the active cursor.  Uses the  COLORS.INI to define the colors from the 9 digit  numerical RGB values.  If you hear numbers instead of a  color name, add that entry to the colors.ini file and  define its color.
?	SayControl - Used in dialog boxes to speak the contents  of a child window along with its prompt.  It reads edit  fields, list boxes, check boxes, radio buttons, etc.
?	SayCurrentScriptKeyLabel - Speaks the Key Name attached  to the current script, honoring both the typing echo  setting and the label as defined in default.jcf.
?	SayCursorPos - Speaks the row and column position of a  cursor.
?	SayField - Reads the field of text where the active  cursor is pointing.  A "field of text" is a section or  block of text that has a common attribute, i.e., bold,  underlined, italics, or strikeout.  The use of the  attribute must be contiguous.  The SayField function uses  logic to determine the text that is to be spoken, while  the SayChunk function simply reads the text that was  stored in the JAWS Off Screen Model as a single unit.
?	SayFocusRect - Says the contents of a focus rectangle.  Returns TRUE if any text was spoken, FALSE otherwise.  A  TrackFocusRect=1 statement should be added to the [OSM] section of the application's JCF file for this function  to work properly.
?	SayFocusRects - Says the contents of focus rectangles.  If there is only one such rectangle, this is exactly like  SayFocusRect.  If there is more than one, this function  says the contents of all of them, while SayFocusRect says  only the first one.  A TrackFocusRect=1 statement should  be added to the [OSM] section of the application's JCF  file for this function to work properly.
?	SayFont - Speaks the font name and size at the active  cursor.
?	SayFrame - Speaks the contents of the specified frame.
?	SayFrameAtCursor - All text within the boundaries of the  frame that contains the active cursor is spoken.
?	SayFromCursor - Reads the text from the cursor to the end  of the line, including the current character.
?	SayToCursor - Reads the text from the start of the line  up to the cursor, not including the current character.
?	SayInteger - Speaks numeric, integer data, often the  contents of an integer variable.
?	SayString - Speaks string data, usually either the  contents of a string variable or a specific message.  If

actual text is used, it should be placed inside of quotation marks within the parentheses that follow the function name. If a variable or constant is used, no quotation marks should be present. It is now recommended that the SayString function be avoided in favor of the SayMessage function (see below).
? SayLine - Speaks the current line. If the PC cursor is active, reading will be restricted to the current item or window. Otherwise reading will include all the text on approximately the same line, even if it is outside the current item or window, unless JAWS cursor Restriction is on.
? SayMessage - This function takes short and long messages supplied as parameters and speaks the appropriate message based on the specified output type, also supplied as a parameter. This function is similar to Say, except that it is possible to assign short and long messages to many output types for JAWS Help and other information. It is now recommended that the SayMessage function be used in preference to the SayString function.
? SayObjectActiveItem - Says the active element in certain controls. For example, in a list view, it will say the selected item. In a menu, it will say the active menu item. In a dialog box, it will say the selected tab.
? SayObjectTypeAndText - Speaks the name and type of the object located at the current cursor's location. This function is similar to SayWindowTypeAndText except that it is more specific. If a particular window contains multiple objects, this function will speak information about the one at the cursor, while SayWindowTypeAndText will speak information about the enclosing window. If the window does not contain multiple objects, then the functions operate identically. When this function is used, it marks the text it reads so that the SayNonHighlightedText and SayHighlightedText functions do not repeat the same information when they are triggered.
? SayParagraph - Reads the current paragraph from the beginning.
? SaySentence - Reads the sentence containing the character on which the active cursor is positioned.
? SayTextBetween - Says the text between two horizontal points on the screen on the same line as the active cursor.
? SayToBottom - Reads the active window from the position of the active cursor to the bottom of the window. The cursors do not move as text is read, the window will not scroll to display additional text.
? SayUsingVoice - Speaks a string of text using a specific set of speech characteristics called voice context.
? SayWindow - Reads the specified window. It either reads

