Cary WinUV
ADL Programming Course

VARIAN

Varian Australia Pty. Ltd.

August 1998
Revised June 2000

Introduction

A brief history of ADL

"ADL" stands for Applications Development Language. This programming language originated with the introduction of the Cary DOS software in 1989. It was developed to enable Varian to provide flexibility of use for its UV-VIS-NIR spectrophotometers. This language allowed us to modify the behavior of the main applications software through ADL. We also were able to control and use external hardware that was capable of linking to our general hardware interface (the accessory control board). In Cary DOS the main interface software and the applications were written entirely in ADL. Since that introduction we have continued to support ADL on our DOS and OS/2 platforms through the ADLNEWS network.

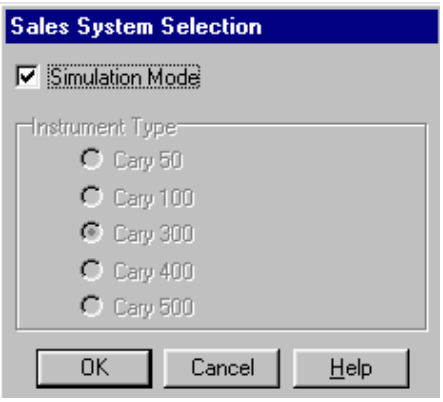
Computer language syntax and sensitivity

Whenever you are writing computer programs, the first rule to remember is that computers are not very clever at understanding what you want them to do. They need to be told in a language that they understand, very often with quite restrictive words and grammar. This is referred to as the SYNTAX of the computer language. Unfortunately we are still some time off from being able to easily communicate to computers through our natural language. Until that time we must all remember that we need to be very careful with the way we convey our wishes to the computer program. We must be generally careful of the case (upper or lower) of any text information that we use.

Alt-Y - running simple programs

It is always better to start slowly when working with computer programs. For that reason we will begin our course using the Alt-Y feature to show some ADL basics. We can perform a number of operations quite satisfactorily from this entry line. Cary WinUV allows single line and multi-line options for entry of ADL commands, allowing you from this easy interface to "write" programs that perform some very clever things without the need for you to understand all the details of ADL program structure. For most of this initial section we will use the multi-line capability of the Alt-Y entry dialog.

Firstly please start the ADL Shell program. This Icon can be found in the Cary Winuv folder.



When the program starts, please select simulation operation

When the application starts up please take time to examine the Alt-Y dialog (Press the Alt and Y keys together). Note that there is a Single line / Multi line selection. For our work we will use the multi-line option.

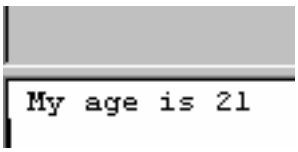
We will also use some simple ADL commands to allow us to print or view our results as we examine the basics of ADL.

The first command we will use is LPRINT.
The LPRINT command (from DOS and OS/2 ADL) is a simple command that allows us to print information to the Reports window of the Cary software. We will use it as follows to print a single item, to the report.

LPRINT(100)
Or
LPRINT("My Name")

The LPRINT command can print several things to the report on one line by placing commas between the item. For example

LPRINT("My age is ",21)



The second command we will use is CPRINT

This is similar to LPRINT except the information appears not in the report but in the "status line" at the bottom of the Cary application window. The command CPRINT is identical to the DOS and OS/2 PRINT command. The name has changed because of compatibility issues with the "engine" for WinUV ADL and Microsoft's VBA language.

We can print simple things as in LPRINT. For example

CPRINT(100)
Or
CPRINT("My Name")

The CPRINT command can print several things to the status window on one line by placing commas between the item. For example

CPRINT("My age is ",21)

Variables and Constants

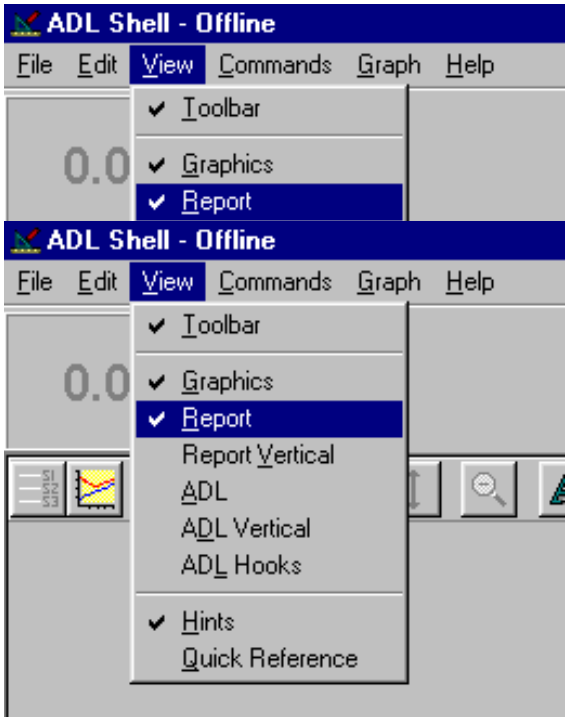
To make programs of a reasonable complexity level you will need to understand what variables are. Also you will come across things called "constants" and having an understanding of them will assist you with your programming.

Variables are computer memory storage items that can change their contents during the running of your programs. For example the following variable X gets given a value of 5.
X=5
This "assignment" may occur in one place in your program but later you may set X again as follows.
X=20
Which alters the value contained inside X. In this case X is a variable. It varies its value. The computer keeps track of the value held for you.

Constants remain fixed during the entire running of the program. Constants such as Yes, No, True and False are easily seen as values that do not change, hence the name "Constants".

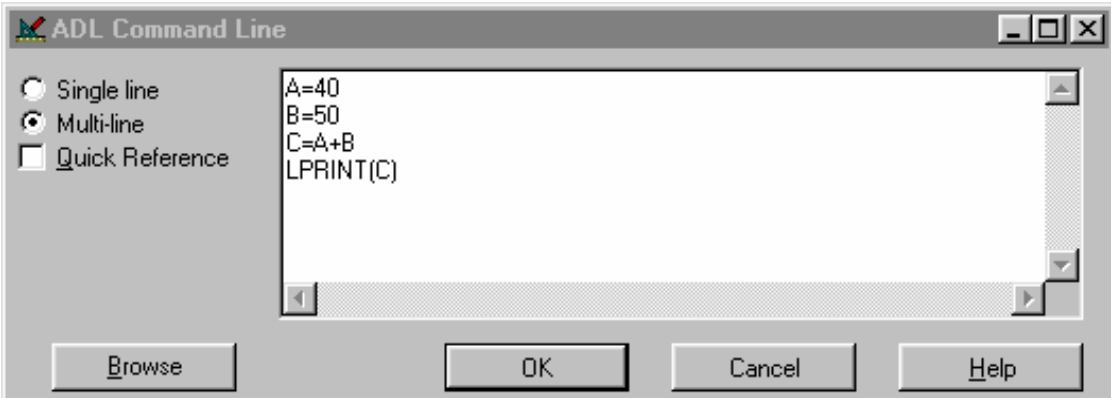
The name for variables can be assembled from the characters A-Z and the digits 0-9. Upper and lower case are NOT differentiated. For example the variable cary_spectrometer is identical to CARY_SPECTROMETER. Additional characters can added. Underlined _ can be used. All other characters (e.g. & #;,.: +*=() ^!) may not be used, as they cause program errors. Translation of the program into other languages can cause unknown problems. Note that very long names for variables can be assigned. It is advisable to assign however as short and meaningful names for your variables as possible. The variable name Cary_IntegrationTime indicates useful information in any program. Also note that some variable names may not be used, since they are already used by internal function/variables of the Cary software. You can find a list of such reserved variable names in the appendix.

Now let us try some simple Variable assignments in the Alt-Y dialog



Firstly make sure that the Reports window is visible as we will direct our results to that window. To do this, select the View menu, then make sure that the Report option is ticked. We will use one of the simplest Cary commands LPRINT to place our results into the report area.

Now enter the following lines into the Alt-Y dialog and select the OK button.



Look at the report to see what happened. The LPRINT command prints the information contained within the variable "C" to the report. You should see "90" on the report as this is the sum of 40 + 50.

Now we will test that the variables are not sensitive to the case of the characters. For example

AnyVariable=100
LPRINT(AnyVariable)

AnyVariable=100
LPRINT(ANYVARIABLE)

The result in both cases is the same

Now we will try to make a variable that the ADL will not understand.

Any Variable=556
LPRINT(Any Variable)
When we do this the program cannot understand the space between the word Any and the word Variable. The follwoing error dialog is show.

Errors

It is reasonable to expect that you will make some programming mistakes along your way. Don't be worried by this as ADL tries to show you where you need to change the program to correct the mistake. For example from high school mathematics we remember that when you divide one number by zero the result is not a reasonable value. E.g.

A=1
B=0
C=A/B

ADL will show you where in your program that the division by zero occurred. Also if you give the computer an instruction that it does not understand then ADL will try and identify for you the area where the misunderstanding occurs.

In the advanced section of the ADL course we will introduce "defensive programming" concepts which will allow you to handle error situations in a graceful programming manner.

Cary Instrument Control

Cary ADL has its strength in allowing you to control the Cary instrument and its accessories. You can perform scans or collect single readings at a wavelength. You can move the accessories around through your programs. You can also manipulate scans. You can generate reports from the data collected by your program or from stored Cary data. These are some of the many ways that ADL differs from conventional program languages.

ADL News

We have a large repository of ADL programs for general use. We publish these programs on our Internet site for you to peruse and download. These programs can provide you with a complete solution to your problem or they may provide you with a solid start to writing your own program. Feedback is always welcome. We are always happy to consider publishing your program for others to use.

What's the best way to start

Is it better to create your own program or modify an existing program? We would prefer that people learn ADL by writing their own programs from "scratch". By doing this you learn more about what the computer is expecting of you. However you will most likely find that there is a program already available that nearly suits your needs. In this case your fastest way to get to your goal would be through modifying the program and therefore building upon the ADL program base. Some of our Varian staff have been very successful in this way by simply changing a few small parts of a basic program to suit a new customer.

More on Variables

Lets take a look at the ways the computer will store the information you give it. The use of variables in programs is one very common way to temporarily save information during the running of your program. Another way would be to save information on computer disk in files. We will discuss that later in the course.

Single Variables

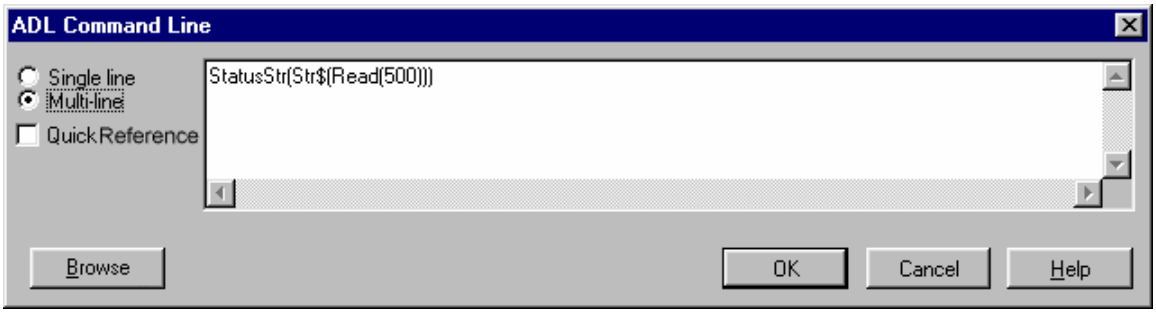
For now let's re-examine one of the simplest types of variables, the SINGLE numeric variable. This name is very "computerish" but don't let that put you off. If you just remember that numbers are SINGLE type variables then we can proceed. Imagine that you want to remember a number (e.g. 45) in your program. ADL will allow you to do this with a simple ASSIGNMENT as follows

A=45

This lets you remember the value (45) in the variable A. You can recall it later to use it in some way. Variable names can be any combination of text (A to Z) and numbers (0 to 9). For example valid variables are

MyFile, SampleNumber, BookMark123

Now that you have this knowledge lets do some simple things with numeric variables. Open the ADL command window (Alt-Y) of any application (choose ADLShell for example). Choose multi-line mode.



We will now put one assignment per line and do some simple arithmetic. Try typing in

A=10
B=20
C=A+B
CPRINT(C)

Then press OK and look at the status line of the Cary application (at the bottom of the Cary window). Did you get "30" as you would expect? Here we are using the CPRINT command of Cary WinADL to let us “see” the result of our addition. With SINGLE variables we can perform many types of mathematical operations. For example try using any of + - / * . and others

NOTE: We can also enter in Scientific notation. For example

A=1.2E+5

More complex math can also be done on SINGLE variables. For example we can take the logarithm of a number, a useful thing to do when changing Transmission data to Absorbance data. Try entering in the Alt-Y window the following

A=10
B=LOG(A)
CPRINT(B)

Then press OK and look at the status line of the Cary application. Did you get 1 as you would expect? No in this case you did not. You got 2.3025 approximately. This is because the LOG is not in BASE 10 but is in BASE "e", the natural logarithm. Don’t worry, we have a LOG10 function for you to use. Try again but using LOG10.

A=10
B=LOG10(A)
CPRINT(B)

Now you should get the value "1" printed. This is what we normally expect.

Strings variables

Another type of information that you want to keep is text information. This is referred to in WinADL as STRING information. For example, your name is only understood by the computer when it is stored as a STRING. We usually place the “$” character immediately after the variable name to let us (and the computer) know that we are using a string variable. This is not mandatory as there are ways to tell the computer that we are using a STRING variable with a simple name. We will address that a bit later. STRINGS are always enclosed between pairs of double-quote characters “”. Use of single quotes in WinADL will cause the computer to get a headache. Some examples of strings are

A$=”My Name”

SampleName$ = “Sample 1”

Notice that the last example mixed text and numbers. Numbers cannot contain alphabet characters but strings can contain alphabet and numeric characters. Because of the mix the computer has difficulty understanding anything other that the STRING variable. What do you think happens when we add STRINGS?

Enter the example we used for SINGLE variables but this time put the double-quote marks around the numbers to let the computer see them as strings. Enter the following lines in the Alt-Y multi-line entry box.

A=”10”
B=”20”
C=A+B
CPRINT(C)

Then press OK and look at the status line of the Cary application. Did you get what you would expect? The correct answer here is that the string values are **not added as numbers** but are **added together to make a string of the combined characters**. The answer printed is "1020". This is useful to make reportable strings. This way we can add text together to make a complete line composed of several pieces of information.

Cary ADL has many more variable types which I will briefly explain here.

Integer:

This numeric variable type is what is known as a 16 bit number. This means that the computer can store any counting number in the range from –32768 (-2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2) to +32768 (+2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2), or -2 to the power 16 up to +2 to the power 16.

Long:

This numeric variable type is what is known as a 32 bit number. This means that the computer can store any counting number in the range from –2 to the power 32 (approximately -2 million) up to +2 to the power 32 (approximately 2 million).

Single:

This numeric variable type is what is known as a 32 bit real number. This means that the computer can store any number including fractional parts in the range from approximately -3.4 times 10 to the power 38 up to 3.4 times 10 to the power 38.

Double:

This numeric variable type is what is known as a 64 bit real number. This means that the computer can store any number including fractional parts in the range from

approximately -10 to the power 300 up to 10 to the power 300. Because Double type has more range that Single type it is useful for precise calculations but the penalty is that calculations will take longer because of the extra precision.

String
This variable type contains text (or printing) characters. We can store many characters into a string variable.

Boolean
This variable type is what is known as a binary variable. This means that it can only store two possible results. In your program this usually represents the TRUE and FALSE results. Booleans are useful when we want to just have these two results. For example when we ask a question of the user and are expecting one of two results. We can check if they gave what we wanted (TRUE) or did not give what we wanted (FALSE).

Note that ADL can handle scientific input for numbers, such as 1.234E+21. This can be useful for applications where high scaling is required such as activity factors.

Other variable types that are possible can be seen in the help section of the ADL editor which we will examine later.

Some Variable operations

To do some simple math with numbers we can do the following.

Addition : C = A + B
Subtration. : C = A - B
Multiplication.: C = A * B
Division : C = A / B

Calculations can be performed concurrently, meaning we can get a result froma complicated expression written on one line, rather than breaking it down into small steps. For example we can do the following
*c= 5 + 2 * 3*
The answer is 11.
The computer uses normal rules for working out the answer. In this case it multiplies 2 times 3 AND THEN adds 5, rather than adding 5 and 2 and then multiplying that part by 3.
The rules of B.O.D.M.A.S. (Brackets Of Divide Multiply Add Subtract) apply. This means that the computer will work out athe answer by first working out the bits in brackets (B), then the divisions (D), then the multiplications (M) then the additions (A) and subtractions (S).

If we wished to make the computer first add 5 and 2 then we use brackets as follows.
*c= (5 + 2) * 3*

Now the rules of BODMAS apply and we first work out the bits in brackets (5 + 2), then multiply that by 3 giving 21.

Exponent: A=EXP(3)
This allows us to get the result of the Eulers constant (approximately 2.71828182845905) raised to any power. The answer here is 20.085.

Power10: A=POWER10(3)
This allows us to get the result of 10 raised to any power (in this case 1000).

Power: A=POWER(5,3)
This allows us to get the result of any base raised to any power (in this case 5 to the power 3 = 125).

Logarithms: A=LOG(100)
As in the EXP function ADL uses LOG of a number to meanthat it is the natural logarithm (base E). The answer here is 4.60517.

Log10: A=LOG10(100)
In this case the more common(?) logarithm of base 10 is used. The answer here is 2.

LogX: A=LOGX(100,5)
Here we can get the log of a number of any base. In this case the logarithm of base 5 is used. The answer here is 2.861353

Sqr: A=SQR(4)
This calculates the square root of a given number. In this case the answer is 2.

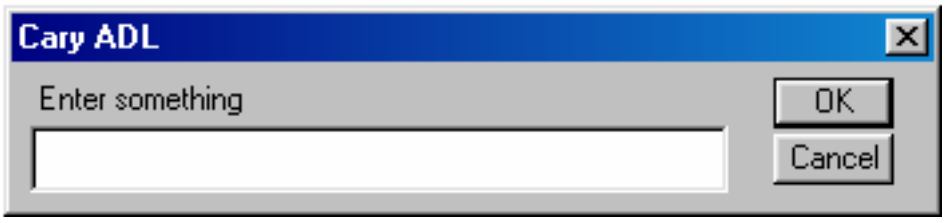
We have also included the universal number PI in Cary ADL as a constant number for your ease of use. As well there are many chemical and physical constants such as light_speed.

Simple Input

As well as getting your answers displayed with the CPRINT command we also want to take input from the keyboard. We have provided a simple command, CINPUT for you to use (similar to the old DOS and OS/2 INPUT command). Try the following commands

A=CINPUT("Enter something")
CPRINT(A)

Note how you are prompted with a simple dialog box.



Whatever you enter is placed into your variable 'A' for printing with the CPRINT command. Let's try another simple example to find the area of a circle when the user provides the radius as an entry value.

Radius = Cinput("Enter the radius of the circle")
Area = 3.142*Radius*Radius
Cprint(Area)

We have also included a constant value for PI in WinADL.