highlighted text or all text in the window, depending on  the parameters entered by the user.
?	SayWindowTypeAndText - Reads a specified series of  information from a window.  It reads the window title  (when one is present), the window type, the contents in  the window, and provides related information about the  current dialog option.  When this function is used, it marks the text it reads so that the SayNonHighlightedText  and SayHighlightedText functions do not repeat the same  information when they are triggered.
?	SayWord - Reads the word or graphic symbol at the active  cursor.  If the PC cursor is active, JAWS looks for the  visible caret or the light bar.  If the JAWS cursor is  active, it speaks the word or graphic at the mouse  pointer.
?	ScheduleFunction - Runs a user defined function in a set  period of time.  This Function is useful when you want to  perform a task and then check on the results at a later  time.  Once this function is used, you can call  UnscheduleFunction to cause the user defined event not to  run.
?	UnscheduleFunction - Used to cancel a ScheduleFunction  call.
?	ScreenEcho - Toggles the screen echo among the three  possible states.  The default is Highlighted, which  speaks only highlighted text when it appears on the  screen.  All speaks all the text that gets written to the  screen, and None speaks none of the text.
?	ScreenGetHeight - Gets the height of the screen in  pixels.
?	ScreenGetWidth - Gets the width of the screen in pixels.
?	SDMGetCurrentControl - Retrieves the ID of the control on  which the active cursor is positioned inside an SDM  dialog box.
?	SDMGetFirstControl - Obtains the control ID of the first  dialog option in an SDM dialog box.  It can provide the  control ID that is needed by the SDMSayControl function.
?	SDMGetLastControl - Provides the control ID for the last  option in an SDM dialog box.  It can provide the control  ID that is needed by the SDMSayControl function.
?	SDMGetFocus - Used to get the Control ID of the active  dialog option or control in an SDM dialog box.  This  function is most often used as a parameter for the SDMSayWindowTypeAndText function to provide it with a  control ID.
?	SDMGetNextControl - Obtains the control ID for the next  option in an SDM dialog box.  It can provide the control  ID that is needed by the SDMSayControl function.
?	SDMGetPrevControl - Provides the control ID for the  previous option in an SDM dialog box.  It can provide the

control ID that is needed by the SDMSayControl function.
? SDMSayControl - Speaks the contents of a child window along with its prompt in an SDM dialog box. It reads edit fields, list boxes, check boxes, radio buttons, etc. This function is equivalent to SayControl, but is exclusively designed for SDM dialog boxes. All child windows in an SDM dialog box have the same window handle, and the control ID is used to distinguish among the various options in the dialog box.
? SDMSayStaticText - Reads a type of text called "static text" that may be displayed in an SDM dialog box. It does not read the text in edit fields, check boxes, etc., or the titles or prompts that are often associated with dialog options.
? SDMSayWindowTypeAndText - Used with SDM windows to read the window title (when one is present), the window type, the contents in the window, and related information about the current dialog box option. When this function is used, it marks the text it reads so that the SayNonHighlightedText and SayHighlightedText functions do not repeat the same information when they are triggered. This function is only used with SDM windows and not with regular dialog windows.
? SelectFromStartOfLine - Sends a SHIFT+HOME to the system. The result is controlled by the application. In word Pad , all text from the start of the line to the PC cursor is selected. (Used primarily by MAGic)
? SelectToEndOfLine - Sends a SHIFT+END to the system. The result is controlled by the application. In Word Pad , all text from the PC cursor to the end of the line is selected. (Used primarily by MAGic)
? SelectFromTop - Sends a CTRL+SHIFT+HOME to the system. The result is controlled by the application. In Word Pad , all text from the top of the document to the PC cursor is selected. (Used primarily by MAGic)
? SelectToBottom - Sends a CTRL+SHIFT+END to the system. The result is controlled by the application. In Word Pad , all text from the PC cursor to the bottom of the document is selected. (Used primarily by MAGic)
? SelectNextCharacter - Sends a SHIFT+RIGHT ARROW to the system. The result is controlled by the application. In Word Pad , the character to the right of the PC cursor is selected. (Used primarily by MAGic)
? SelectPriorCharacter - Sends a SHIFT+LEFT ARROW to the system. The result is controlled by the application. In Word Pad , the character to the left of the PC cursor is selected. (Used primarily by MAGic)
? SelectNextLine - Sends a SHIFT+DOWN ARROW to the system. The result is controlled by the application. In Word Pad , the line below the PC cursor is selected. (Used