Radius = Cinput("Enter the radius of the circle")
Area = PI*Radius*Radius
Cprint(Area)

We can also change the title on the dialog box by adding an extra text string after the prompt string. For example

Radius = Cinput("Enter the radius of the circle" , "Radius Entry")

And as well we can provide the user with a DEFAULT value for entry. To do this we add a third optional item as follows.

Radius = Cinput("Enter the radius of the circle" , "Radius Entry" , "1.5")

The optional value can be any type of variable. In this example I have used a string but I could have use the number value instead. The effect is the same.

Radius = Cinput("Enter the radius of the circle" , "Radius Entry" , 1.5)

It is also possible in ADL to omit the optional parts. To do this we need to indicate that there is no value for an item. We do this by placing the comma separator as before but then instead of entering the value we put another comma to "move on" to the next value.

Radius = Cinput("Enter the radius of the circle" , , "1.5")

Here as we have no title string the dialog displays the default title of "Cary ADL". This use of optional parameters is very useful and you will see examples of these in many ADL commands.

We will use the CINPUT command for more examples to come. There are many other ways to get input from ADL and we will examine these later.

Converting between variable types

Sometimes you need to convert between one type of variable and another in order to let the computer be happy. To please the “computer gods” we use the computer commands STR$ and VAL for this reason. For example to get the actual numeric value of a string which appears to have just a number we can enter

A$=”100”
B=VAL(A$)

This converts the string inside string variable A to a number inside single variable B.

However if the string A$ contains text that is not a complete number then we have problems. For example

A$="Kim"
B=VAL(A$)

This will cause the computer to find an error trying to convert the text "Kim" to a number.

To convert a number to a string we can use STR$. For example

A=100
B$=STR$(A)

These conversions may seem pointless now but he computer knows best and will let you know when it needs information in one form or the other.

ADL has several other ways to allow us to convert between data types. These are preferable to use for several reasons. Not the least is that the naming conventions are easier to remember.

CSNG: A=CSNG(“123.4”)
Allows us to convert the text to a Single type number as the VAL command does.

CINT: A=CINT(“123.4”)
Allows us to convert the text to an integer number. The number will be rounded up or down to the nearest whole number. The example gives 123 (rounded down).

CLNG: A=CLNG(“123.6”)
Allows us to convert the text to a long type number. The number will be rounded up or down to the nearest whole number. The example gives 124 (rounded up).

CSTR: A=CSTR(123.4)
Allows us to change a number to a text string as the STR$ command does.

CBOOL: A=CBOOL(0)
Allows us to find out if the number (or text equivalent of the number) is zero (FALSE) or other (TRUE). The example gives FALSE.

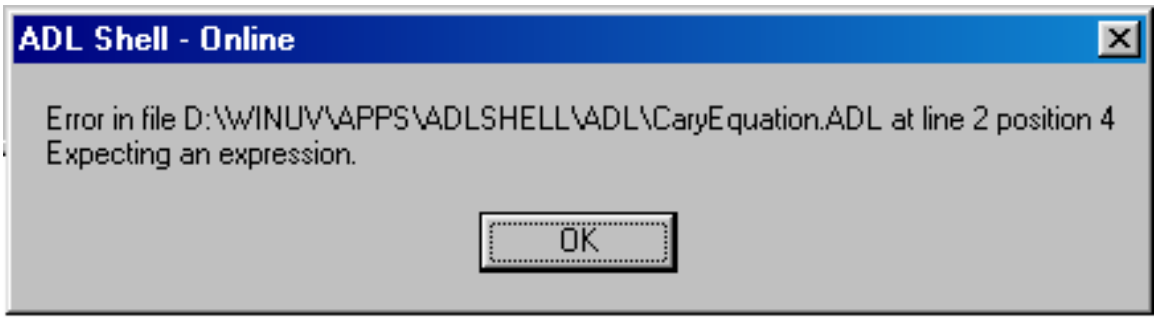
There are more that you may want to peruse in the SAX section of the ADL help.

Quotes, spaces and brackets

Up to now we should have been relatively safe with our simple ADL commands. However it's very easy to make mistakes by leaving something out or using slightly different wording. For example quotes are important to ADL. Try using single quotes

A$ = '10'

Problems ! We get the following error message



Also try using different brackets than the round ones ()

CPRINT[10]

Or

CPRINT{10}

More problems ! The first type of brackets will not cause an error to the SYNTAX checking but will give wrong results in your programs. The second type of brackets cause a SYNTAX error.
Just remember to use double quotes and round brackets for Cary ADL.

Controlling the Cary Instrument system

There are many WinADL commands that allow you to control the operations of the Cary system. For now we will examine just a few to gain some familiarity.

The Read Command

First lets look at how we can read the value at a particular wavelength. We use the READ command for this

A=READ(500)

This will read the value at wavelength 500 nm and give back the result.
Note that the result is in the current Y mode of the application.
Also note that the number in the brackets is always a wavelength in nm for this command.

To write this number directly to the report we use the LPRINT command.

A=READ(500)
LPRINT(A)

Or we could make it a bit more useful by

A=READ(500)
LPRINT("The result at 500 nm is ",A)

As we discovered before the LPRINT command can print many items on one line. Each item to be printed must be separated from the previous one by a comma character. The example we just did had one comma between the text and the variable A.
Each LPRINT command starts on a new line on the report.

This READ command is extremely useful when combined with simple mathematics to produce some common expressions. For example a 260/280 ratio can be done as

A=READ(260) / READ(280)

Note the combination of two readings and the division. We can also print this to the report as we collect the data by

LPRINT("The ratio is ", READ(260) / READ(280))

You can see examples of this type of ADL computation in the **UserCollect** fields of certain applications.

Other Cary controls

Often we want to change parameters of Cary other than the wavelength. To do this we will introduce the SETVAL and SETUPINST commands. In later sections we will look closely into the SETVAL command but for now we will just work with a few commonly used Cary properties. These are the SBW, the averaging time, the data interval for scans, the beam mode, the lamp, the instrument wavelength and the detector gain. To set any of these we use the command

SETVAL(Item to set, Value to set)

We use items called “Common something” where there is a dual possibility for our Cary 500 system. For all other instruments it is good practice to keep using these common items, as your program will then work on all Cary models.

SBW:
SETVAL(“Common Slit Width”, 1.5)

Averaging time (in seconds):
SETVAL(“Common SAT”, 0.2)

Data Interval: (scanning)
SETVAL(“Common Interval”, -1.0) ‘ note the interval for wavelength scans is negative, a change from DOS and OS/2

Beam mode:
SETVAL(“Beam Mode”, Beam_Mode_Dual_Single)
SETVAL(“Beam Mode”, Beam_Mode_Double_Fixed_Slit)

Lamp:
SETVAL(“Source”, Source_Uv)
SETVAL(“Source”, Cary_Auto)

Instrument wavelength (nm):
SETVAL(“Goto Wavelength”, 456.7)

Detector gain (Photomultiplier only):
SETVAL(“Gain”, 100)

Each of the above examples will change the method settings that will be used for data collection. In order to get the Cary to set to these conditions we use the SETUPINST command. For example, combining a few of the above items we can-

SETVAL(“Common Slit Width”, 1.5)
SETVAL(“Common SAT”, 0.2)
SETVAL(“Beam Mode”, Beam_Mode_Double_Fixed_Slit)
SETUPINST

Note the SETUPINST command is used. This sends the conditions that the SETVAL commands have created to the Cary instrument.

Now when we issue the READ command the new conditions apply.

We will leave scanning for the general section of the ADL course.

Arrays and the DIM statement

Imagine that you want to keep all your information inside one variable name (e.g. Samples). There is a term called ARRAY variables that allows you to do just this. To understand what this means, we will use the analogy of a building. Suppose that we want to keep boxes on different floors of the building. The floor number is one way that we can easily identify what is on what floor. For example we could refer to each box by combinations of the floor number and the variable Box. E.g. the box on floor 1 could be Box1, the one on floor 5 is Box5 and so on. This would give us many slightly different labels, one for each floor. It can be more convenient (in programs) to use just one name and have some identifier for the floor. The simple way that the computer can understand this is with ARRAY variables. The computer needs one name and needs to know how many floors we wish to use. The computer syntax for (say) 10 floors is

DIM Box(10)

Note the use of the round brackets.

The key parts here are the word DIM in front of the variable name and the round bracket pair after the name. The number between the brackets is the allowable number of floors we wish to refer to. In this case, 10.

The DIM statement explains clearly to the computer that you want to use multiple items within one name. To refer to any box on any floor in our building (say floor 3) we simply use

A=Box(3)

Let us try a simple example. Open the Alt-Y multi-line box again and enter

DIM Box(3)

Box(1) = 10
Box(2) = 20
Box(3) = 40
C=Box(1)+Box(2)+Box(3)
CPRINT(C)

When you press OK the sum "70" is printed.

The power of these arrays is more apparent in examples where the array is much larger than 3 items and we have smarter ways to add up all the items in the array Box.

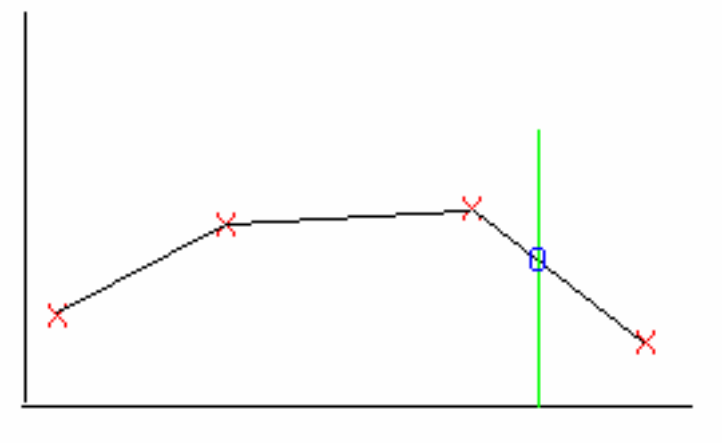
Don't get too worried if you can't immediately follow this section. We will come back to it later and examine some of the stranger parts of array declaration and usage.

Continuum variables

This is just a brief introduction to Cary specific scan type data. We will use it more in the second part of the ADL course. Continuum data was created by us to allow collection and processing of large amounts of data from Cary instruments using very simple commands. The way that the Cary instrument collected the data is hidden from your eyes, just the results are given. For example we can collect a wavelength scan and put all the data inside a continuum variable. Then in one command we can plot that entire scan data. We can also convert to different "Y modes", perform smoothing, derivatives and many more operations.

Continuum data is like an array of all the data collected by a scan. It has some special requirements placed upon the data. The data must be monotonic. That means that each new X value must be different from the previous X value AND it must be moving in the same X direction for all points. For example if we collected data at wavelengths 500, 450, 400 and 350 nm we can place these points into a continuum as they are collected. However if we change the collection order to 500, 300, 400, 450 the data "reverses" at the third point (400). This is not allowed.

When we have successfully placed the data into a continuum we can for all purposes forget how the collection was done. If we wish to know the data value at any point in the range of stored X values then the continuum will give up the value that corresponds to that X value. It does this by linearly interpolating between adjacent X values to find what it thinks is the most likely Y value.
In the following picture the red crosses represent real data. The straight lines drawn between them indicate the linear interpolation between adjacent points. The vertical line marks the point at which we want to find out the Y value. The blue circle on the interpolated line will be the point that we can get by interpolation.



Simple messages

WinADL provides you with a simple means to put up a message box with an OK button. For this we use the MSGBOX command. Try the following example.

MSGBOX "Hello from ADL"

Or another example is

A$="My Result is 1.234"
MSGBOX A$

There are other variations on this command with OK, Cancel, Yes and No button options. These use the following constants.

VbOkOnly, vbOkCancel, vbAbortRetryIgnore, vbYesNoCancel , vbYesNo, vbRetryCancel,

The MSGBOX can tell us which button was pressed using the following constants.

VbOk, vbCancel, vbYes, vbNo, vbAbort, vbRetry, vbIgnore

To make a simple dialog with an OK and a Cancel button we make use of the optional parameters that this MSGBOX command can have.

A$="Do you want to continue? "
MSGBOX A$, vbOkCancel

To test the returned value from MSGBOX, i.e. which button was pressed we need to understand Flow Control.

Flow control

Programs generally do not run line after line but take different routes depending upon the decisions that the program must make. This area of programming is called “Flow Control”. For this section we will use a variation on CINPUT called VINPUT. This command always gives us NUMBERS.

IF THEN ELSE

The first type of flow control we will examine is testing the result of some operation and deciding to choose one path of two. This is done with the IF THEN statements. For example

```
A=VINPUT("Enter a number")
B=0
IF A=0 THEN
  B=1
END IF
CPRINT(B)
```

Seems simple enough. The key parts here are the test to see if A=0, the use of the word THEN to decide what to do next and the closing part END IF. Everything between the THEN and the END IF will be done if the statement A=0 is correct.

If we add the key word ELSE we can choose one of two outcomes

```
A=VINPUT("Enter a number")
B=0
IF A=0 THEN
  B=1
ELSE
  B=2
END IF
CPRINT(B)
```

In our simple example A will be set to the value you enter. Depending upon if you enter zero or a non-zero value we set and printout B to 1 or 2.

The computer is capable of the following comparisons

=	*for equality*
<	*for less than*
>	*for greater than*
<>	*for not equal to*

Let's revisit our MSGBOX example from before. We had the following code.

```
A$="Do you want to continue? "
MSGBOX A$, vbOkCancel
```

To find out which button was pressed we use an IF statement.

```
A$="Do you want to continue? "
If MSGBOX(A$, vbOkCancel) = vbOK then
  MSGBOX "The user pressed the OK button"
Else
  MSGBOX "The user pressed the Cancel button"
End If
```

The MsgBox command should be used whenever you want to get a simple response (not a number or text) from your program's user. E.g. you may want to ask the user if they are ready to load a sample, or run a baseline, or if they want to print an save the data.

FOR NEXT loops

In situations where we want to perform some repetitive operation and we know in advance how many times we want to repeat the operation we use a FOR NEXT loop construct. The construct needs to know the start and end numbers and also a numeric variable to keep the count within. The syntax is-

```
FOR  VariableName = StartPoint TO EndPoint
   ' Do some things
NEXT
```

For example enter

```
For I = 400 to 450
  Cprint(I)
Next
```

And watch the status line rush along printing numbers from 400 to 450.

To add up all values between 1 and 100 we can do the following

```
Sum=0
For I = 1 to 100
  Sum=Sum+I
Next
CPRINT(Sum)
```

To read all wavelengths between 400 and 450 nm and display the results we could do the following (if we have an instrument connected)

```
For I = 400 to 450
  A=Read(CSng(I))
  Cprint(A)
Next
```

Note that the loop variable is always an integer type variable

WHILE WEND loops

In situations where we know the start point of a repetition but the ending is to be determined by the result of something that occurs while repeating our section of program we use the WHILE WEND statement pair. For example

```
A=0
While A<10
  A=A+1          ' the section to repeat
WEND
CPRINT(A)
```

This loops around starting at variable value A=0, incrementing A until A = 10 then it prints out the end point of our loop. Note that we check the test A<10 BEFORE we enter the repeat section where we increment the variable A. More on this later.

For a more useful example we could read the data from Cary every 10 nm between 400 and 450 nm with the following code.

```
UserA=400
While UserA<450
  Cprint(Read(UserA))
  UserA=UserA+10
WEND
```

DO LOOP UNTIL loops

Like WHILE loops we can also arrange a loop that tests after it has done at least one time through the section between the beginning and ending. The difference with this loop construct is that we must pass through the loop repetition section at least once before we perform our test to enable us to exit or continue. For example

```
A=0
DO
  A=A+1
LOOP UNTIL A>=10
CPRINT(A)
```

Or a simple result screening example would be

```
DO
  A=Read(500)          ' take a reading at 500 nm
  Lprint("The result is ",a)
LOOP UNTIL a>1.0       ' screen results until a high reading is found
```

This gives the identical result to the WHILE WEND example but in a different form.
Don't worry if you never find a need to use these, but just in case, there they are.

To avoid unnecessary typing, all these constructs are available for rapid insertion in the ADL Quick Reference dialog.

Early exiting of loops

Occasionally we want to stop running around a program loop depending upon some condition that has occurred. In order to do this we use the EXIT command in conjunction with a simple test within the loop. For example if we were reading data at all wavelengths between 400 and 450 nm but wanted to exit as soon as the Absorbance was over 1.0 Abs then we could do the following.

```
UserA=400
While UserA<450
  B= Read(UserA)
 If B > 1.0 Then
   Exit Sub              ' leave this loop when the Abs reading is greater than 1.0
 End If
  Cprint(B)
  UserA=UserA+10
WEND
```

There are many ways that the EXIT command can be used to leave loops and other WinADL program sections. We will leave the subject of EXIT use until later. For reference there is a QUIT command that will terminate the entire application.

Inserting comments

If you want to put some notes in your program but want the computer to ignore them then we use the REM statement or the single quote ‘ character. Anything after these on the same line is not seen as program information.

ADL General Course

Introduction to WinADL

VBA compatibility and the future of ADL

VBA stands for Visual Basic for Applications. Visual Basic is a Microsoft programming product that allows creation of very sophisticated programs from a very visual direction. Users do not have to be programming wizards in order to make quite appealing programs. VBA is Microsoft's language that they derived from Visual Basic. They use VBA inside their key programs (Word and Excel) to allow advanced users to customize the way that the programs operate. For example you can place buttons on a spreadsheet that when pressed perform some manipulations on the data in the spreadsheet.

Why use VBA compatibility?

This language (VBA) is becoming widely accepted within the computer world because of Microsoft's platform dominance. We chose to develop the Windows Cary ADL using VBA supplied from SAX Software corporation in order to maintain compatibility with the general user software base. Also VBA is very similar to our DOS and OS/2 ADL languages. This makes the migration to VBA based ADL quite straight forward.

Changes to ADL for consistency and future growth

We have made some changes to our ADL in order to make our future ADL more consistent in its syntax and style. These changes are not too radical and so you should have little problems migrating your existing ADL programs to the new Cary WINADL platform.
As one safeguard we have tried to make our ADL insensitive to text case, unlike many computer languages. For example the following term can be used for the current instrument wavelength.

"CURRENT WAVELENGTH"
"Current Wavelength"
"current wavelength"

All these are equal. The only important part is the space between the words 'current' and 'wavelength'.

NOTE: The use of double-quotes is necessary for all string variables

Some backwards compatibility

Wherever we could reasonably do so we maintained compatibility with the existing DOS and OS/2 ADL language commands. Old faithfuls such as LPRINT and READ are still there as we have already seen. New commands have been introduced where the new environment is significantly different to the past. We will see more of these later on. Note there is a table of DOS vs OS/2 vs. WinADL in the appendix E. Look briefly at the table given to see the compatibility list. Please note that this is not the full list of WinADL commands, just those that have some backwards compatibility.

For most of this general course we will use ADL with the Quick Reference list in conjunction with a simple text editor. In the advanced course we will continue exploring ADL but using our more sophisticated "in-built" editor.

What has changed

Overall

In general we have made changes to graphics, continua and reports. We have defined new constants for use with the filing system, scan collections, database access and more. We will examine new commands as we look at each section.

What's the same

In the Instrument

Instrument control has not changed at all. All existing low-level commands are still available. Where we could, we have added new sections to make routine operations easier to do. Take time to examine the command set. We always are trying to take the difficult work out of programming for you. For example in order to perform a wavelength scan and have it graphed we have introduced a special command that takes the work out of making graphs, starting the scan and plotting the data. (See CollectAndGraph). There are a few new commands such as InstSet and InstGet that are useful for advanced WinADL.

In the accessories

The Cary accessory controller board commands are also unchanged from DOS and OS/2. The SPS has changes its command set slightly in name but still has all the old familiar operations. In the past we used commands like ADS_Park. Now we use ParkSps. The name change is more consistent with our use of the SPS hardware so we felt it was worthwhile for future growth in ADL programming to tidy up this area slightly. Again take time to look at the list. A list of accessory codes is in the appendix G.

With data

This is one area where we have had to make changes. There are two types of data that are available in DOS, OS/2 and now in WINADL. Firstly we have our "database" values such as the current wavelength and the slit width. Then we have our familiar collected data that we refer to as CONTINUUM data. Both of these have changed somewhat since OS/2 and DOS. However we still have the same (and in many cases, more) functionality as before. Just the names and type of access may have changed. We no longer support the DEPOSIT and RECALL commands of older ADL. However the WinADL commands perform the same functions and more.

Database data still exists but where we used indexing to get values we now refer to values by a string name (see appendix B). For example we reference the current wavelength value by the string "Goto Wavelength". SAX Basic doesn't let us use SET and GET as we did before so we implemented SETVAL and GETVAL to do the same. We use the SETVAL command to change the value and the GETVAL command to examine the value in this case. For example to look at the value we use

A=GetVal("Goto Wavelength")

To change the value to 560 nm we use

SetVal("Goto Wavelength", 560)

We refer to each entry in our database as a NODE. A complete list of Cary database node names is available in the appendices. It is also available from the ADL word list as seen in the dialog selectable from the Alt-Y dialog, multi-line version. Try reading and printing the values of a few items.

Nodes can be many different types depending upon the type of information that we keep. For example we keep wavelength data, batch names, Y mode choices etc. We will see more of this in the re-visit to variable types.

Numeric nodes have allowable entry ranges, formats for display and many more capabilities. They form the basis of each application METHOD.

With files

Files in Cary still provide for methods, reports, continua etc. We have provided the usual commands to RETRIEVE and ARCHIVE your data from ADL. As well there are more alternatives available for simpler program interaction. The internal structure of the files is very different from previous DOS and OS/2 files but this is of no concern to ADL programs. The ADL view of data is unchanged.

With reports

Operations to make reports are basically the same as they were in DOS and OS/2. We provide the same means to clear a report, insert text and tables and as well we have added much more, such as inserting graphs into reports and printing special headings on reports. Your old ADL programs should run with very little changes needed. The main change is that we use the VBA standard for specifying text format options for tables. More on that later.

Comments

The use of the "{ }" comment characters is gone. We now use the standard VBA comments of either the keyword REM or the single quote character '.

Alt-Y quick reference mode

Within the Alt-Y dialog we now have single line and multi-line options for command entry. The single line entry has a history list to easily allow you to recall some previous commands rather than type them in again. When you select the multi-line option you also get the option to bring up the ADL quick help dialog. From that dialog you can select items to be transferred into the multi-line dialog. The items include some basic Flow Control syntax, the available Cary constants for file types, X and Y modes and others, a list of the Cary extensions to SAX Basic, and a list of the database node names for the application that you are currently running.
This quick reference also has a direct link to the Cary ADL HTML help file. Take time to examine the list. In order to make operations easier you should practice using the list to insert simple commands and as well more complex structures such as FOR NEXT loops. Using this insertion will prevent most common typing errors in programs.

Searching:

Search mode 1: Try pressing a keyboard key such as 'R' when focus is on the list box. Note that it moves to the next word beginning with 'R' **with each press** of the 'R' key. This can be useful if you want to find something, for example the "Read" command. Also we have included a search facility on the entry line.

Search mode 2: Try the following. Highlight or clear the "Insert into ADL" line. Then type the letter R as before. The software finds the first match with your entry. Then type E (now you should have RE) Again the search finds a match. Then complete the line by typing READ(and we see that the software has found the command we wanted. Then just click in the command in the list box to accept the match. Now it is ready to be inserted into our program. This mode of searching is more useful if you remember part but not all of the command.

If you have ADL program sections that you also want to add into the ADL command list then you can place these inside a file named "ADLUserExtensions.txt". This file must be in the installation folder for Cary. For example I can create a new command that lets me set the SBW and SAT as part of one command. This new command could have the name of SetSBWAndSAT (with appropriate code and structure). I can then place it inside this file (ADLUserExtensions.txt) then it appears in the list immediately before the section headed by the marker "REM Database names<--------". Now I have a new command available for easy insertion.

NOTE: We also have on-line ADL help about the VBA environment as well as the Cary extensions to VBA than make it into ADL.

Finding errors

If you make a programming error that is detected when you run the program a dialog will appear giving the line number and position of the line where the computer had a problem understanding you. If you ran the program from the ALT-Y line then the line number given to you is shown as one more than the actual line where the fault is found. This is because the Alt-Y line automatically puts a previous line that is

necessary for the commands to be understood. You never see this line but it is line number 1. The first line in the Alt-Y command area is line 2. Try entering something that the computer will not understand, for example your name.

Output

SAX Basic, the base for WinADL does not permit us to use the PRINT command as they have a special use for it so we changed ours to CPRINT as you may remember from earlier parts of this course. It works the same, just looks a bit different. You can still use CPRINT with several items to print as in our earlier example. The items may be any valid variable types and the command understands how to print them to the Cary application.
Note that this command CPRINT writes to the screen, not the printer. Printing information will be examined later.

Input

SAX Basic does not permit us to use the INPUT command as they have another use for it so we changed ours to CINPUT. It works similarly to before except where you used to do the following

PRINT("Enter a value")
A=INPUT

You now do both on one line as

A=CINPUT("Enter a value")

Try it and see the difference. The prompt appears as a dialog rather than on the status line.

Try it combined with CPRINT, e.g.. (use multi-line Alt-Y)
A=CINPUT("Enter a value")
CPRINT(A)

CPRINT has extra bits (options) which can change the title of the dialog as well as giving a default value on the entry line. For example

A=CINPUT("Enter a value","Number entry","100")

Or
A=CINPUT("Enter a value", ,"100")

We also have a special version of CINPUT that works just with SINGLE values. Try **VINPUT** with the same syntax.

NOTE: The difference is that CINPUT can accept STRING and SINGLE information whereas VINPUT just gives back SINGLE information.

Constants

We have added many constants to WinADL. A large number of these are listed in the appendices and can rapidly be inserted into your program from the ADL quick reference dialog. They provide a convenient and easy to read way to refer to items such as the instrument beam mode selections. Bring up the Alt-Y line and type

CPRINT(Wavelength_Xmode)

Then press the OK button. Note on the status line the value shown.

Try some others from the appendices (Minutes_Xmode, Temperature_XMode)
In general we have use the '_' character with constants.

Graphics

Apologies in advance but Graphics in WinADL is much different to graphics in DOS and OS/2. However we believe that it's much easier now to make graphs than before. We use the application that you run ADL from to display the graphs. For example if you are running the SCAN program then that's where we create graphs for ADLs run from SCANs Alt-Y line.
For some background Graphs and Continua are very closely related. The graph labels and modes actually come from the Focused Continuum, rather than being properties of the graph. The scales can be independently set (GraphScales). Graphs can also be sized (SizeGraph) and positioned (PositionGraph). We still have the PLOT command as before. The concept of the Focused or selected graph and trace is continued from OS/2 to WinADL.

Creating a graph

Graphs are referred to by STRING names as they appear on screen. For example in SCAN, the graphs are named "Graph 1", Graph 2" etc. We can create graphs with the NEWGRAPH command. This command gives you back a string to use (if you wish) in place of the full name of the graph. Create a simple graph as follows.

A$=NewGraph("My First Graph")
Cprint(A$)

It's that easy to make one. There are obviously more advanced capabilities such as placement, sizing, arrangement, annotations and linking to continua that we will address later. These functions include -

PositionGraph, SizeGraph, NewAnnotation, GraphScales, GraphPreferences, ShowCtm

Removing a graph

Just as you easily made a graph you can delete it with REMOVEGRAPH. Try the following

RemoveGraph("My First Graph")

We have added functions to allow copy and paste using the clipboard.

GraphCopyToClipboard
GraphPasteFromClipboard

Continua

Sorry but there are significant changes here. SAX Basic does not allow us to define “natural data types”. That means we can’t let SAX Basic know about our Cary Continua. However all is not lost. We have a solution that allows you to do all you could before with a small inconvenience. Firstly you need to understand that like Graphs, we use STRING names for continuum data. For example

Ctm$=NewCtm(“My First Continuum”)
Cprint(Ctm$)

By itself its not much use but we can do lots to it. We can fill it using AddCtmData, change its display label with SetCtmXLabel, place it in a graph with ShowCtm or Plot, delete it with RemoveCtm and many more operations. We will leave some of these till later. We also have the old CurvefitData commands. For better program readability these commands also have other names, for example AddCtmData is more clear than Curvefitdata.

Reports

We still use the LPRINT command to send many things to the report. We also use the TABLE set of commands as before. The format specifiers (the bits that tell Cary how to treat the data for formatting) have altered. We now use the SAX Basic format specifiers. In general you can use the "#" character where we used the "D" character and “0” where the “Z” character was before. If you want more options they are available and are listed in the ADL help. For example

TABLEFMT("#.###","###.#",6A")
TABLEHDR("Result","Wavelength","Units")
TABLEDATA(1.2, 456, "g/l")

Or

Lprint(Format$(Read(500) , ”#0.000”))

Simple Formatting

The simple rule to formatting numbers for display is to decide how many characters are required after the decimal place and also if leading zeros are needed. Then translate this to the format string. The simplest translation is to use the “0” character as a place marker. For example we want three characters after the decimal place and at least one leading zero. This translates to a format string of “0.000”, being one marker before the decimal point and three after the decimal point.

The STARTPRINT, TABLE and LPRINT commands still exist. We have added new functions to allow printing report headings, setting the report body font, copy and paste using the clipboard. The Table command is useful for printing XY or Peak tables. The syntax is still

Table(“The Name Of the Trace”, Xlow, Xhigh, Xinterval ,TableType)

Where TableType = 0 for XY tables and 1 for Peak tables (based upon the “Peak Style” and “Peak Threshold” database items.

REPORTMASTERHEADING("Start of my report")
REPORTCOPYTOCLIPBOARD
REPORTPASTEFROMCLIPBOARD

Filing

Filing dialogs are provided to make this easier. ARCHIVE and RETRIEVE are still available to work with Data_Files.
Some of the new commands are

SAVEAS("File Name",Data_Files)

SaveAs allows us to save Cary information as data, methods, reports, batches, ASCII, Grams or other formats. See the quick reference on File Types for all allowable output types. ARCHIVE is a sub-set of the SAVEAS command with DATA_FILES as the given type.

OPENAS("File Name", Method_Files)

OpenAs allows us to read in any of the file types that are in the quick reference File Types section. For example we can read in a method, or read in a batch of data. RETRIEVE is a sub-set of OPENAS with Data_Files as the given type.

As well as these direct commands we can display the Cary open and save dialogs to allow interactive file operations. The commands ShowOpendDlg(xxx) and ShowSaveDlg(xxx) display the appropriate dialogs. The xxx part refers to the possible selection of the file type that appears as the "default" file type when the dialog us show, For example xx could be Batch_Files. A special feature of the ADLShell allows us to set the default file extension. Normally ADLShell starts with the extension of "x??" where the x is B for batch, D for data etc. The ?? allows the shell to see files from any of our Cary applications such as Kinetics (KN) and Scan (SW). This default can be changed with a special Cary variable called FileExtent.

FileExtent = "SW"
ShowOpenDlg(Data_Files)

Will bring up the Open file dialog with the default of batch files for the scan application.

Database

As we mentioned before Cary database consists of NODES. Each node has a type as defined by a Cary constant (see Appendix A). Types such as String_Node, Real_Node, Integer_Node are known by Cary. Nodes contain a value of the appropriate type. Where applicable they contain allowable range information and formatting style. Nodes can be created by WinADL and as such will extend the method of that application. Mostly you will find that your ADLs only require you to read and write to existing nodes. These may be the X and Y modes, the instrument controls mentioned previously and some others. E.g. to set the Y mode to Abs mode

SETVAL("Y Mode", Abs_Ymode)

Or to set it to percent transmission

SETVAL("Y Mode", PercentT_Ymode)

To make a new node is very easy. Just tell ADL the name of the node and the type of the node. Try a simple String node.

AddNode("My First Node",String_Node)

You can then use

SetVal("My First Node","The contents")

Or

A$=GetVal("My First Node")
Cprint(A$)

For more complicated types such as numeric nodes lets have a go.

AddNode("My Second Node",Real_Node)
Cprint(GetVal("My Second Node"))

You can set the limits of this node to range from -100 to 999.9 as follows

SetNodeLimits("My Second Node",-100, 999.9)

You can change the default format of 2 decimal places when formatted to 3 decimal places. To do this use

SetNodeFormat("My Second Node ","%9.3f")

The format specifier used here is not the VBA format specifier but is a low level software type. See the appendix D for information on these. More advanced users can also experiment with full text and format strings.

You can also refer to nodes by numbers. This is somewhat faster access for intensive operations. To do this you do

NodeIndex = GetNodeIndex("My Second Node")
Cprint(NodeIndex)

Then you can use the SetValByIndex and GetValByIndex commands. Try these from the Quick Reference list.

Cary Dialogs

All dialogs that are common between applications can be shown from ADL. Lets take some time to examine them. Later on we will learn about interacting with dialogs. These dialogs all start with SHOW and end with DLG. For example ShowTracePreferencesDlg.

Lets try a few from the list. Bring up the Alt-Y dialog and find the dialog section. Try them. Note that many require a graph and some need a trace to be shown with. You may need to load a batch or data file first to use these. Choose Holmium Oxide or similar.

It's worth noting that we can tell if the user has pressed OK to exit the dialog by testing the "return value" of the dialog. This is done using the IF THEN construct as follows

IF ShowTracePreferencesDlg THEN MSGBOX "The user pressed OK"

Buttons

All buttons that are common between applications can be pressed from ADL. Lets take some time to examine them. These buttons all start with PRESS and end with BTN. For example PressSetupBtn.
Let's try a few from the list.

Try PressSetupBtn as a simple example of getting your ADL to interactively work. Load up an example file data file or batch file and explore a few other options relating to graphs and traces. For example PressAxesScaleBtn

For general access to any button or menu there is a ClickIt command. This command looks for the button/menu with the closest matching name and "presses" the button/menu. For example to press the Clear Report button use –

ClickIt("Clear Report")

Or

ClickIt("Clear Rep") ' would be enough text to find the command

The "AS" keyword and the DIM keyword

WinADL uses a special keyword "AS" to describe the relationship between names and the type of information the name refers to or contains. For example a variable MyVar by itself doesn't tell the program what type of information it contains. It may be a number or a piece of text. To explicitly tell the program we can use the AS keyword with the DIM keyword. The DIM keyword was also used in older ADL platforms for the same types of operations. However in DOS ADL it was restricted to the use of ARRAY variables only. Examples of the use of DIM and AS are

DIM MyVar AS String
DIM SampleNumber AS Integer
DIM Reading AS Single

These definitions then lock the use of the name to the type of information that is needed for that name.
Again the uppercase and lowercase use of these is not significant to the computers' understanding.

We can also combine this with ARRAY definitions to give

DIM MyLuckyNumbers(45) AS INTEGER

Or

DIM Results(5) as SINGLE

Note that we use round brackets here rather than the square "[ ]" brackets we used in the past DOS and OS/2 ADL programs.

You will see that the AS keyword can also be useful when building a larger program.

We can also make "multi-dimensional" arrays. For example we can reference places on a chess board by the row number and the column number. To tell the computer how to do this we use

DIM ChessBoard(8,8) as string

And refer to by row and column (first index and second index)

ChessBoard(1,1) = "Rook"
Chessboard(2,1) = "Pawn"
ChessBoard(1,2) = "Knight"

Before proceeding we need to be comfortable with the Flow Control and SAX Basic general syntax discussed in the beginning course. Please refresh your grasp of these at this time. In particular the use of IF THEN ELSE tests and FOR NEXT loops as these are generally the most commonly used Flow Control statements.

Program structure

What makes a program

A WinADL program is composed of several sections. In the smallest configuration you must have a MAIN section. This section is the part of your program that will be run (executed) when you load it from disk (using the File Open menu, or Drag and Drop or Icon association). We would typically write a program and save it to disk if it was significantly more than a few lines that could be remembered in the Alt-Y dialog, or if the program is needed elsewhere. For that reason we need to understand more about programs than the Alt-Y allows.

A program may also have several sections, or SUBroutines as they are referred to. In earlier ADL platforms we used the DEFINE and ENDDEF keywords to nominate each section of program. We could make a larger program by combining several subroutines. For example we could have written a program as follows

```
SUB  DoTheStartupBits
  {put ADL code here to initialize the program}
END SUB

SUB DoTheCollectionBits
  {put ADL code here to collect data}
END SUB

SUB DoTheReportBits
  {put ADL code here to generate a report}
END SUB

SUB DoTheSavingBits
  {put ADL code here to save any collected of created data}
END SUB

SUB DoTheWholeProgram
  DoTheStartupBits
  DoTheCollectionBits
  DoTheReportBits
  DoTheSavingBits
END SUB
```

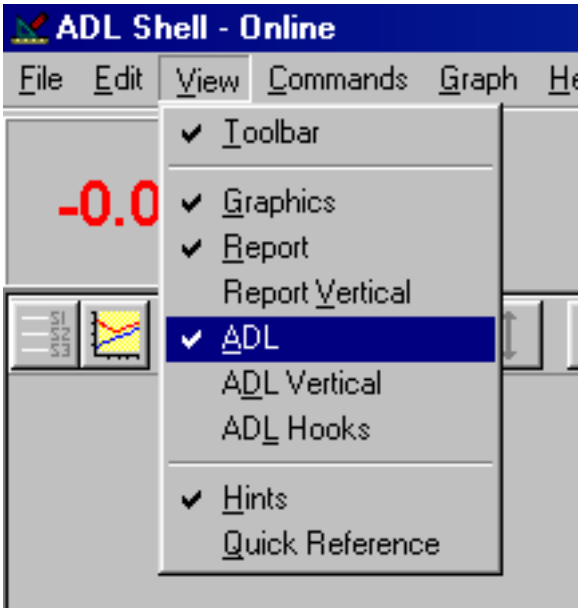
If the above example were run in DOS ADL the last section “DoTheWholeProgram” would be run first. That in turn would run in order the other sections listed inside that subroutine. Note that the names are usually chosen to describe actions.