primarily by MAGic)
? SelectPriorLine - Sends a SHIFT+UP ARROW to the system. The result is controlled by the application. In Word Pad , the line above the PC cursor is selected. (Used primarily by MAGic)
? SelectNextScreen - Sends a SHIFT+PAGE DOWN to the system. The result is controlled by the application. In Word Pad , The next screen of text is selected. (Used primarily by MAGic)
? SelectPriorScreen - Sends a SHIFT+PAGE UP to the system. The result is controlled by the application. In Word Pad , the prior screen of text is selected. (Used primarily by MAGic)
? SelectNextWord - Sends a CTRL+SHIFT+RIGHT ARROW to the system. The result is controlled by the application. In Word Pad , the word to the right of the PC cursor is selected. (Used primarily by MAGic)
? SelectPriorWord - Sends a CTRL+SHIFT+LEFT ARROW to the system. The result is controlled by the application. In Word Pad , the word to the left of the PC cursor is selected. (Used primarily by MAGic)
? SetDefaultJCFOption - Changes the default value for an option in the default JAWS configuration file. Settings in DEFAULT.JCF are used until a JCF is loaded for an application. When a JCF is not available for the application, then settings in the default JCF continue to be used. The function SetJCFOption is used to set a value in the JCF for an active application.
? SetJCFOption - Changes a value in the JAWS configuration file for the active application. A new JCF file is loaded each time a different application is used. When a JCF is not available for the application, then the settings in the DEFAULT.JCF are used. The SetDefaultJCFOption is used to change a default JCF option.
? SetJAWSLanguage - In the international versions of JAWS, changes the JAWS operating environment to the specified language.
? SetFocus - moves the focus to a specified window.
? SetGraphicLabel - Adds a graphic label to the specified graphics file (jgf).
? SetRestriction - For all cursors except the PC cursor, this function sets the area within which the current cursor is free to move.
? SetRestrictionToFrame - For any active cursor except the PC cursor, this function restricts that cursor's movement to within the specified frame.
? SetRestrictionToRect - For any active cursor except the PC cursor, this function restricts that cursor's movement to within the specified rectangle.

? SetSynth - Causes JAWS to switch to the specified synthesizer.
? SetSynthLanguage - For synthesizers that support multiple languages, changes the synthesizer to the specified language.
? ShiftTabKey - If the virtual cursor is not active, this function simply passes the SHIFT+TAB key combination through to the application. If the Virtual Cursor is active, it is moved to the previous link or control in the tab order.
? TabKey - If the virtual cursor is not active, this function simply passes the TAB key through to the application. If the Virtual Cursor is active, it is moved to the next link or control in the tab order.
? ShouldItemSpeak - This function is used to determine whether or not the user has specified that items spoken with a given Output Mode should be spoken in the current verbosity level.
? ShowHelpByID - Opens up winhelp using a specified file and topic ID.
? ShowHelpByName - Opens up winhelp using a specified file and index topic.
? ShutDownJAWS - Terminates the JAWS application.
? SpeechInUse - Checks to see if speech output is being used.
? SpeechOff - Causes the synthesizer to be muted. This is particularly useful when you need to free the synthesizer so that a wave file can be played or if you wish to mute speech during script or function playback. You can reverse this action with SpeechOn.
? SpeechOn - UnMutes the synthesizer. Reverses the action of SpeechOff.
? SpellString - Spells a string of text. This is similar to the SayString function, however it spells the string letter by letter instead of speaking words.
? SpellWord - Spells the word at the active cursor.
? StartJawsTaskList - The JTL allows the user to minimize, maximize, close, start, and switch to applications.
? StringContains - Determines whether a specified text string is contained within another text string. You must specify both the string to be searched and the text to be searched for. The latter is case sensitive. This is the only way to check strings in a case sensitive fashion since logical operator comparisons are not case sensitive.
? StringLength - Used to find the length of a string.
? StringLower - Converts a mixed case string to all lower case. This might be useful, for example, in combination with the StringContains function when one does not wish the check to be case sensitive.