Typically a programmer would divide the program up into subroutines that each would do a specific task. That way the overall program would be easier to write, examine and possibly change. However there is no requirement to do so, just good practice.

A program also can have an optional DECLARATION section where we state what types and names of variables that we want to use in our program. This becomes important to advanced ADL

We will now use the ADL editor to enter and save our programs.
WinADL has incorporated an advanced editor for ADL programming. This editor has many capabilities far beyond what you will need to write simple programs but in

addition it has some extremely useful features. Let's explore the editor. To start it up Select the ADL item under the View menu.



When the application has started you should see the ADL editor positioned under the report.
In the Cary main VIEW menu you can arrange the layout of these "windows".
You can also use the "drag handles" to resize the windows.
For now we will just use the editor as a simple editor. We could have used any text based editor to create ADL programs. In the later sections we will explore the capabilities of the editor.

The basic layout and syntax

SUB

Sub stands for SUBROUTINE as we previously mentioned. In WinADL this is the equivalent to the old DOS DEFINE command. We create a subroutine similarly to the previous example, just replace DEFINE with SUB and ENDDEF with END SUB. For example

```
SUB DoTheStartupBits
  'put ADL code here to initialize the program
END SUB

SUB DoTheCollectionBits
  'put ADL code here to collect data
END SUB

SUB DoTheReportBits
  'put ADL code here to generate a report
END SUB

SUB DoTheSavingBits
  'put ADL code here to save any collected of created data
END SUB

SUB DoTheWholeProgram
  DoTheStartupBits
```

```
  DoTheCollectionBits
  DoTheReportBits
  DoTheSavingBits
END SUB
```

So far no real differences.

Sub Main

In WinADL there is one difference that must be observed. The starting point for the program is not the last subroutine listed but is the subroutine named **MAIN**. This is also similar to other programming languages. In the example above we need to replace the last section name “DoTheWholeProgram” with “Main”. E.g.

```
SUB DoTheStartupBits
  ‘put ADL code here to initialize the program
END SUB

SUB DoTheCollectionBits
  ‘put ADL code here to collect data
END SUB

SUB DoTheReportBits
  ‘put ADL code here to generate a report
END SUB

SUB DoTheSavingBits
  ‘put ADL code here to save any collected of created data
END SUB

SUB Main                    ‘ the only change is the name Main
  DoTheStartupBits
  DoTheCollectionBits
  DoTheReportBits
  DoTheSavingBits
END SUB
```

This program is now WinADL compliant

Function

Another type of subroutine that WinADL has is called a FUNCTION. This is also useful when writing programs.

Is it different to a SUB?

No prize for guessing that it is. A function is a special subroutine that will give back a result at the completion of it’s section of program. For example we have already used a Cary function to get the result of a reading. The READ command is a function that returns you the value of the reading. What occurs inside the READ function doesn’t concern us, just that we get the result. Also the READ function is designed so that it does not interfere with the rest of the program subroutines. No other data or variables are adjusted by it. The syntax of a function is

```
FUNCTION ItsName AS FUNCTIONTYPE
  ‘the section of code goes here
```

```
  ItsName = ResultToReturn
END FUNCTION
```

This is very similar to SUB but with one exception. That is the FUNCTIONTYPE of the function. This is given after the keyword AS which follows the name of the function. Remember the AS keyword was encountered with the DIM keyword. This allowed us to define the TYPE of the variables used. Here it allows us to define the type of result the function returns. There is no difference really in terms of its use. A simple example would be to wrapper the READ command and do a small calculation upon the result.

```
FUNCTION FactoredRead As SINGLE
  DIM MyResult As Single
  MyResult=read(500)            'an example wavelength only
  MyResult = MyResult * 1000
  FactoredRead = MyResult
END FUNCTION
```

This example, while trivial shows that we could extend the ability of the READ command to make it a factored read (factor of 1000).

Global and local variables

One feature of subroutines is that they can have parameters. You can provide them with information that they treat in a special way. In DOS ADL we only had what we called GLOBAL variables. That is, they were readable and changeable from anywhere in the program. OS/2 ADL introduced PRIVATE (local) variables. These were ones that could be hidden inside subroutines. The reason for this is so a subroutine could prevent other parts of the overall program changing its private variables. Let's examine a simple example to demonstrate the meaning of this concept. A two subroutine program could be

```
SUB NewOne
  DIM MyResult As Single
  MyResult = 1000
  CPRINT(MyResult)
END SUB

Sub Main
  NewOne        'print the number
  NewOne        ' and again
End Sub
```

In this example the variable MyResult is local to the subroutine NewOne. It cannot be "seen" from outside the subroutine and therefore cannot be changed from outside.

Remembering that the program starts with SUB MAIN. This program will print the number 1000 twice. A similar program could be

```
DIM MyResult As Single

SUB NewOne
CPRINT(MyResult)
END SUB

Sub Main
  MyResult = 1000
```

```
  NewOne        'print the number
  MyResult = 2000
  NewOne        ' and again
End Sub
```

This example will print two different numbers, 1000 and 2000. The reason here is that because the DIM MyResult line has been moved outside the SUB NewOne statement. It can then be changed from outside the subroutine. In the former case it cannot be seen from outside the subroutine. This concept is very useful to more advanced programming.

The place where the "DIM MyResult As Single" statement appears is called the DECLARATION area. This is important when we do the advanced editing course.

Giving information to SUB and FUNCTION

Our simple Factored read program could use some improvement. For example we cannot change the wavelength easily. However using the concept of PARAMETERS we can change this program. Parameters are values that we can give to a subroutine or function that it can use in its internal code section. For example we really should be able to change the wavelength.

```
FUNCTION FactoredRead(Wavelength As Single) As SINGLE
  DIM MyResult As Single
  MyResult=read(Wavelength)
  MyResult = MyResult * 1000
  FactoredRead = MyResult
END FUNCTION
```

A further step would be to provide a factor other than 1000. To do this we just comma separate the parameters on the FUNCTION definition line. For example

```
FUNCTION FactoredRead(Wavelength As Single, FactorValue As Single) As
SINGLE
  DIM MyResult As Single
  MyResult=read(Wavelength)
  MyResult = MyResult * FactorValue
  FactoredRead = MyResult
END FUNCTION
```

Now we can use this from our SUB MAIN as follows

```
Sub Main
  LPRINT(FactoredRead(560, 1000))
  LPRINT(FactoredRead(800,5000))
End Sub
```

This example will print in the report the two factored results of two readings, one done at 560 nm with a factor of 1000, the other done at 800 nm with a factor of 5000.

This concept of "parameter passing" as seen in our Factored Read example also applies to subroutines. For example we could do the same example as

```
SUB FactoredRead(Wavelength As Single, FactorValue As Single)
  DIM MyResult As Single
  MyResult=read(Wavelength)
  MyResult = MyResult * FactorValue
  LPRINT(MyResult)
END SUB

Sub Main
  FactoredRead(560, 1000)
  FactoredRead(800,5000)
End Sub
```

The result is identical, just the means is different.
The use of FUNCTION versus SUB depends upon the nature of the program that you are doing. If we wanted to further process the FactoredRead in the 800 nm reading then the FUNCTION construct would have allowed us to capture the result, modify just that one and then proceed. It all depends upon the requirements of your program.

Types of data

UserResult - a special link to Cary data

We have kept the UserResult and UserCollect concept in WinADL. The name used to be User_Result but we shortened it to keep naming consistency in our WinADL language. This special variable is a SINGLE variable that contains the result of the current READ value. Having a special name is useful in our advanced programming section when we want to modify the behavior of existing applications programs without wanting to rewrite the entire applications. Also WinADL has other special variables for linking to the Thermal Analysis program. For example that program can use the TM, DNALength, Temperature and other variables. We have many special

User variables sich as UserString, UserInteger, UserA, UserB … UserZ. These are global variables. This means that they can be used to remember information within an entire ADL program or even across a suite of ADL programs all run from within the one application.

General Program Structure

We generally start out developing any program by separating it into three major groups. Accordingly the ADL commands can be divided into these same groups. These are INPUT, PROCESSING (or CONTROL) and OUTPUT.

Program Structure

INPUT	Here you can get input for your program by direct interaction with your program user with the CINPUT statement. You can also get input from other sources such as interrogating the conditions of the instrument and accessories using commands such as GETVAL and MEASUREACC. Input also comes from data files by the RETRIEVE command or method files by the OPENAS command. Further input comes from interrogation of this information with commands such as HEADER, which reads the continuum header information, or EXTRACT which "reads" the "Y" value for any "X" position.
PROCESS	As its name suggests, this is where you use the input to control the system and process the data to produce your results. This can be as simple as the READ command which gets the current value of the ordinate or it may be as complex as a multi-cell temperature collection.
OUTPUT	Again the name OUTPUT describes what to expect. This section of your program deals with providing output of your processed results for display on the screen (CPRINT, PLOT, PLOTSYMBOL), sending to the printer (LPRINT, STARTPRINT).

It is also reasonable to add an Initialisation section to start the Cary in a known state and also to set any variables to known values. Output could also be split into

Developing a Program

Your program structure may not be simple to partition in terms of the three basics of a program. At times you want to process information as you collect it and output the results as soon as they are processed. Our multi-cell collections do just this. However it is a good idea to construct your program this way if you can. We will start with a simple example.

A Simple Example

Lets say that you want to develop a method of analysing a batch of samples for the existence of some compound. You are going to manually present each sample into the Cary for analysis. The program must analyse the collected data and produce a report of those samples which match our criteria and a list of those samples which are rejected. The results are to be saved on disk and a printed report is to be done. Later on we will use some accessories to present multiple samples, gather extra information during the collection for recording, and printing a report. The first place to start is with specification development.

List INPUTS

To commence this program we determine what Inputs the program needs to provide. Lets assume that the user knows how many samples to analyse and so we will stop the analysis after the samples are all done. We also need to know the criteria for testing whether the result is reported or rejected. We may also wish to know some specific user identification to be printed with the report. We will assume that results are to be saved automatically. Some internal inputs will be needed such as an indicator of the current sample number being analysed.

List PROCESSES

Having decided upon the inputs, we then identify what processing is to be done. For this example we will start with reading the Absorbance at the current wavelength. The Absorbance is then tested against our input value for reporting or rejecting.

List OUTPUTS

The outputs can be listed as the report printout, the saved results file and the display of results while the analysis is proceeding.

In starting this way we could very quickly write the overview of the program as follows. Enter the ADL editor and create a new “sheet”. Type in the following:

Step 1

```
SUB GET_INPUT
END SUB

SUB DO_PROCESS
END SUB

SUB DO_OUTPUT
ENDSUB

SUB THE_START
  GET_INPUT
  DO_PROCESS
  DO_OUTPUT
END SUB
```

As you see this follows the standard three steps. We will now fill in the empty definitions to build the program.

For the GET_INPUT definition we will firstly initialise any special counters that we need and then prompt the user for the required input

Step 2 User input

```
SUB GET_INPUT
  STARTPRINT          ‘ CLEARS THE REPORT
  BATCH$=CINPUT("PLEASE ENTER BATCH DESCRIPTION ")
  LPRINT("BATCH : ",BATCH$)
  USERNAME$=CINPUT("PLEASE ENTER YOUR NAME ")
  LPRINT("OPERATOR : ",USERNAME$)
  ABS_LIMIT = CINPUT("REPORT RESULTS FOR ABS ABOVE WHAT VALUE ")
END SUB
```

We a must now tell the computer about variables that we want for global access. To do this we put a series of DIM statements at the top of our program (line 1)

```
DIM SAMPLE_LIMIT AS INTEGER
```

```
DIM ABS_LIMIT AS SINGLE
```
We did not need to DIM the Batch$ and UserName$ variables as they are used immediately in the SUB and never needed again.

We also need to know that the system is set up for Absorbance and Sample number analysis. To do this we could either load a stored method or use the SETVAL command to select relevant modes and ranges.

Question : Which do you think is more suitable and why ?

We will use the SETVAL command here because it is more interesting and also makes the program independent of stored method.

Step 3 Add Controls

Now we add the controls for setting modes.

```
SUB GET_INPUT
  STARTPRINT          ‘ CLEARS THE REPORT
  BATCH$=CINPUT("PLEASE ENTER BATCH DESCRIPTION ")
  LPRINT("BATCH : ",BATCH$)
  USERNAME$=CINPUT("PLEASE ENTER YOUR NAME ")
  LPRINT("OPERATOR : ",USERNAME$)
  ABS_LIMIT = CINPUT("REPORT RESULTS FOR ABS ABOVE WHAT VALUE ")
  SAMPLE_LIMIT = CINPUT("ENTER THE NUMBER OF SAMPLES ")
  SETVAL(“Y Mode”,0)        ‘ ABSORBANCE MODE
  ‘ Note that we can run in any X mode as we will be creating our own data and plotting it accordingly
END SUB
```

Step 4 Compile the program

This is enough to start our program off. We will run it straight away, eliminating any simple mistakes and seeing if the setup conditions are O.K. Press the F5 key or the run toolbar button. You should see it check the code for errors, then prompt you for the batch name when it runs.

Step 5 Simple processing

As a means of testing the program we will implement a very simple processing algorithm which is typical to many situations. We will read the Absorbance at two wavelengths and produce a number according to a simple scale factor plus sum. Go back into the ADL editor as before and move to the DO_PROCESS definition. Add the line:-

```
SUB  DO_PROCESS
  MY_RESULT = READ(456) + 0.42*READ(621)
END SUB
```

We need to DIM the loop variable MY_RESULT as follows at the top of the program.

```
DIM MY_RESULT AS SINGLE
```

Step 6 Reporting

Because it is so simple we will not compile it now but rather we will proceed and do the reporting section. Move to the DO_OUTPUT definition and add the following:-

```
SUB DO_OUTPUT
  LPRINT(“The result is “,MY_RESULT)
END SUB
```

Step 7 Looping

Now the program is ready to run for a single sample. To make it useful we need to use a loop to repeat the collect and process sections. To do this move to the THE_START section and add the loop lines until it looks like this:-

```
SUB THE_START
  GET_INPUT
  For i=MY_SAMPLE to SAMPLE_LIMIT
    Do_Process
    Do_Output
  next MY_SAMPLE
END SUB
```

We need to DIM the loop variable MY_SAMPLE as follows at the top of the program.

```
DIM MY_SAMPLE AS INTEGER
```

Testing the program

Because we may not have immediate access to the instrument system it is sometimes very useful to put in a simulated collection step to allow debugging of the program.

Step 9 Simulation

Edit the DO_PROCESS definition as follows:-

```
SUB DO_PROCESS
  ‘ MY_RESULT = READ(456) + 0.42*READ(621)
  MY_RESULT=CINPUT("ENTER SAMPLE RESULT")
END SUB
```

The program should now work sufficiently well to allow you to run it and produce some simple results output.

Step 10

Press the F5 key or the run button to run the program. If there are any errors reported, note the error description and line number. Re-enter the editor, fix the problem and then re-run the program. (Class exercise until working program)

Step 11 Adding Graphics

We will now add graphics output to the program before adding more complex input and processing routines. To add graphics we will use the PLOTSYMBOL command to place a symbol on the graph display to represent the Absorbance reading just obtained. We also need a graph. Modify the DO_OUTPUT section as follows:-

```
SUB DO_OUTPUT
  NEWGRAPH(“My Graph”)
  GRAPHSCALES(“My Graph”,0,0,10,1)
  PLOTSYMBOL(“My Graph”MY_SAMPLE,MY_RESULT,”X”,14)
  LPRINT("SAMPLE ",MY_SAMPLE," ABS = ",MY_RESULT)
```

```
END SUB
```

This will place a yellow diamond symbol on the graph. Run the program and watch the results.

Step 12 Saving the data

Because we will want to keep the data and archive we will now include the required commands to save the data into a continuum. Change the GET_INPUT and DO_PROCESS definition as follows:-

```
SUB GET_INPUT
  STARTPRINT          ‘ CLEARS THE REPORT
  BATCH$=CINPUT("PLEASE ENTER BATCH DESCRIPTION ")
  LPRINT("BATCH : ",BATCH$)
  USERNAME$=CINPUT("PLEASE ENTER YOUR NAME ")
  LPRINT("OPERATOR : ",USERNAME$)
  ABS_LIMIT = CINPUT("REPORT RESULTS FOR ABS ABOVE WHAT VALUE ")
  SAMPLE_LIMIT = CINPUT("ENTER THE NUMBER OF SAMPLES ")
  SETVAL(“Y Mode”,0)        ‘ ABSORBANCE MODE
  ‘ Note that we can run in any X mode as we will be creating our own data and plotting it accordingly
  ‘ We need to make a continuum
  NEWCTM(“My Results”)
END SUB
```

And to prepare to archive the data add the following to the THE_START definition.

```
SUB THE_START
  GET_INPUT
  FOR MY_SAMPLE=1 TO SAMPLE_LIMIT
    DO_PROCESS
    DO_OUTPUT
  NEXT MY_SAMPLE
END SUB
```

We now have the solid basis for our program. If you run this then you should see the results appear on the GRAPHICS window and as well on the REPORTS window.

Step 13 Scanning the samples.

Now that we have the simple program running we will enhance it by performing wavelength scans on each sample and getting some measure from the scan as our result for output. Lets take the first peak maximum we find within the scan range as the result. To do this collection we will be switching Abscissa modes from Sample Number to Wavelength and back again during the program cycle. We will assume that your user has set the peak mode to MAX and the threshold appropriate to the type of solutions being scanned.
This is type of operation done by the ADL program SCANPEAK.ADL that is provided with the release disk.