? StringUpper - Converts a mixed case string to all upper  case.
? StringLeft - Extracts a specified number of leftmost  characters from a string.
? StringRight - Extracts a specified number of rightmost  characters from a string.
? StringSegment - When a string contains delimiters,  StringSegment can be called to extract a segment of the  string.  One is the index of the first string.
? StringToInt - Converts a string value to an integer and  returns the integer value.  This is used on string names  which are comprised of numbers.
? SubString - Extracts part of a string from another  string.  It could be used to read a portion of the  information that appears on a status line or obtain a  part of a string for logical comparison to another  string.
? StopSpeech - Silences the synthesizer.  This is identical  to pressing the CTRL key.
? SwitchToScriptFile - Removes the currently running script  and loads a new one in it's place.
? SysGetDate - Obtains the current system date.
? SysGetTime - Obtains the current system time.
? SysTrayGetItemCount - Obtains the number of items in the  system tray.
? SysTrayGetItemToolTip - Used in ListTaskTrayIcons to  obtain the tool tip that corresponds to a specific task  tray icon.  This function takes a single parameter, the  index of the task tray icon for which the tool tip is  desired.  It returns a string value that is the text of  the tool tip in question.
? SysTrayMoveToItem - Moves the JAWS cursor to the location  of an item in the system tray.
? TrapKeys - Turns Trap key mode On or Off.  When Trap Key  mode is on, any keys not attached to scripts are simply  ignored and not passed on to the current application.  The primary use for this feature is in the Keyboard Help,  where keys not attached to scripts should be ignored.
? TurnOffFormsMode - Used to exit from forms mode and turn  the virtual PC cursor back on.
? ToggleHomeRow - Toggles Home Row mode on and off.  Home  Row mode is a shifted state for the keyboard; it is  analogous to the numlock on the numeric keypad.  When it  is on, the alphabet keys and number keys can be used to  perform script functions.  A suite of special window  functions is also available in this mode.  (See the  section on Utility Functions later in this manual.)  When  Home Row mode is turned back off, then the keys perform  their standard functions.
? ToggleRestriction - Toggles JAWS between restricted and

unrestricted mode.  These modes do not affect the  movement of the PC cursor.  When Unrestricted mode is  used, the other cursors can be freely moved within the  active application window.  When Restricted mode is used,  the movement of these cursors is limited to a child window.  When the function is used, it returns a constant  value representing the new setting.  "ON" = restriction  on, and "OFF" = restriction off.
?	TypeCurrentScriptKey - Passes the key attached to the  current script directly through to the application.
?	TypeString - Used to simulate the typing of a string of  characters.  For Example, to send a group of characters  through to the application simulating the string "test,"  you could either enter the line {t}{e}{s}{t}, or you  could use TypeString ("test").
?	VerbosityLevel - Controls how much JAWS will say when the  focus changes or other events occur on the screen or in  response to key strokes.
?	While - The While operator marks the beginning of a  "While Loop, the end of which is indicated by an EndWhile  statement".  While loops can be constructed to perform a  series of script statements repeatedly until a condition  becomes true or false.  Thus, while Loops can be used to  perform repeated actions and save space or to perform a  series of activities repeatedly when the user does not  know in advance how many times the repetition will be  required.

# 17	Appendix D

Important Built-In Functions Arranged by Task

Note - This list has been updated to reflect new functions  added in JAWS 3.7.  Those using earlier versions will not  have all of the functions shown here.

## 17.1	Cursor

Use the following functions to activate and position the  four cursors available in JAWS: PC, JAWS, Invisible, and  Virtual.

GetActiveCursor
SayActiveCursor
GetCursorCol
GetCursorRow
SayCursorPos
CaretVisible
SaveCursor

RestoreCursor  InvisibleCursor
RouteInvisibleToJAWS  RouteInvisibleToPC  IsInvisibleCursor  JAWSCursor  RouteJAWSToPC
RouteJAWSToInvisible  IsJAWSCursor  ToggleRestriction  SetRestriction  SetRestrictionToFrame
SetRestrictionToRect  GetRestriction  GetRestrictionName  PCCursor  RoutePCToJAWS
IsPCCursor  RouteBrailleToJAWS  RouteBrailleToPC  RouteJAWSToBraille  RoutePCToBraille
IsVirtualPCCursor  TurnOffFormsMode

## 17.2 Positioning

Use the following functions to position a cursor, get  information at the desired location, and select text.

MoveTo  MoveToControl  MoveToGraphic  MoveToWindow  MoveToFrame  JAWSEnd
JAWSHome  JAWSPageDown  JAWSPageUp  PriorCharacter  GetCharacter
GetCharacterPoints  GetCharacterFont  NextCharacter  PriorChunk  GetChunk  NextChunk
PriorLine  GetLine  GetLineTop

NextLine  PriorWord  GetWord  NextWord  GetField  GetCharacterAttributes  GetCharacterWidth
NextParagraph  PriorParagraph  NextSentence  PriorSentence  NextNonLink  GetSelectedText
SelectNextCharacter (Used primarily by MAGic)  SelectPriorCharacter (Used primarily by MAGic)
SelectNextWord (Used primarily by MAGic)  SelectPriorWord (Used primarily by MAGic)
SelectNextLine (Used primarily by MAGic)  SelectPriorLine (Used primarily by MAGic)
SelectToEndOfLine (Used primarily by MAGic)  SelectFromStartOfLine (Used primarily by MAGic)
SelectNextScreen (Used primarily by MAGic)  SelectPriorScreen (Used primarily by MAGic)
SelectToBottom (Used primarily by MAGic)  SelectFromTop (Used primarily by MAGic)  TabKey
ShiftTabKey  BackspaceKey  EnterKey  NextCell  PriorCell  UpCell  DownCell  GetCell
GetColumnHeader  GetRowHeader  InTable

## 17.3 Conditional Processing and Looping

Use the following functions to perform conditional  processing and looping.

If-Then  EndIf  Else  ElIf  Not  While  EndWhile

## 17.4 Say

Use the following functions to speak information, either  from the screen or using messages within a script.

Say  SayAll  SayInteger  SayCharacter  SayCharacterPhonetic  SayWord  SpellWord  SayChunk  SayControl  SayField  SayLine  SayToCursor  SayFromCursor  SayTextBetween  SayToBottom  SayFont  SayString  SpellString  SayParagraph  SaySentence  SayUsingVoice  SayCell  SayColumnHeader  SayRowHeader  SayMessage  SayControlInformation

## 17.5 String and Integer Manipulation

Use the following functions to obtain data about and  manipulate strings.

StringContains  SubString  StringToInt  IntToString  StringLength  StringLower  StringUpper  StringSegment  StringLeft  StringRight  ProcessNewText  FormatString

Min  Max

## 17.6 Application

Use the following functions to acquire information about an  application.

GetAppFileName  GetAppTitle  GetAppFilePath  GetTreeViewLevel  GetProgramVersion
GetWindowAtPoint  IsPointInWindow

## 17.7 Braille

Use the following functions to query Braille conditions and  display lines on the Braille display.