Modifications are required to the DO_PROCESS routine to switch modes, scan and find the peak. A modification is required to the GET_INPUT routine to ask for a wavelength range to scan the sample over. Change the routines as follows:-

```
SUB GET_INPUT
  STARTPRINT          ‘ CLEARS THE REPORT
  BATCH$=CINPUT("PLEASE ENTER BATCH DESCRIPTION ")
  LPRINT("BATCH : ",BATCH$)
  USERNAME$=CINPUT("PLEASE ENTER YOUR NAME ")
  LPRINT("OPERATOR : ",USERNAME$)
  ABS_LIMIT = CINPUT("REPORT RESULTS FOR ABS ABOVE WHAT VALUE ")
  SAMPLE_LIMIT = CINPUT("ENTER THE NUMBER OF SAMPLES ")
  SETVAL(“Y Mode”,0)        ‘ ABSORBANCE MODE
```

‘ Note that we can run in any X mode as we will be creating our own data and plotting it accordingly
NEWCTM(“My Results”)
Add these lines ->

```
SCAN_UPPER = INPUT ("ENTER THE SCAN START WL ")
SCAN_LOWER = INPUT ("ENTER THE SCAN STOP WL ")
SETVAL(“X MODE”,WAVELENGTH_XMODE)
END SUB
```

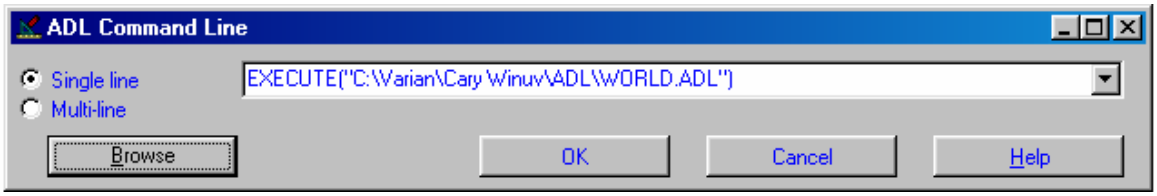
For the processing we will again simulate the operations but now we enter the real code.

```
SUB  DO_PROCESS
Remove this line ->MY_RESULT = READ(456) + 0.42*READ(621)
COLLECTANDGRAPH(“My Scan Data”,”My Scan Graph”,SCAN_UPPER, SCAN_LOWER,
                    WAVELENGTH_XMODE)
A!=PEAK(“My Scan Data”,0,1)       ‘FIND PEAKS
A=PEAK(“My Scan Data”,1,1)       ‘FIRST MAX
Up to this line -> MY_RESULT = PEAKY
‘MY_RESULT=INPUT ("ENTER SAMPLE RESULT")
 ADDCTMDATA(“My Results”,MY_SAMPLE,MY_RESULT)
END SUB
```

How to run your program

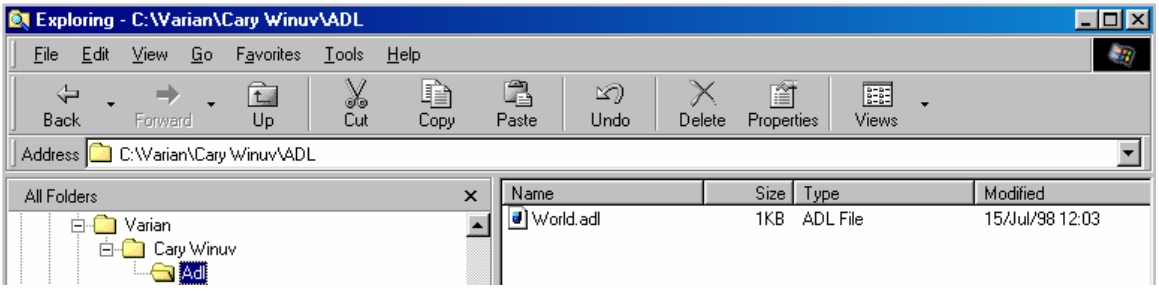
EXECUTE

The EXECUTE command can be seen when we select the BROWSE button and choose an ADL file from the Alt-Y dialog. This wraps the ADL file name inside the EXECUTE command. Any ADL file can be run by this method.



Where are ADL files normally located?

ADL files by default live in the ADL folder immediately “under” the folder you installed Cary applications into.



Do I need the full file path and extent?

If you do not explicitly provide the full file path to the EXECUTE command then the default ADL sub-folder is used.

Drag/Drop

ADL programs can be run by the drag and drop method. That is, you can press the mouse button down on them in the Windows Explorer, and while holding the mouse down, drag them onto either a running application or onto the application EXE name in a folder and then release the mouse. This is what is meant by drag and drop.

File Association - the desktop icon

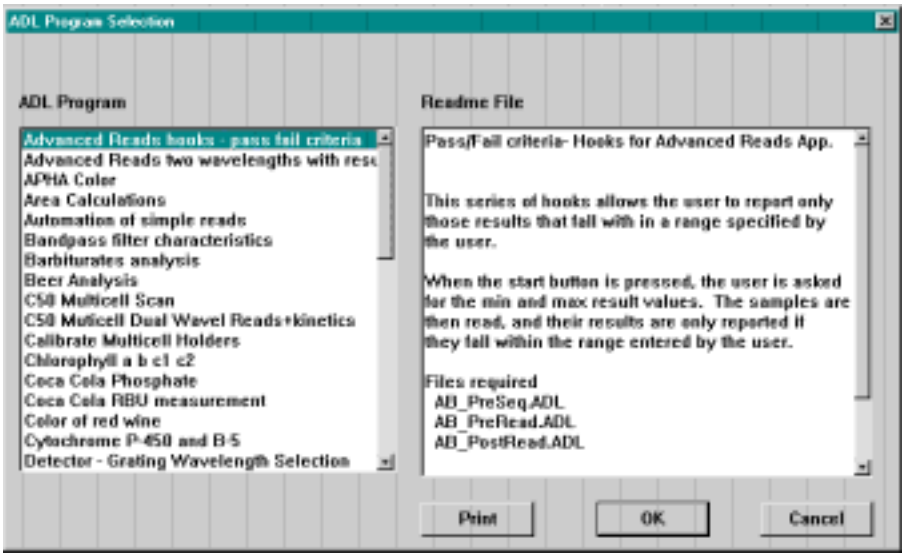
By installing the Cary software we create a link for the operating system to know that when you double-click on a file that has the '.ADL' extent that the operating system must run the ADLShell program and automatically load the ADL file mentioned. This is called FILE ASSOCIATION. This allows you to create simple ICON analyzers from your ADL programs. It's that easy. If however you want to change the installed default association from the ADLShell to (say) the SCAN program then you must modify the association link using the VIEW .. OPTIONS menu control of the operating system.

Note that ADLShell is minimized to the desktop by default when running an ADL file from an Icon. A command SetWindowState is provided for those ADL programs that would like to show the ADLShell program in full screen, minimized or normal modes.

ADL programs can be run by all the following methods -

1. Use the Alt-Y dialog to select an ADL file (Browse) then press OK
2. Open the ADL file from the file menu
3. Double click an ADL file (this runs the ADL program from within ADLShell.exe)
4. Drag and drop an ADL file onto a Cary executable program
5. Set the "Parameters" field in the desktop icon shortcut to the appropriate ADL file
6. Load the program into the ADL editor and press RUN
7. The ADL Program Selector

The last method is our preferred technique for running ADLs. The ADL Program Selector provides a tidy way to look at what ADL programs have been installed and give the user some guidance about their use. The selector is available from a desktop shortcut if any of our ADL solutions (from our web page) have been installed.



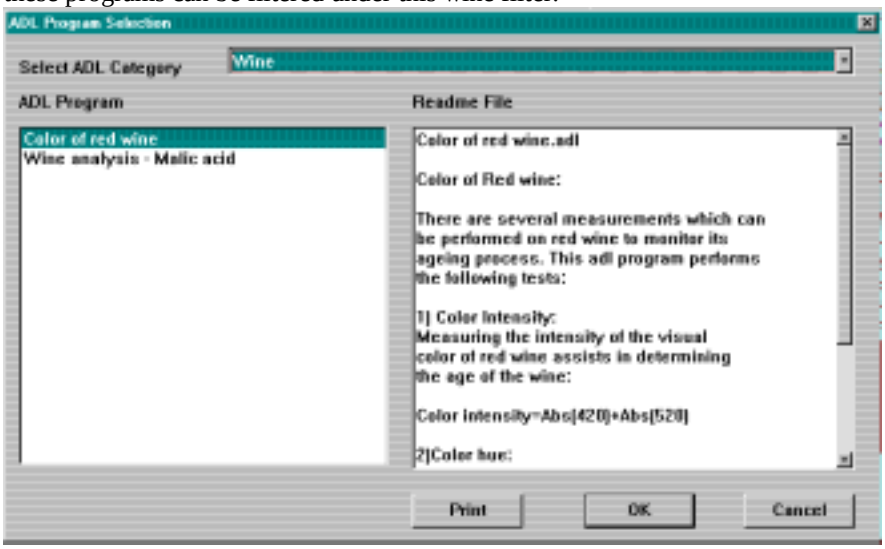
How to Install an ADL into the ADL Program Selector

There are a few simple steps required to make an ADL appear in the ADL Program Selector. These are-

1. Make a folder under the existing installed ADL folder with a suitable name to describe the ADL.
2. Place the ADL program(s) into that folder
3. Name the ADL tha you want to run with the same name as the folder, plus the ".ADL" extension
4. Make a readme.txt file with information that your user will see.

If you are making an ADL HOOK suite of ADLs then Steps 3 and 4 are slightly different. In that case there is no need to rename any ADL programs. Also the Readme.txt becomes Readme hook.txt.

A filtering mechanism on the ADL Program Selector allows us to group programs using new a common file name placed into each folder to be linked. The name of the file must end with the extent ".PAK". The name will appear in a filter list in the ADL Program Selector. For example to link all the wine analysis programs We could make a file (containing nothing) called Wine.PAK. Place a copy of this file in each folder that has a wine analysis ADL solution. Then when we run the selector program we see all these programs can be filtered under this wine filter.



Multiple filters can be installed. As well each folder can have several ".PAK" files to filter it under several groupings.

Continuum mathematics

Because of the restrictions imposed by SAX Basic upon creation of new data types, we could not use the same types of simple operation styles that we normally used in DOS and OS/2. For example in DOS we could add two continua together as follows

A#=B#+C#

We can still do the same operations but we needed to make a special continuum handler to allow us to do this. We use the CTMOP continuum function to do our math. For example with adding two continua together we do

NewCtm$ = CTMOP("Ctm1","+","Ctm2")

We must place the operations in quotes to get the same result as in DOS or OS/2. Numbers need not be in quotes. We can chain these together to make complex equations. Allowable operators are "+ - * /" for combining continua.
To proceed please run the MakeCtms.ADL program. This will give you some simple data to work with.

NewCtm$ = CTMOP("Ctm1","+","Ctm2","/",2.5)

This will evaluate the expression from left to right to give the result of

$$\frac{"Ctm1" + Ctm2"}{2.5}$$

NewCtm$ = CTMOP("Ctm1","+","Ctm2","/",2.5,"-",10)

Will give

$$\frac{"Ctm1" + Ctm2"}{2.5} - 10$$

For operations on one continua we can do the following operations "Log10, AntiLog, Ln, Exp, Sqr, Sqrt"

NewCtm$ = CTMOP("Log10","Ctm1","+","Ctm2","/",2.5)

Will give

$$\frac{Log10("Ctm1") + Ctm2"}{2.5}$$

Chaining is allowable to make more complex equations such as

NewCtm$ = CTMOP("Ctm1","+",CtmOp("Ctm2","/",2.5),"-",10)

Will give

```
"Ctm1"   +    "Ctm2"       - 10
              ------
               2.5
```

The middle CTMOP function will evaluate the expression Ctm2/2.5 as one continuum. For derivative math try the following

PLOT(Deriv(“Ctm3”,1))

It's time for some more experiments. Try to make some example programs.
Where you may need to just work with data files, use the File…Open dialog, and take the trace name as displayed on the legend/trace preferences etc. as the continuum to work with. If this is not suitable for some reason then you can construct sample data files as follows.

```
DIM X As Single
DIM Y As Single
A$=NewCtm("SampleData")
ClearCtm("SampleData")
X=0
For I = 1 to 100
  Y=I/100          ’ some fake Y data
  AddCtmdata(A$,X,Y)
  X = X + I/100      ’ move on a bit
Next
```

Prac session: Simple example programs

Hello World example
Make a simple ADL that puts up a simple dialog box containing the message "Hello World" plus an OK button, then add a Yes/No button set.

Simple Reading example
Make a simple ADL that prompts the user for the following values
Wavelength
Averaging time
Number of samples
The program will then setup the instrument with the values entered.
Then take readings and prints a simple report with the following form.
"Result is : xxx.xxx"

Cell changer
Perform a simple multi-cell collect in Abs mode. Provide a prompted means to enter the start and stop times in minutes. Provide a prompted means to select how many cells the user wants to use (1 to 6 only). Provide graphics scaling as entered by the user.

The ADLConverter program

In order to make the migration from DOS and OS/2 ADL programs we have created an ADL conversion program. This program (ADLCONVERT.EXE) will do its best to change your existing programs where it can and indicate where you need to make further changes to get them to run. A number of existing ADL programs will run with only a few automatic changes. Some will require greater changes, especially where graphics has been used. It is worth noting that often the best way to convert a program is to re-write it rather than translate it. That way it is easier to produce a clean version, using the latest commands that are available.

Lets examine a few simple programs to see how it works and to see its pitfalls. Examples are DOS ADLNEWS14 (easy to translate) and DOS ADLNEWS5 (some problems).
Note that the converter can be run as a “windowed” program OR you can run it as a command line program with multiple parameters. Each parameter can be the name of a DOS or OS/2 ADL program to be converted. In this command line mode the converted programs are automatically renamed with the prefix “WINADL” attached to the original name. This way you still have the original program untouched and a new converted program with a very similar name.

Combined operations for your ease of use

We have added several new commands for your convenience.

CreateCtmAndGraph
This command allows you to make a continuum and a graph with default scales and appropriate modes set. Try this command with the following parameters

CreateCtmAndGraph("Wavelength Data","Scan Graph",600,500,Wavelength_Xmode)

Or

CreateCtmAndGraph("Time Data","Time Graph",0,0.5,Minutes_Xmode)

Collect
This command will start a collection and put the data into a pre-existing continuum. For example we can make a continuum and then collect in the current mode with the default start and stop as follows.

A$=NewCtm("Scan Trace")
Collect(A$)

Note that this will not display as it collects. To do the combined operations of making a graph and a continuum plus collecting and displaying we have the following.

CollectAndGraph is a special ADL command that allows you to specify a graph and a continuum and some simple SCAN parameters. The command will do all that is necessary for you to make the graph and continuum, setup correct labels and default scales and complete the collection, showing the result as it is collected. Try the following

CollectAndGraph("Wavelength Data","Scan Graph",600,500,Wavelength_Xmode)

Several old commands still exist to allow you to make reports and to save your data.

For reports you still have the old commands STARTPRINT to clear a report and the trio of TableFmt , TableHdr and TableData to make presentable tables. For example we can make a simple table as follows.

TABLEFMT("#.###","###.#",6A")
TABLEHDR("Result","Wavelength","Units")
TABLEDATA(1.2, 456, "g/l")

Or

TABLEFMT ("###.###","000","#0.00E+00")
TABLEHDR ("One big line","two","third column")
TABLEDATA (1.2,2%,45.7)
TABLEDATA (4.567,5%,125.67)

You may note that the formatting commands have changed. If you refer to appendix C you can see the extent of the available formatting controls.

Also as previously mentioned we still have the TABLE command. This command allows you to make an XY pairs data table from a data file. With this command we can specify the range and interval of data points to put into the report. For example to make a table from a wavelength scan, reporting data every 5 nm in an XY table we can do the following.

Table("My Data",500,400,5,0)

The last parameter value 0 produces an XY table, value 1 produces a peak table.

Storing data is as simple as it ever was. We still have the ARCHIVE command as follows.

ARCHIVE("My Data File.DKN")

This saves the selected trace to the given file. It always saves data files for "backwards compatibility". We can also use the new Cary SaveAs command to allow us to specify any file type to save as. For example

SAVEAS("File Name",Data_Files)
OPENAS("File Name", Method_Files)

As well, we have also seen the way to bring up a SaveAs dialog with the Show…Dlg commands. E.g.

ShowSaveDlg.

Also the old command ArchiveReport is still available for those who just want to save the report file.

Prac session: Simple example programs

The class will construct the following examples:

Scanning
Perform a simple wavelength collect in %T mode. Provide a means to enter the start and stop times in minutes. Provide graphics scaling as entered by the user. Provide the ability to perform a baseline. Provide repeat cycle operations.

Time collect
Perform a simple time collect in Abs mode. Provide a means to enter the start and stop times in minutes. Provide graphics scaling as entered by the user.

A Stirene analysis example
Translate the program Stirene.adl

Beer analysis example
Translate ADLNews21 - Alpha and Beta Acids in Hop powders and pellets

If time permits try the following ADL programs.
RBU.adl
ADLNews14.adl
ADLNews16.adl

ADVANCED ADL

Advanced Variable Declaration

For the variable types that we looked at previously there is also a shorthand way to tell the computer that we are using a specific type of Variable. For example when we wished to declare a SINGLE variable we used

DIM MyVariable As Single

The shorthand way to do this for SINGLE variables is with the '!' character. For example

DIM MyVariable!

Is equivalent.

Integers (2^16) use the '%' character such as

DIM MyIntegerVariable%

Strings (as we have seen before) use the '$' character

DIM MyStringVariable$

Booleans use the integer shorthand %

DIM SaveTrace%

Double precision reals (type Double) use the # character

DIM PreciseNumber#

Long integers (2^^32) use the & character

DIM LongerNumber&

Currency variables use the @ character

DIM DollarValue@

Also this convention can be used for FUNCTIONs. For example

FUNCTION FactoredRead(Wavelength!, Factor!)!

END FUNCTION

Arrays of variables that we previously declared with the DIM statement have a strange "computerish" twist. When we made an array variable such as

DIM Box(10)

We are actually making 11 items, not 10. The software recognises Box(0) as a valid entry. For most work you can effectively ignore item zero.

Dynamic arrays in ADL are also possible. By this we mean that sometimes we don't know how many items we want in an array until we run the program. ADL allows us to define an array variable and determine its required size later. This can be useful in saving memory and as well in knowing how many items are in the array. Before examining this we will look at two array items LBOUND and UBOUND. These indicate the index of the first and last item in the array respectively.

CPRINT(Lbound(Box),"->Ubound(Box))

Will print 0 -> 10

Indicating the range allowed.

To declare a dynamic array we leave out the number of items in the DIM statement.

DIM Box()

To size it we can use the Redim command

Redim Box(3)

Would make the array upper limit now 3.

Cprint(Ubound(Box))

Prints the upper limit as 3.

Redim Box(5)
Cprint(Ubound(Box))

Prints the upper limit as 5.
There is more to be discovered with dynamic arrays but that's best left for examples in the help. For now just know that you can size them at run time rather than be required to know ahead of time how many items in the array to create.

Up until now we have made Windows ADL look similar to DOS ADL where we did not have to declare certain variable types. Instead we could just use them in the programs and they were effectively Automatically DIMed. Unfortunately when we do this we can get into problems with the variables not having correctly been initialized. For example if I wrote the following small program.

Cprint(SlitWidth)

The computer says the value **is not initialized**, meaning it has **no value** nor a variable type. Programs having unknown variable types can have serious operational problems.

I could use the Dim statement to make its type known and give it a value (zero is the default value for this type).

I could also avoid having this type of problem in my programs by issuing a command to ADL to make it insist that all variables are identified explicitly. This is with **Option Explicit.**

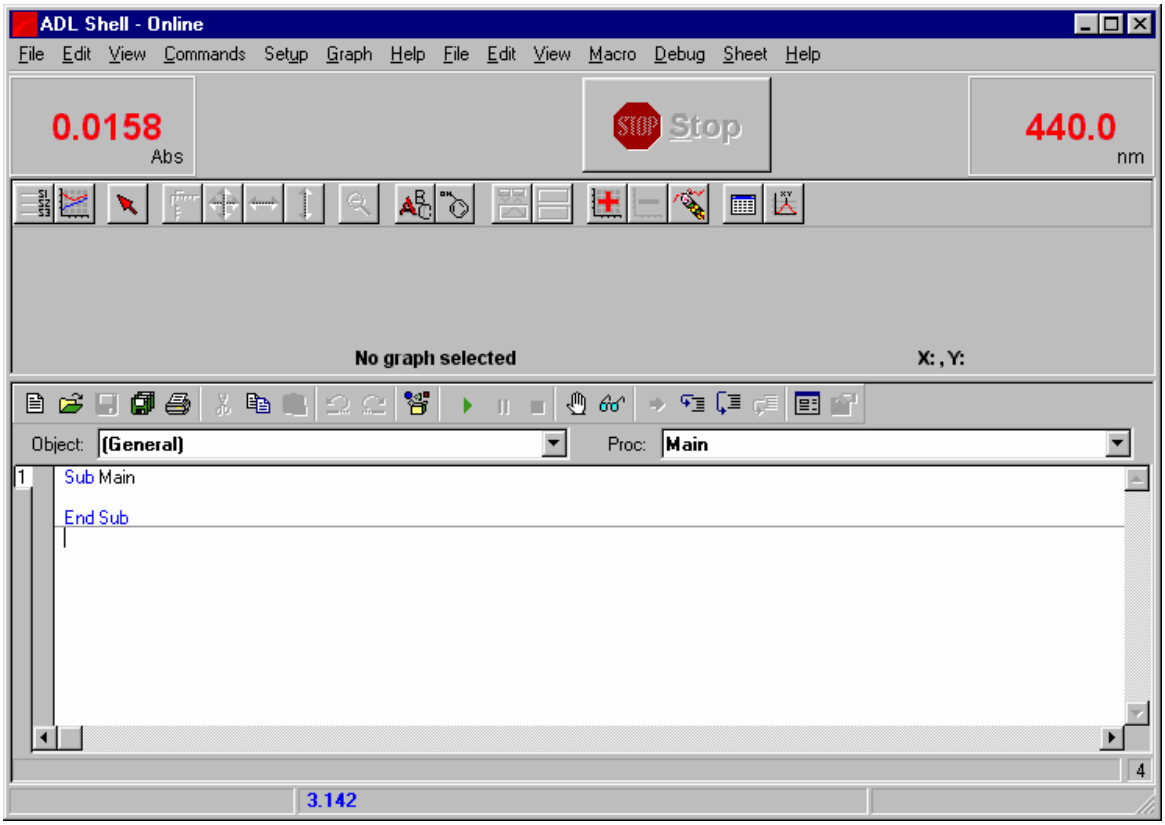
Option Explicit
Dim SlitWidth As Single
Cprint(SlitWidth)

If I failed to DIM all variables with the Option Explicit directive in my program then the program would not run and would indicate where the offence occurred. In this way we cannot make mistakes where the type is not identified. Note however that I still have not initialized the variable SlitWidth.

Using the editor

Load the ADL editor as before.

Notice there are some new Speed Buttons in the ADL window. These allow opening and saving of files, running your programs, examining what happens while you run your program and a Dialog editor



The editor has the usual Find dialog, just press control-F to bring it up.

Configuring the ADL editor
Move the mouse to the ADL area and press the right mouse.
Notice that there is a popup menu with several entries. This is an extensive list of the capabilities of the editing environment. The Speed buttons represent a set of these.
Use the View option of the popup to change the appearance of the editor.

The major part to notice here is the area with

Sub Main
End Sub

This is the section we will work with. As we have already seen this is the part of the program that ADL will run first before all other sections of SUBs and FUNCTIONs.

Syntax coloring as a guide to understanding

Look at the basic layout of the ADL editor. You will see

Sub Main
End Sub

Notice the words Sub and End Sub are in blue. The text Main is in black.
The font can be changed under the View…Font menu item.

Enter a simple command between the Sub and End Sub commands such as

MSGBOX "Hello World"

Notice the color of various words. The keyword MSGBOX changes from black to teal and also assumes different text case. As you type the word in the line is black. As you move off that line the color changes to indicate the computers understanding of the words. Try entering some rubbish on a new line and see the effect. Notice that the words do not change color. The computer did not understand your rubbish. For example

MsgBoxx "Hello"

Now put in a comment so that you program looks like this

Again the color of the text is different. Comments appear in green. Words that the editor knows appear in color. Later on you will see that errors are also color-coded. This Syntax coloring is a good aid to help you with your programs.

Lets run this simple program and see what happens?
To run the program either click on the tool button with the VCR like arrow pointing to the right OR you can press the F5 key. Try it.

If all went well you should have seen the dialog appear with your message.

Saving your file
Try saving this program to disk. Either with the ADL file button or the popup menu file option.

Opening an ADL file
Select the File Open menu or press the Open speed button. Now lets find the program labeled CaryEquation.ADL. You will find it in the ADL folder. Open this file and examine its contents in the ADL editor. This program is the last ADL program that we typed in using the Alt-Y command line ADL operations.

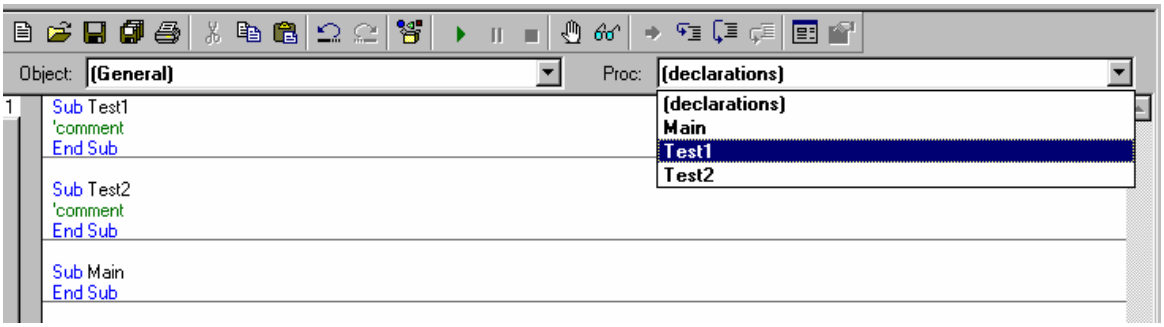
Multiple SUB and FUNCTION blocks
You can add as many SUBs and FUNCTIONs as you like to the sheet. Each one will be visibly separated on the screen by a line inserted automatically. For example enter a new SUB as follows

Sub Test1
'comment
End Sub

Sub Test2
'comment
End Sub

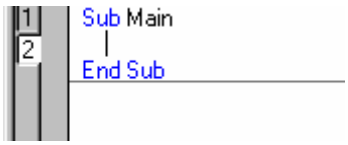
Sub Main
End Sub

When you do this you can quickly move between the various SUBs by using the PROCEDURES "drop down list" in the editor. In this trivial example you will see the three SUBs listed.



Multiple sheets

The ADL editor is not restricted to just one program open at a time. You can open several programs as "Sheets" to take a term from Excel. Each sheet contains its own program that can be run independently or they can be linked.
Create a new sheet with the NEW speed button or file item.
See that there are now numbers down the left side of the ADL editor. Click on any one of these and we switch between the sheets.



You will also note that the opened file from the previous File Open has been loaded into a new sheet. Your existing "Hello World" program is still there in another sheet.

The advantage of having multiple sheets is that you may be debugging a program (for example) that uses an accessory. While that program is working you ca open a new sheet, enter some other commands and run that program, then when the new program completes you can return to your first program.

If you find there are too many sheets to keep track of then you can close some or all from the Sheet popup menu. The maximum open at any one time is 9 sheets.

Stopping a program

There is a Stop button, which looks like a small red square just near the Start button. This will stop execution of any running ADL programs. Also whenever an ADL program is running, the main Cary application STOP BUTTON will be active. This will also allow you to stop executing your ADL program (provided that you program allows buttons to be pressed - see the DoEvents usage in the "What's new - more advanced" section).

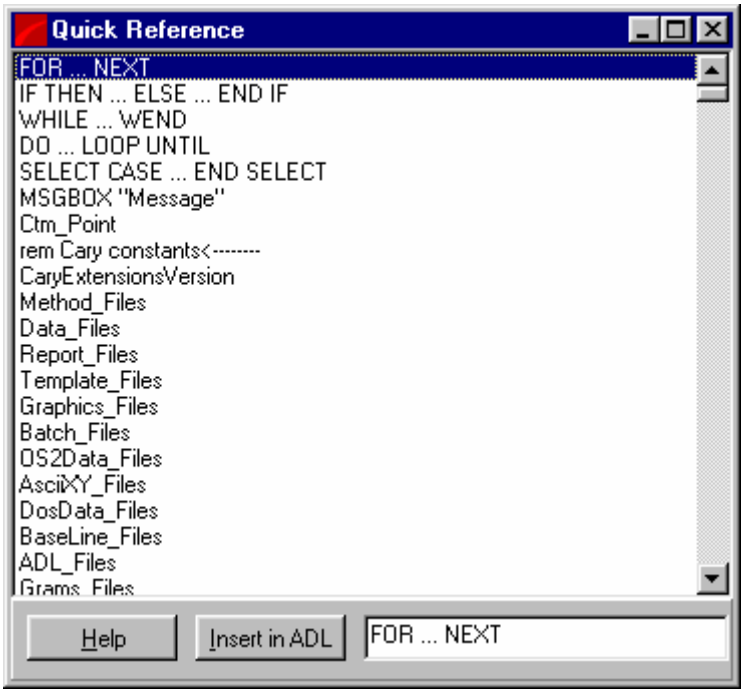
Note that WinADL user dialogs are MODAL. That is they must be dismissed before you can interact with the rest of the system buttons menus and other items.

Help

WinADL has context sensitive help. You get this by positioning the cursor on a word and pressing Shift-F1, or you can use the menu and select Topic Search. At present the full ADL manual is unavailable. You must do the hyper-linking by selecting the item from the help file of the browser.
We have initially also included the SAX basic command set in the HTML help format and as well it is in the ADL folder as a windows ".HLP" file.

Alt-Y quick reference mode

This guide that was available from the Alt-Y dialog is also useful for entry into the ADL editor. To bring it up, simply select it from the Cary VIEW menu (not the ADL popup View menu). As before you can use it to see or even directly transfer values into the editor at the currently selected cursor position. So just place the cursor where you want the command/information to go and then select and transfer your choice to the ADL editor. Here's a quick tip. If you want to replace something in the editor then you first highlight the entry in the ADL editor. This will cause it to be replaced when you transfer the information over.
Items in the quick reference list are ordered by function groups.



Finding Errors in the Editor

When running programs from within the editor and an error occurs the offending line will be highlighted in red and the cursor is placed at the location that the computer had difficulty with. This is the best indication you will get about the problem. It may be as simple as non-matching brackets (you must have the same number opening and closing) or it may be missing double quotes to start or end a STRING field.

Uses clause

When you want to have a series of program routines that you want to write once but use in several different programs then we use the USES clause. The syntax is

'#USES "FileName"

Place this in the Declaration section of your new program.
Your program can now reference routines within the filename specified in the USES clause.
Note that the file referenced by the USES clause has no SUB MAIN. Here is a simple example
In sheet 1 we have

```
'#Uses "B.WWB"
Sub Main
   CPrint BFunc("Hello") 'this will print "HELLO" on the status line
```

End Sub

And we also have a saved program module (B.WWB in this case) with the following lines. Note that the name need not have ".ADL" in it.

```
Public Function BFunc(S As String) As String
   BFunc = UCase(S)
End Sub
```

The reference to “Public “ allows items in the module to be referenced from outside its world. If they were untitled or explicitly referenced as “Private” then they cannot be seen outside the module. This latter ability allows us to have the same names for functions and subs inside different modules but not be confused about names outside the modules.

As previously mentioned we can also put commonly needed SUBs and FUNCTIONS and variables in the file ADLUserExtenstions.TXT.

Classes, Objects and OLE

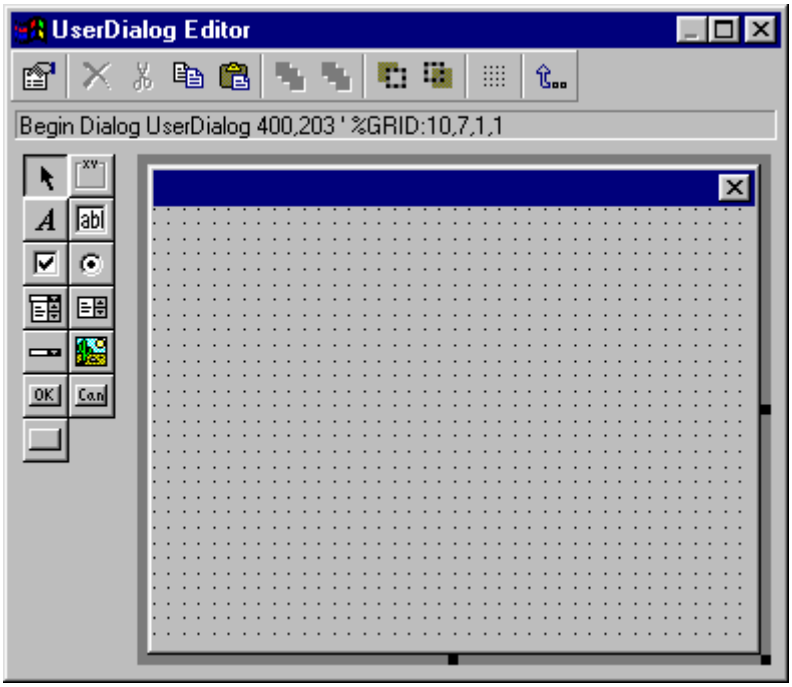
This is an area that we will leave to the software experts to explore. I suggest that unless you understand these terms then don't proceed any further. If you do then refer to the help as to their uses or see Appendix J. You will find them quite advanced.

Dialogs – the simple way

In the ADL editor there is a very simple to use dialog editor. Follow as we make a simple dialog.

First select a new sheet.
Place the cursor on the line between the SUB MAIN and the END SUB command.
Then select the dialog editor either from the speed button or the popup menu.

What you see is probably the greatest leap forward in ADL over OS/2 and DOS.
From here you can create your own dialogs that contain buttons, radio controls, check boxes, text, entry fields, a picture, list boxes etc. There are more than enough controls for you to use.



Lets take a simple tour and make a simple dialog. Click and drag the OK button onto the design form.

To place this dialog that you have drawn into you ADL program press the button that looks like an up-arrow with a bent tail (on the right at the top of the dialog editor form). You should observe that the SUB MAIN is now filled with lots of words with DIALOG or similar to the following.

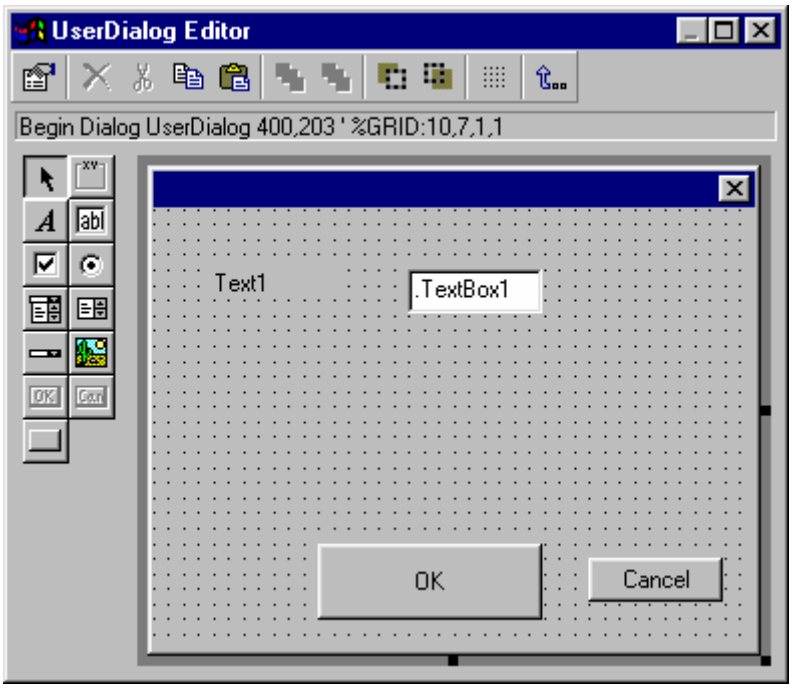
```
Sub Main
        Begin Dialog UserDialog 400,203 ' %GRID:10,7,1,1
                OKButton 110,154,150,35
        End Dialog
        Dim dlg As UserDialog
        Dialog dlg

End Sub
```

Now RUN the program. Your dialog appears instantly. Press the OK button or CANCEL button to close the dialog. That is the basic dialog.

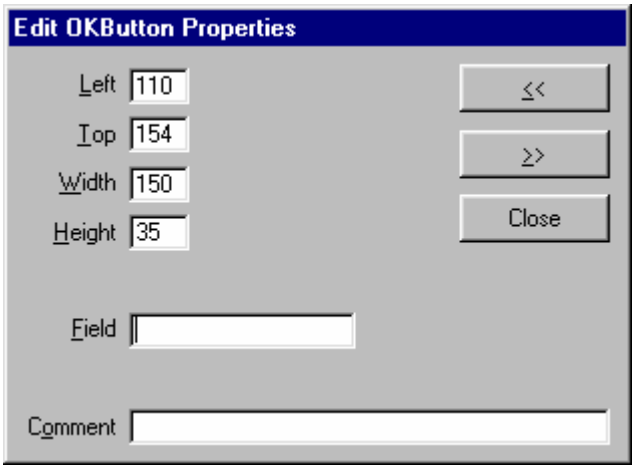
Let's explore this more. Examine the text that was entered. It defines the dialog size, the type and location of the controls you used and some extra code that actually creates and displays the dialog.

To get back to the dialog editor with this dialog you simply place the cursor on the dialog code section and press the dialog editor button or menu selection. You should then see your dialog as it was. Let's expand this some more.

Add a Cancel button
Now put a text field on the form (use the "A" button)
Now put an entry field on the form (use the "ab" button)



Run this example. See how it is growing. You can resize any of these items at design time. By selecting an item with the right mouse in the dialog editor section you will see a popup "properties" window. Here you can change the captions and change the given names for the dialog components. These names are important for us to place information on the dialog prior to showing it and to allow us to get information back from the dialog after the user has closed it.



Let's change the text captions on the items we placed on the dialog.
Change the text field caption to read something like "Wavelength (nm).
Change the entry field name to **Wavelength**.

Run the program and see the effect. Notice that the wavelength Entry Field does NOT have any text in it. Let's rectify that.

Find the code that looks like this

Dim dlg As UserDialog
Dialog dlg

To set the wavelength field we need to access that item on the dialog. The way the software understands this is by a parent to child relationship. This is where the dialog (in this case the variable dlg) and the entry field (named wavelength) are linked with the dot "." character. E.g. Dlg.Wavelength now is the item that we can change and read to pass information to anf from the dialog. Add the following line to it so we have the following.

```
Dim dlg As UserDialog
dlg.wavelength = "500.0"
Dialog dlg
```

Run this program now and we have the value 500.0 appearing in the dialog for our "default" wavelength value.

Now for the last stage of interacting with the user. We will get information back from their entry. This is in two steps. First we must find out if they pressed the OK button. This is as easy as follows.