BrailleInUse  PriorBrailleString  BrailleString  NextBrailleString  BraillePanLeft  BraillePanRight
BrailleLine  SixDotBraille  EightDotBraille  GetBrailleCellColumn  GetBrailleCellRow
GetBrailleMode  SetBrailleMode  GetLastBrailleRoutingKey  BraillePriorLine  BrailleNextLine
BrailleG2StringLength  BrailleRefresh  BrailleString  IsBrailleCursor  BrailleAddFocusItem
BrailleAddFocusLine  BrailleAddFrame  BrailleAddString

## 17.8 Mouse

Use the following functions to move, click, and get  information about the mouse pointer.

MouseUp  MouseDown  MouseLeft  MouseRight  LeftMouseButton  LeftMouseButtonLock ShiftLeftMouseClick  AltLeftMouseClick  ControlLeftMouseClick  IsLeftButtonDown RightMouseButton  IsRightButtonDown  GetCursorShape  DragItemWithMouse

## 17.9 Find

Use the following functions to find specific information on  the screen.

FindFirstAttribute  FindPriorAttribute  FindNextAttribute  FindLastAttribute  FindGraphic FindString  JAWSFind  JAWSFindNext  FindColors  FindDescendentWindow FindTopLevelWindow

## 17.10 Windows and Objects

Use the following functions to acquire information about  windows and objects and to move from window to window.

GetForegroundWindow  GetCurrentWindow  HasTitleBar  GetRealWindow  GetAppMainWindow GetFirstWindow  GetPriorWindow  GetNextWindow  GetWindowLeft  GetWindowRight GetParent  IsWindowDisabled

IsWindowObscured  IsWindowVisible  GetFirstChild  GetWindowClass  GetWindowName
GetWindowsOS  GetWindowSubtypeCode  GetWindowType  GetWindowTypeCode
GetWindowText  GetWindowTextEx  JAWSWindow  MinimizeAllApps  SayWindow
SayWindowTypeAndText  GetFocus  SetFocus  HasFocusRect  SayFocusRect  SayFocusRects
GetFocusRect  GetFocusRectBottom  GetFocusRectLeft  GetFocusRectRight  GetFocusRectTop
GetTextInFocusRects  ActivateMenuBar  DialogActive  GetDialogPageName  GetDialogStaticText
IsMultiPageDialog  GetControlID  GetCurrentControlID  GetDefaultButtonName  MenusActive
GetObject  GetObjectAtPoint  GetObjectType  GetObjectTypeCode  SayObjectTypeAndText
GetCurrentObject  GetFocusObject  GetObjectName  GetObjectValue  GetObjectRect
GetObjectState  GetLineTop  GetWindowBottom  GetWindowTop  RefreshWindow

ToggleHomeRow  ControlCanBeChecked  ControlIsChecked  GetGroupBoxName  GetControlAttributes  SayObjectActiveItem  IE4GetCurrentDocument  GetHTMLFrameCount  GetLinkCount  MSOGetMenuBarObject  GetItemRect  GetWindowRect

## 17.11 Frames

Use the following functions to read and manipulate frames.

GetFrameDescription  GetFrameNameAtCursor  GetFrameSynopsis  SayFrame  SayFrameAtCursor  GetTextInFrame

## 17.12 Scripts

Use the following functions to get information about scripts  and script files and control their operation.

GetCurrentScriptKeyName  GetScriptDescription  GetScriptKeyName  GetScriptKeyNames  GetScriptSynopsis  IsSameScript  PerformScript  AddHook  RemoveHook  TrapKeys  SwitchToScriptFile  GetScriptFileName

## 17.13 SDM

The following functions are used to get information about  SDM Dialogs only.

SDMGetFocus  SDMGetFirstControl  SDMGetPrevControl

SDMGetNextControl  SDMGetLastControl  SDMSayControl  SDMSayWindowTypeAndText
SDMGetCurrentControl

## 17.14 Speaking Level

The following functions query verbosity settings and/or  reset them.

GetScreenEcho  ScreenEcho  GetVerbosity  VerbosityLevel  ShouldItemSpeak

## 17.15 Options

Use the following functions to get JAWS settings information  and change them.