Change the line above from

```
Dialog dlg
```

To the new line

```
Pushed = Dialog(dlg)
```

And then after that complete the code as follows.

```
If Pushed = -1 Then
  Msgbox "we pushed the OK button"
End If
```

Your changes should look like the following

```
Dim dlg As UserDialog
dlg.wavelength = "500.0"
Pushed = Dialog(dlg)
If Pushed = -1 Then
  Msgbox "we pushed the OK button"
End If
```

Try running this and press OK or Cancel.

What we have done here is ask the dialog what key was pressed. If the user presses OK then we get back the value "-1". If the user presses Cancel we get back "0". If we add more buttons then each button returns "x" where x is the number of the button added. For example we add two more buttons. Regardless of their names, when pressed they return value "1" or "2" for the first button added, and the second button added respectively.

The final stage is to examine the entry the user returns when the OK is pressed. This is as simple as it was to put the value in prior to showing the dialog.

We simply replace the line
```
Msgbox "we pushed the OK button"
```
With
```
MsgBox "we pushed the OK button and the wavelength is "+dlg.Wavelength
```

Save this as Entry.adl

NOTE: Dialogs are MODAL. That means that when they are displayed you cannot access other parts of the program. In order to work within the dialogs we need to explore Dialog Functions. We will just take a quick tour here through some examples rather than enter any code. Lets select and run each of the following program examples in the ADL folder.

Button.adl
ListBox.adl
Radio.adl
DropList.adl
Combo.adl

Note we can also have dialogs within dialogs. For more advanced Dialog interaction it is relatively easy to alter the Dialog characteristics when the program is running. I suggest that you look at the DialogFunc section of the ADL help for examples on this.

What's new - more advanced

Handling Input

InputBox

Sax Basic has an InputBox statement that we have used to make the Cinput statement. The syntax of the statement is
InputBox[$](Prompt$[, Title$][, Default$][, XPos, YPos])

The bits in the square brackets mean that they are optional.

For example you could query for input in the center of the screen as follows

A$=InputBox("Enter something", "Entry Title", "Default Value")

Or we could place the box at the top, left of the screen using the optional x,y position

A$=InputBox("Enter something", "Entry Title", "Default Value",1,1)

With the InputBox command, you always get a string value back. If the user cancels you get back an empty string, otherwise you get the entered value.
The Cinput and Vinput functions are easier to use and we recommend that you use them in preference.

Flow Control

When you are writing loops its very easy to write a loop that appears to lock up the computer. I have lots of experience at this. In order to prevent this you need to use a special command DOEVENTS that allows the rest of the things in the computer to "have a go". For example the following program will run "forever".

```
A=0
UserB=400
WHILE A=0
  UserB=UserB+1
  C=Read(UserB)
  CPRINT(C)
WEND
```

Although quite elegant, it is useless. With the addition of DOEVENTS it can at least be stopped.

```
A=0
UserB=400
WHILE A=0
  UserB=UserB+1
  C=Read(UserB)
  CPRINT(C)
  DoEvents
WEND
```

This is much cleaner (but still not useful).

Variants - what are they?

Variants are items that do not have a specific type until they are actually used. Then they take the type from the item that they are set by. For example if we declare a variable MyVar as such

DIM MyVar

Then it has no explicit type as different from the declaration. It is also “empty” until an assignment (MyVar=*something*) has been done.

DIM YourVar As String

Which has a type STRING.

We can test if the Variant is empty with the IsEmpty(MyVar) function.

Creating Formatted Output

There are many WinADL commands to make your output (report, screen, files) more tidy. For example you may have a number such as 1.2345678 and wish to have it formatted to just two decimal places. This number is to be printed in your report. With normal LPRINT of a number you get a default formatting which may not suit your report style. In order to format output you need to understand the process of variable type conversions.

Conversions

We have spoken about some conversions between different types of variables and information. We mentioned the VAL and STR$ conversions. Now here are some more conversions. We need to convert between different variable types because the computer is not smart enough. We need to tell it exactly what type of information that it is dealing with.

FORMAT$
This command allows you to make a String from any type, just by using the correct “transferal information” by way of Format Specifiers. (see appendix C). The syntax of the command is basically

FormattedString = Format$(Variable, "Format specifiers")

The most common ones to remember are number format specifiers "#" and "0". The "#" character will put in characters in place of the "#". The "0" specifier will always display characters even if not given.

For example if we want to format a number to always display two decimal places we can do the following.

```
B=Vinput("Enter a number")
A$=Format$(B,"#.00")
Cprint(A$)
```

The "00" part always displays two characters even if we enter a whole number. The "#" will allow us to not display numbers unless they are entered.

If we always want to display time to two characters (e.g. seconds display) we use the following

```
A$=Format$(TheTimeValue,"00")
Cprint(A$)
```

There are many more examples of formatting in Appendix C. Take time to examine these as they are very useful for making tidy reports.

Scope of variables

Whenever you create an ADL program you will invariably want to use your own variables for storing and manipulating information. The computer can "see" these variables in either of two ways. They can be understood at every point of the program OR they can be understood within some local context only. The term for this is the SCOPE of the variables. The former case is called **global** scope and the latter is called **local** scope.
To differentiate between these two scopes the computer looks at where (if at all) you have DIMed your variables. If the DIM statement for a variable is outside your SUBs then it is defined as global. E.g.

```
DIM MyVar As String

Sub Main
  MyVar=Cinput("Please enter your name")
  Lprint(MyVar)
End Sub
```

This shows global scope.

If we put the DIM statement inside a SUB (or omit it entirely) then it shows local scope as in the following example.

```
Sub Main
DIM MyVar As String
  MyVar=Cinput("Please enter your name")
  Lprint(MyVar)
End Sub
```

This is important if we have 2 or more SUBs in our program. For example

```
DIM MyVar As String

Sub Test
  MyVar="Hello"
End Sub

Sub Main
  MyVar = "Start"
  Test
  Lprint(MyVar)          ' prints the value assigned in Test -> "Hello"
End Sub
```

In this example the variable MyVar is known and can be seen or changed from any point I the program as the DIM statement is outside the SUBs. It is changed inside SUB Test and observed (Lprinted) inside Sub Main.

With local scope of variables the result of above is different.

```
Sub Test
  DIM MyVar As String
  MyVar="Hello"
End Sub
```

```
Sub Main
  DIM MyVar As String
  MyVar = "Start"
 Test
 Lprint(MyVar)          ' prints the local value only -> "Start", ignoring the
assignment in Test
End Sub
```

The result of having MyVar declared inside the SUBs is that the value of the variable in one SUB cannot affect the value in the other SUB. In this case the result prints differently. This mechanism allows you to either keep information known to all SUBs (and functions) or keep it local to one SUB and therefore not changeable by any other part of your program. Omission of DIM statements is the same as if they were DIMed locally. That is the value assigned inside a SUB is not seen outside that SUB.

NOTE: If your program appears to not keep the value that you assign to a variable, check that you have placed the DIM statement in the correct place (if at all). Global scope, placing all DIM statements at the top of the file, outside all SUBS is the simplest way to get ADL programs to work successfully. The down side of this is that if the variable is changed in more than one line of your program it can sometimes be a bit harder to debug.

With your programs that you want to share or reuse with other programs you previously linked them in with the USES clause. We need to note that the subroutines defined in the USES module can be **PRIVATE or PUBLIC.** You may remember seeing these words in some of the prior example programs.

When you want a declared variable in a sheet/module to be available for use by other sheets then you need to place the keyword PUBLIC in front of it. The same applies for SUBs and FUNCTIONs. For example you can say

```
Public Const MyADLVersion = 1.0

Private Sub TestCode
' the rest goes here
End Sub

Public Function FactoredResult As Single
' the rest goes here
End Function
```

In this example we have two things that can be seen by the ADL world and one that is only seen in the sheet that it is written. Why would you want to do this?

It is possible that you use the same name for a SUB in two different sheets that you are "USING" (with the USES clause) in your main program. This will cause confusion. Either you have to change the name of one, or you need to change the SCOPE of one by using the keyword PRIVATE.

Debugging

Complex programs rarely work correctly the first time that they are run. Therefore we need to DEBUG them to fix up the problems. The WinADL editor incorporates some advanced features that allow you to slowly step through your program and check that it is working correctly. These features come under the general heading of DEBUGGING tools. We will examine them more in this next section.

Printing out along the way - debug.print, lprint ...

One simple way to debug programs is to print out information as the program is executing. To do this we need to see it somewhere. We can use the CPRINT and LPRINT commands to record information. There is also a special DEBUG.PRINT command. This will write to what is labeled as the "Immediate" tab in the ADL editor window. Try a simple example

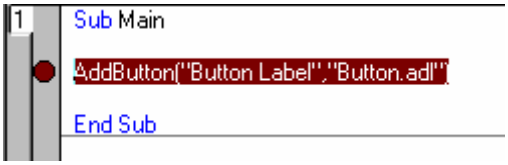
```
Sub Main
  A=Vinput("Enter a number")
  Debug.print A
End Sub
```

Stepping through the program

We can execute our program one line at a time if we want using the STEP debugging tools. These are under buttons and the DEBUG popup menu as "Step Into, Step Over and Step Out". The first one (Step Into) will execute each line of code in sequence, pausing after each line to allow you to decide what to do next. The Step Over will allow you to pass over SUBs and FUNCTIONs in one operation rather than go inside them. The Step Out will get you out of a SUB or FUNCTION quickly without having to execute all the lines of code.

What are breakpoints ?

We can also let the program run until it reaches a pre-determined spot. This spot is called a BREAKPOINT because the programs BREAKs or pauses at that spot. To set or remove a breakpoint use the F9 key or the DEBUG popup menu "Toggle Break". You will see there is a red-brown dot placed at the left edge of the line you have set the breakpoint at.



You may set several breakpoints at different parts of your program. When the program pauses at the breakpoint you can continue with the Start button (F5) or use the "Step" controls, or stop execution with the Stop control.

What are watches

Watches in this sense are ways that we can automatically view the changes occurring to variables while running our program. We can see the watches that we set in the Watches tab of the ADL editor. Use the Ctrl-F9 shortcut key to set a watch on a selected variable in your program.

We can also examine any variable immediately with the Quick Watch ability. Just position the cursor on the item desired and select from the Debug Popup item the Quick Watch or Add Watch selection. Quick watches are displayed in the Immediate tab.

What is the call stack?

This strange item shows us the SUBs and FUNCTIONs that we are currently inside when running our program. For example if we have the following program

```
Sub Test3
  Msgbox "hello from test3"
End Sub

Sub Test2
  Test3
End Sub

Sub Test1
  Test2
End Sub

Sub Main
  Test1
End Sub
```

This simple program starts with Main "calling" test1 which in turn calls Test2 which calls Test3. If we step through our program to the line in Test3 where we print the message then the call stack will show that We started in Main, then Test1, Then Test2. This feature can be useful in complex programs to allow you to trace the execution of you program , observing its execution path.

What is the browser (fear not)

The browser is for very advanced ADL. Programmers who understand the object nature of VBA can use this feature to browse their objects. It is necessary for those who wish to use OLE communications with other programs such as Excel and WinWord. For the less knowledgeable I suggest that you skip it otherwise see Appendix J.

Button extensions

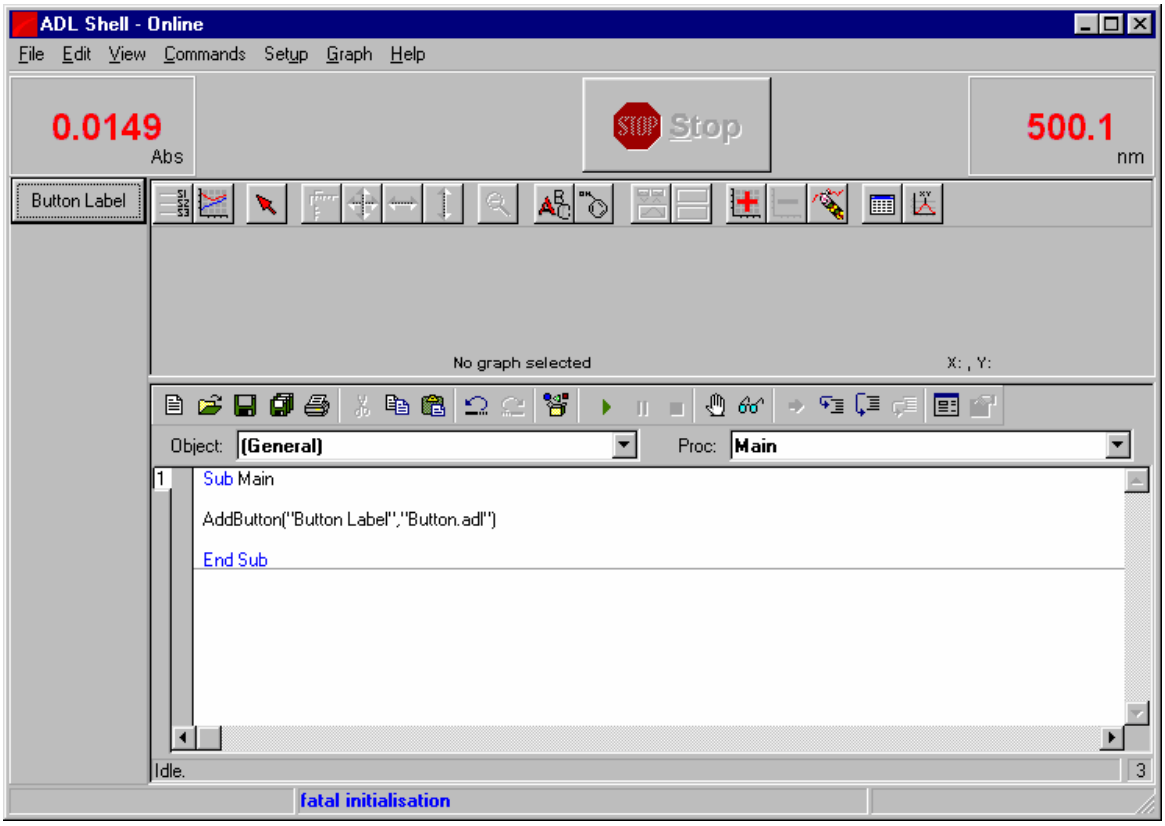
How to make a simple analyzer

We have provided WinADL with two types of commands that allow you to place buttons on the main application form and on the setup dialog of the application. Linked to each button that you create is an ADL program that will run when you click the button. By doing this you can make some interesting analyzer programs that provide a series of buttons to perform a simple set of tasks for your customer. For this section we will use two of our previously examined programs

Try the following code in the Sub Main section

AddButton("Button Label","button.adl")

When you run this simple program you see that the button panel on the left side is now replaced by YOUR button labeled "Button Label".



If it is not there then the most likely cause is that the ADL programs listed in the AddButton command do not exist. You may have to use the full file path to find it. Try pressing the button and observe.

As you can see we could make a series of buttons that could be the steps of a Cary analyzer. This could lead our users through stages of setup, collection and reporting of some program.

As well as AddButton there is RemoveButton, FocusButton, and EnableButton. To get rid of the panel that the buttons are on, you must remove all buttons.

Also we have created a similar capability to extend JUST the SETUP pages of Cary applications. The
AddSetupButton, and its' Remove, Focus and Enable variations will place buttons on an ADL tab in the setup window. Try it and see how you could use the Setup pages to bring up ADL dialogs with specific new accessory capabilities to be set by your users.

AddSetupButton("SetupButtonlabel","listbox.adl")

Files

How to read and write to files - the outside world

Files are a complex part of Sax Basic. In order to use them you need to understand some basic things.
Firstly files must be created if they don't exist or opened if they do exist.
Secondly you can read or write information with the file.
Thirdly you must close the files that you have opened when you are done.

When you open a file, you provide the file name and a reference number. The number is then used to refer to the file for all further operations.
Here is a simple example of writing information to a file

```
Sub Main
   A = 1
   B = 2
   C$ = "Hello"
   Open "MyFile" For Output As #1        ' the name is "MyFile" and the number
is 1
   Print #1,A;",";B;",""";C$;"""         ' now we write to file number 1
   Close #1                              ' and finally close file 1 when done
End Sub
```

The #1 is the File Number that is referenced for all file operations between the OPEN and CLOSE statements. The PRINT statement writes information to the file. The semi-colon ";" character between items allows us to write several items on one line. Numbers are automatically written out as if you had performed a STR$() operation on them.

We could then read back our information as follows

```
Sub Main
   ' read the first line of MyFile and print it
   Open "MyFile" For Input As #1
   Input #1,A,B,C$
   Cprint(A,B,C$)          ' on the status line
   Close #1
End Sub
```

If you want to understand more on files then the help manual contains many more examples and operations. Note that files can also be COM ports, although we have more sophisticated ADL interfaces to COM ports (see the RS232 commands)

File Dialogs

Cary WinADL provides the ability to use the standard File Open and Save dialogs that each application uses. These can be simply be displayed with the Show…Dlg

operations. The dialog shown will have the application two letter suffix, e.g. SW for the SCAN application.

ShowOpenDlg(FileType)
ShowSaveDlg(FileType)

Just specify the FileType constant from the Cary list in appendix A. For example try

ShowOpenDlg(Data_Files)

A very useful dialog is the ShowNodeDlg dialog. With this, and a knowledge of the Cary database nodes you can change the Cary model (when off-line) to any available model. This is useful for simulating different models, or creating methods to be run on different Cary instruments. You must enter the following information in the Alt-Y dialog and press OK.

ShowNodeDlg("Model")

When the dialog appears, choose

0 for Cary 50
1 for Cary 100
3 for Cary 300
4 for Cary 400
5 for Cary 500

Reports

Lining up columns

If you want to have a report that has items in two or more columns then you need to use what is called a "monospaced" font. These types of fonts make all characters of equal width. With most fonts the character width varies with different characters, making lining up of columns quite difficult. Fortunately Windows has such fonts. "Courier New" is a monospaced font. With our TableHdr, TableFmt and TableData commands you have all the capabilities already given to you.

Try the following code (refer to Appendix C) for details on the format specifiers.

```
Sub Main
        TableFmt("###.###","000","#0.00E+00")
        TableHdr("One big line","two","third column")
        TableData(1.2,2%,45.7)
        TableData(4.567,5%,125.67)
End Sub
```

The ADLConvert program will automatically convert all DOS and OS/2 table formats to the new formats. If you find difficulty understanding the new specifiers then try taking your old DOS or OS/2 table commands and running them through the converter.