SetJAWSLanguage  GetJCFOption  SetJcfOption  GetDefaultJcfOption  SetDefaultJcfOption
GetJAWSDirectory  GetJAWSSettingsDirectory  GetJAWSHelpDirectory  GetJAWSUserName
GetJFWVersion  GetJFWSerialNumber  InHJDialog  IsJFWInstall

## 17.16 System Activities

Use the following functions to interact with the system.

MessageBox  ExMessageBox  InputBox  GetHotKey  PassKeyThrough  PlaySound  Pause
Delay  Refresh  Run

FileExists  GetFileDate  DLGSelectItemInList  DLGSelectScriptToRun  DLGSysTray  DLGSelectControls  ScheduleFunction  UnscheduleFunction  ActivateStartMenu  Beep  CopyToClipboard  ClipboardHasData  GraphicsLabeler  SetGraphicLabel  GetGraphicID  GraphicsEnumerate  GraphicsListHelper  ShutDownJAWS  ScreenGetWidth  ScreenGetHeight  SysTrayGetItemCount  SysTrayGetItemToolTip  SysTrayMoveToItem  SysGetDate  SysGetTime  StartJawsTaskList  GetWindowsSystemDirectory  ShowHelpByName  ShowHelpByID  IsKeyWaiting  IniReadInteger  IniWriteInteger  IniReadString  IniWriteString  IniRemoveKey  IniRemoveSection

## 17.17 Colors

The following functions can be used to learn about and  manipulate screen colors.

GetColorBackground  GetColorText  GetColorName  ColorToRGBString  RGBStringToColor  GetColorField

## 17.18 Control Synthesizer

The following functions are used to control synthesizer  activity.

ResetSynth
SpeechOn
SpeechOff
IsSpeechOff
StopSpeech
SpeechInUse
SetSynth
SetSynthLanguage  GetVoiceVolume  SetVoiceVolume
GetVoiceRate
SetVoiceRate
GetVoicePitch
SetVoicePitch  GetVoicePunctuation  SetVoicePunctuation  GetVoicePerson  SetVoicePerson
GetSynthVolumeRange  GetSynthRateRange  GetSynthPitchRange  GetVoiceParameters
SetVoiceParameters

18 Appendix E

How to Scope Out and Customize an Unknown Application  (courtesy of Glen Gordon)

I. Determine how well the app works with no customizations  A. Do dialog boxes speak properly?
B. Can the user quickly locate important information on  the screen?
C. Can the keyboard be used to accomplish all tasks?  D. What information should be combined on the Braille  display for quick access?  II. investigating dialog boxes  A. The Utility scripts on the Home Row keys (See  Chapter 8.1) and the ScreenSensitiveHelpTechnical  script (CONTROL+INSERT+F1) (See Chapter 8.2) can help  with the following:
1. Determine if each control is its own window.  2. if controls have nonstandard window classes, try reassigning them. Does this cause the right  window types to be spoken when tabbing to them?  (Unknown window classes can be reassigned to known

classes using the Window Class Dialog accessed  with INSERT+7 or with INSERT+F2 followed by the  letter W and ENTER.)  Reclassification of edit  fields in Internet Explorer in this way results in  proper reading of these fields.
If the PC cursor doesn't track the proper item in a  nonstandard control:
1. Try setting TrackFocusRect=1 in the application  JCF file.  This is done by adding this line to the  [OSM] section of the appropriate JCF file.  This should  allow the PC cursor to track the focus if it is a focus  rectangle.  (Some spreadsheets like Microsoft Excel use  a variant of the focus rectangle which can only be  tracked by checking the Include Lines checkbox found in  the Advanced Options dialog of the Configuration  Manager's Set Options menu.)  If either of these cause  the focus to track, automatic speaking can be scripted  by adding SayFocusRect function calls to the scripts  bound to the cursor and Tab keys.
2. Modify the FocusChangedEvent or  SayFocusedWindow function to do custom processing  for troublesome dialogs.  See the EUDORA.JSS  FocusChangedEvent function for examples.
3. Write a custom function for each dialog, and  have the FocusChangedEvent call that function when  that dialog is active.  For an example, see the  FocusChangedEvent function in the WINWORD.JSS  which has been modified to handle SDM windows and  two versions of Microsoft Word.
4. When tabbing between controls, does the  FocusChangedEvent get called?  If not, add the line ForceFocusChange=1 to the [OSM] section of  the application's JCF file, and try again.  The FocusChangedEvent function is normally only called  when the new focus has a different window handle  from the prior focus.  In some applications like  certain parts of MS Access, an edit field can have  the same handle as the prior field.  Enabling the  ForceFocusChange option forces the FocusChangedEvent to run if the new window has  different pixel coordinates from the old.
5. Do controls have unique control IDs?  If so,  use them to identify which control is active.  Thus, one could use an If-Then statement to check  the value of the window's Control ID and then perform certain operations or speak certain  messages for each unique Control ID.
6. Use pixel coordinates to identify which control  is active.  (Coordinates should be window, not screen relative.)  Thus, one could use an If-Then  statement to check the value of the current pixel  location and then perform certain operations or  speak certain messages for each unique set of  coordinates.
7. write custom code for each problematic control.  a. if the dialog is just one big window  (i.e., controls do not have unique Control  Ids), by setting TrackFocusRect=1, the PC  cursor may move as you TAB among controls.  b. write a script to say the proper  information based on pixel location, and  attach it to the TAB and SHIFT+TAB keys.
III. Locating important information on the screen:  A. Create frames around areas of interest.  Assign them  to hot keys or make them speak automatically.  (An  example of where this could be done is the status line  page numbers in Microsoft Word.  One could read the  current page number with a hot key set to read the  frame or create a frame to echo all changes in the page number automatically.)
B. Traverse the window hierarchy to find information of  interest.  That is, move up or down to parent and child  windows or among windows at the same logical level to find a  control or text field of interest.
C. Search for text or a graphic on the screen.  An  example of this would be to search for the toolbar  buttons in Microsoft Office to check their pressed or  non-pressed status.  This is not as reliable as  traversing the hierarchy.
IV. Providing enhanced keyboard access
A. write scripts to move to and click on particular  locations and attach them to keys.  This simulates keyboard  access and increases efficiency.
1. Use the function, MoveToFrame, to position to  important locations within user-defined frames.
2. Use the function, MoveTo, to position to  important locations anywhere on the screen.  3. Use the function, FindGraphic, to position to  important locations associated with graphics.  4. Use the function, FindString, to position to  important locations that contain specific text.  V. Customizing information for Braille
A. Modify BrailleBuildLine to combine relevant information  in different circumstances Such as braille build dialog,  menu, and other in the default scripts.
B. Modify BrailleBuildStatus to use  BrailleSetStatusCells to provide context dependent

information such as attribute information.
C. Add informational messages.
1. Provide custom versions of HotKeyHelp  (INSERT+H).
2. Provide custom versions of ScreenSensitiveHelp  (INSERT+F1).
VI. Naming application-specific scripts and functions:  A. If it has a similar purpose as a script or function  in the default file, give it the same name, and don't  make an application specific key assignment.  The  assignment in the default keymap will invoke it.  B. Use the application specific keymap to assign keys  to scripts that are unique (and uniquely named) to that  application.  For example, see the use of F7 for Spell  Check in Microsoft Word.
C. If a script handles both normal and special case  situations, then have that script call the default  version from the DEFAULT.JSS for all normal situations  rather than duplicating the default code in the  application script file.  The application script should  contain the special code needed for the special  situations.
1. Use scripts to create actions that can be  attached to keys.
2. Use functions for creating subroutines that  will not need to be attached to keys.
A.	Make use of the Synopsis and Description fields when  creating scripts and functions.  They provide the text  spoken when in Keyboard Help mode.