Heading types

We have provided you with some convenient functions to quickly place a heading into a report in large bold font. Try the

ReportMasterHeading("Big heading")

Or

ReportSubHeading("Smaller heading")

Changing the font

The following example is from the ADL training CD.

```
ReportMasterHeading("Master Heading")
ReportSubHeading("Sub Heading")
ReportSpecialInfo("Special information")
' change the system report font from "Courier New" to "Arial" and make it 18 point bold
ReportSetTextFont("Arial", 18, 0, 1)
LPrint("General text")
TableFmt("@@@@@@@@@","#0.000")
TableHdr("Sample Names","Abs data")
For i = 1 To 5
   TableData("Sample "+CStr(i),CSng(i*Rnd()))
Next i
' this will change ALL general text to this font, including prior text
'Set the report now in Arial, 8, normal font
 ReportSetTextFont("Arial", 8, 0, 0)
 LPrint("More text")
'Add one line in an alternate Courier New 20 font
ReportTextFontAttr("Another line of text", "Courier New", 20)
'The next line is in the Arial font still
ReportTextWithFont("StrVal As String")
ReportAttachText(2,"   Attached Text")        ' to line 2
'Add a modified banner at the start of the report
ReportModifiedBanner("Text has been modified")
' Insert text at a prior line
ReportInsertText(4, "Inserted text")
```

Graphics

ActiveTrace

Traces are analogous to continua (data). A trace is the graphical picture of the continuum as seen in one or more graphs. A trace has color and style. A continuum becomes a trace when it is Shown (ShowCtm) or Plotted (Plot). There may be more continua than traces if continua are made for data manipulation and not Plotted or Shown. Therefore we have two ways of getting at data, either as continua or traces. Just remember that traces are graphical views. The data is the same with a few embellishments.

The string function ActiveTrace will give you back the string information that you need to manipulate the focused or active trace. This can be used in conjunction with the ActiveGraph string function. Also in this area of manipulation are SelectTrace and SelectGraph to choose the item that you want. This will force the selection, rather than assuming that the user has not changed it away from the program created selection. For example you may want to prompt the user to add a label to your trace. To ensure this operation succeeds you may need to have

SelectTrace("My Continuum")

ShowAddTextDlg

Placing, sizing, scaling, line and symbol drawing

Graphs can be moved and sized with the following commands.

PositionGraph(GraphName as String, GraphLeft as integer, GraphTop as integer)
SizeGraph(GraphName as String, GraphWidth as integer, GraphHeight as integer)

The graph scale is 0 to 1000 for nothing to full screen. With these two commands you can create a series of equally spaced graphs just suited to your needs.

Graphs can have lines and symbols drawn on them with the following series of commands

DrawLine(GraphName as String, X1 as single, Y1 as single, X2 as single, Y2 as single, PenColor as integer)

And

PlotSymbol(GraphName As String, X as single, Y as single, Character As String, Color As Integer)

The symbol that you plot can be any character in the alphabet or even a text string. This can be very useful if you want to place a heading on a graph for example.

These items are not associated with any traces in a graph but are related to the graph only. This means that when you delete all traces, the symbols and lines remain until you ClearSymbols or ClearLines from a graph.

Annotations

These are items (text or pictures) that can be added to any trace or to sections of the graphics region. The following ADL lines demonstrate the capabilities (assuming there is a trace and a graph)

```
RemoveAllAnnotations
a$=AddAnnotation(Text_Annotation, "A text label", ActiveGraph, ActiveTrace, 500, 50)
b$=AddAnnotation(Peak_Annotation, "A peak label", ActiveGraph, ActiveTrace, 600, 15)
MsgBox "There are now two annotations on the screen, and now to add a picture"
c$=AddAnnotation(Bitmap_Annotation, "C:\Windows\Tiles.bmp", ActiveGraph, ActiveTrace,
SetAnnotationVisible(a$, 0)
MsgBox "Click the graph and one annotation will disappear, another will move"
MoveAnnotation(b$, 400, 15)
```

Continuums

Labels and modes

Continuums contain their own X and Y labels that become displayed on a graph whenever the continuum/trace becomes the focused trace in a graph. You can set the labels with the commands. A continuum is created with the current X and Y modes of the database so you will need to set these BEFORE you make your new continuum. As we saw earlier we use the NewCtm string function to do this and the SetVal command to set the modes

SetVal("X Mode", Wavelength_Xmode)
SetVal("Y Mode",Abs_Ymode)
CtmName$=NewCtm("My New Data")

SetCtmXLabel(Ctmname$,"My X label")
SetCtmYLabel(Ctmname$,"My Y label")

We can put data in with the CurveFitData commands from DOS and OS/2 and as well we can use the
AddCtmData command to do the same. We need not "close" a continuum as we had to in DOS with the CurveFitData(…,2) option.

One difference with WinADL applications is that the continuum can be linked (or associated) with a graph (or many graphs) in order to be automatically displayed as data is added (as discussed in the previous section on Traces). This linking needs to be done once in your program.

ShowCtm("Graph Name","Continuum Name")

This command does the linking for you.

After that as you add data to your continuum it will be drawn automatically in each graph that you have mentioned in the ShowCtm commands that you have used. For example if you had two graphs then you can display the same continuum in both graphs as follows

ShowCtm("Graph 1","Continuum Name")
ShowCtm("Graph 2","Continuum Name")

We also have the PLOT command from DOS that will plot a given continuum in the active graph. E.g.

PLOT(“Continuum Name”)

Smoothing, derivatives

The old Smooth and Derivative commands of prior ADLs are available in slightly different syntax. To smooth a continuum we use

SmoothedCtm$ = Smooth(CtmName as String, SmoothOrder As Integer, Optional SmoothInterval)
E.g. an 11 point smooth is
SmoothedCtm$ = Smooth(“CtmName”, 11)

To get a derivative we use

DerivatizedCtm$ = Deriv(CtmName as String, DerivOrder As Integer, Optional Interval, Optional FilterSize)
E.g. a 1st order ferivative is
DerivatizedCtm$ = Deriv("CtmName", 1)

To get a cubic Spline fit resultant continuum we use

SplineCtm$ = Spline(CtmName as String, NewCtm As String, DataFactor As Single)
E.g. to fit a spline to a continuum at a factor of 5 (5 times "interpolation")
SplineCtm$ = Spline("CtmName", "SplineCtmCtm", 5)

UserDataForm, Header, Headerstr

Continuums have a 50 slot header for keeping SINGLE values in just as we used to have. We use the Header and SetHeader commands to access them just as before with the exception that for HeaderStr and SetHeaderStr the old "slots" are now replaced by the names of the column headings as seen in the user data form. E.g.

```
SetHeaderStr("My Ctm","PH,"7.8")
A$=HeaderStr("My Ctm",PH")

SetHeader("My Ctm",14,450.7)
A=Header("My Ctm",14)
```

Also continuums use the UserDataForm to keep STRING information. This allows us to use the SetHeaderStr and HeaderStr command to access the UserDataForm. We can print the UserDataForm with ***PrintHeader***.

Peaks

These have changed a little bit, with some improvements. We still have the basic PEAK function but it changes slightly to determine ahead of time what type of peak we want to find.

Peak(CtmName as String, PeakValue As Single, PeakType As Integer, Optional X1 as single, Optional X2 as single)

PeakValue is still 0 to initialize, 1 to get the next PeakType

Note we can now restrict the peak search to a sub-section of the continuum with the optional X1 and X2 values. This eliminates the need to use the MIDC to get a sub-section of the continuum for peak detection.

The **FIELD_X** and **FIELD_Y** values have gone. They are replaced by the functions **PEAKX** and **PEAKY** which give back the result of the PEAK command which are more meaningful names.

The new command ***BestPeak*** will find the highest maximum or the lowest minimum over the peak search range.

As a simple example we could do the following

```
a=peak(Collect$,0,Max_Peak)    'initialize and find maximums
If a <> 0 Then
  a = peak(Collect$,1,Max_Peak)
  XVal = PeakX
```

YVal = PeakY
End If

DataBase

Our database has some other useful items and uses.

Status dialog

We can add entries to the "float on top" status dialog. To do this we use

ClearStatusDisplay
AddStatusEntry(NodeName As String)

Often we want to format a result just the same way as some displayed item. If that item is a database item then we can "pass" our number to the database and tell it to format it AS IF it was the value in the database node. For example you may wish to format a result to the exact format of the Y display. To do this we use

Formatting

A$ = GetFormattedByNode("Current Y",MyValue)

This is extremely useful to make your reports more presentable as (for example) we can format any value by the current system formatting. In the case of wavelength values that you may wish to have in an ADL program that get reported, you can format the wavelengths to the current system wavelength format using

A$ = GetFormattedByNode("Goto Wavelength",MyWavelengthValue)

General

Time/Date

SAX Basic provides us with several ways to use the time and date of the computer. Examine the SAX help for TIME, DATE, NOW and other useful items. See that we can get the current date with the NOW function. For example in a report we could do the following

CPRINT(Now)

Or

LPRINT(Now)

There are also options to perform mathematics on time and date differences (See DateDiff in the help).

Writing timed operations

With DOS and OS/2 we could write special timed FOR NEXT loops. We can have the same capability in WinADL but we need to do it a bit differently. We use the TIMER function of SAX Basic to give us a time value. For example we could write a simple delay loop.

```
StartTime = TIMER       ' this is always seconds
DO
  Cprint(TIMER-StartTime)
  DoEvents              ' good practice
LOOP Until (Timer-StartTime) >= 5
```

SAX Basic also has a WAIT function which also delays in seconds

```
WAIT 5    ' will delay for 5 seconds
```

The first delay loop is more useful as it gives us some way to inform the user that we are delaying. The help on Timer indicates that it is seconds since midnight. Therefore the example will have a problem at midnight. In order to get around this we need the following code.

```
HoldTime = 10       ' seconds
StartTime = Now
Do
  CurrentTime = Now  TimeWaited = DateDiff("s", StartTime, CurrentTime)
  DoEvents
Loop Until (TimeWaited > HoldTime)
```

DoBaseline

Because we no longer have a BASE SYSTEM as we use to have in DOS or OS/2 the concept of a BASELINE is only relevant to those applications that scan. However in order to make life easier for ADL we have implemented the **DoBaseline** command. This command will use the current settings in the database for
Scan Start
Scan Stop
X Mode
Common SAT
Common Interval

On completion of the collection a new continuum "Baseline#" will exist. You can check if the baseline is active with the BaselineActive variable.

Adding DialogFunc capability

When we run our dialogs that we created with the dialog editor we sometimes want to modify the dialog contents depending upon the operations done. For example if we had a checkbox on our dialog and we wanted to make an entry field available only when the checkbox is checked then we need to understand dialog functions.

Dialog functions are special ways to trap the events that occur when running a dialog. We can trap the creation of the dialog, each button press or value change, each combo or list change, change of focus or any function key press. Lets do this by example. Start with the one called Question.adl and run it. You will see that a button appears or disappears depending upon operations in the dialog.

The basic layout for a Dialog function is

```
Function Dialogfunc(DlgItem$, Action%, SuppValue%) _
   As Boolean
   Select Case Action%
   Case 1 ' Dialog box initialization
   ...
   Case 2 ' Value changing or button pressed
   ...
   Case 3 ' TextBox or ComboBox text changed
   ...
   Case 4 ' Focus changed
   ...
   Case 5 ' Idle
   ...
   Case 6 ' Function key
   ...
   End Select
End Function
```

This can be either manually entered or automatically created by the dialog editor. The latter approach is done by editing the properties of the dialog (double-click on the dialog background) and entering a function name in the Dialog Function entry field of the properties box. When saving the dialog back to ADL you will be prompted to create the framework of the dialog function. Respond with “Y” for yes.

An example is

```
Sub Main
   Begin Dialog UserDialog 200,120,.DialogFunc          ' we put the the reference
here noting we are using a dialog function
   Text 10,10,180,15,"Please push the OK button"
   TextBox 10,40,180,15,.Text
   OKButton 30,90,60,20
   PushButton 110,90,60,20,"&Hello"
   End Dialog
   Dim dlg As UserDialog
   Debug.Print Dialog(dlg)
End Sub

Function DialogFunc%(DlgItem$, Action%, SuppValue%)
   Select Case Action%
   Case 1 ' Dialog box initialization
      Beep
   Case 2 ' Value changing or button pressed
      If DlgItem$ = "Hello" Then              ' trap the Hello press
         MsgBox "Hello"
         DialogFunc% = True 'do not exit the dialog
      End If
   Case 4 ' Focus changed
      Debug.Print "DlgFocus=""";DlgFocus();""""
   Case 6 ' Function key
```

```
      If SuppValue And &H100 Then Debug.Print "Shift-";          ' examples of
getting Alt,Shift,Ctrl
      If SuppValue And &H200 Then Debug.Print "Ctrl-";
      If SuppValue And &H400 Then Debug.Print "Alt-";
      Debug.Print "F" & (SuppValue And &HFF)
   End Select
End Function
```

Sending Email

We can send information via EMAIL provided that your computer has TCPIP email capability and you know the mail server name. The latter must be set in the database as

SetVal("Mail Host","hermes.osi.varian.com")

Then simply send as

SendEmail(kim.darby@osi.varian.com,"c:\autoexec.bat")

For example this could be simply sent at the end of your ADL program, sending you your batch file of data.

Hooks

As in the DOS and OS/2 we have placed what we call "hooks" inside our applications. These hooks allow us access to such points as the press of the start button, the end of the run sequence, pre and post reading etc. The names chosen are application specific. Our basic list of hooks is

online.adl
offline.adl
preseq.adl
postseq.adl
zero.adl
recalc.adl
preread.adl
postread.adl
shutdown.adl
postsetup.adl (ADLShell build > 50)

To make the link to each application we need the two letter prefix for the application. Then we construct the file name as follows.

Prefix + "_" + HookName

Prefix is one of the following-

GP AppName:'GALP'
AD AppName:'ADL'
TD AppName:'Tablet dissolution
SR AppName:'Simple reads'
KN AppName:'Kinetics'
CN AppName:'Concentration'
VO AppName:'Validate'
TM AppName:'Thermal'
LA AppName:'Lampalign'
DN AppName:'RNA-DNA'
MT AppName:'Maths'
EK AppName:'Enzyme Kinetics'
SK AppName:'Scanning Kinetics'
CO AppName:'Common'
SI AppName:'System Information
BS AppName:'Base System'
SW AppName:'Scan'
SG AppName:'Sunglasses'
FP AppName:'Fabric Protection
CL AppName:'Color'
AB AppName:'Advanced Reads

For example, to make a program that always runs when the Start button is pressed in the Scan application we create an ADL file labelled **"SW_PreSeq.adl ".**

ADL Hooks are activated by selecting the menu item “ADL Hooks” in the Cary application. When selected you see the keyword [ADL] in the application title bar indicating that you want to use the hooks. Otherwise they are OFF by default. We have several examples of performing pre-scans on data, post-reads changing the values of data etc.
Individual applications have their own extra hooks. E.g. The scan application also has **PreScan** and **PostScan** ADL files. To see the power of the ADL hooks, the

Varian web site has downloadable ADL programs that are set up as hook programs. These allow standard applications to be greatly enhanced with ADL extensions.

ADL hook style programs can be run from the ADL Program Selector. This path will hide the needs to make all the necessary links and settings. The user only has to select the name provided (See the section on ADL Program Selector).

Defensive Programming - Trapping errors and handling them

Sax Basic allows us to trap errors in the code in the VBA style of error handling. For example we can do the following

```
Sub Main
On Error GoTo ErrorSpot
  Open "AnyFile" For Input As #1
  Close #1
  Exit Sub
ErrorSpot:
  MsgBox "I got an error"
End Sub
```

If the file AnyFile does not exist then we get an error. As you can see we can handle exceptions and errors with careful programming. The basic rules are that each SUB or FUNCTION can have its own error handler. The handler is activated by the key words ON ERROR. Any error encountered during running will cause the handler to be used. This is particularly useful when dealing with disk files or numeric computations possibly involving computational errors. See the SAX Basic manual for details and extensions.

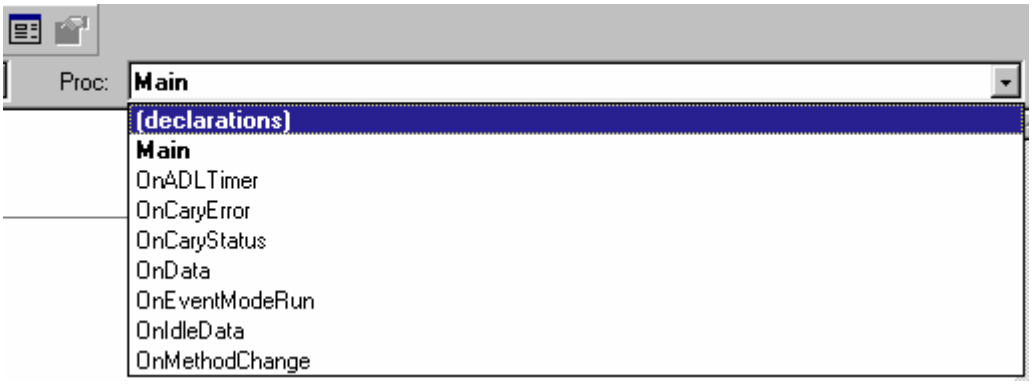
Event nature of VBA

VBA allows concurrent operations. That means that while your ADL program is running you can allow other Windows events (mouse clicks etc.) to be processed. This can cause disastrous results if you aren't aware of what is happening. The usual trick is to make sure that the section of program you are running can't be run again while you are still running it. This is called re-entrancy. The way you do this is to disable potentially harmful buttons while things are happening. There are some examples of use on the Cary installation disk and in the WinADL compilation on the Varian web site..

Later extensions to ADL (see ADLShell version 2.01(64) or later) have event routines which provide the user with much more flexibility to control their system. There are seven new data handlers.

OnData
OnIdleData
OnMethodChange
OnCaryError
OnCaryStatus
OnEventModeRun
OnADLTimer

These handlers can be seen in the Proc: section of the ADL editor. This is a "drop-down" selector that shows you all your Subs and Functions. Examination of this selector in the ADLShell will show these new handlers.



Selecting any of them will bring up the skeleton required by ADL to interact with your program. For example the **OnData** handler has the following layout.

Public Function OnData(X As Single, Y As Single, YRear As Single) As Boolean
End Function

We have provided the X and Y values for each data point collected by Cary. Filling in this we get the following ADL program that uses the variable **ModifiedY (or ModifiedX or ModifiedYRear)** to alter the Y value that is collected. By doing this we could construct a complex mode conversion based upon any algorithm that we desire. The resultant data will pass on to the application for normal handling.

' We can change the data by altering ModifiedX, ModifiedY, ModifiedYRearPublic Function OnData(X As Single, Y As Single, YRear As Single) As Booleany3=Rndlprint("X:",x," Y:",y3)modifiedy=y3
End Function

Use this in conjunction with the following Sub Main that performs data collection.

```
Sub Main    StartPrint    ReportMasterHeading("Interactive Data Collection")
SetVal("Scan Start", 500)    SetVal("Scan Stop", 400)    SetVal("Scan Mode",
Wavelength_XMode)
   a$=NewCtm("Test Ctm")
  clearctm(a$)
  b$=NewGraph("Test Graph")
  ShowCtm(b$, a$)
SetupInst
  startinst(a$)
  While ScanningInst
    DoEvents
  Wend
End Sub
```

We can also alter the big red numbers (the idle data) with the **OnIdleData** handler. This information is processed less frequently than scan data as the update rate is mainly for visual display

Public Function OnIdleData(X As Single, Y As Single) As Boolean
End Function

Fill it is as follows.

' Change just the Idle data (the big red numbers)Public Function OnIdleData(X As Single, Y As Single) As Boolean
y3=Rnd
lprint("X:",x," Y:",y3)
modifiedy=y3

End Function

The **OnMethodChange** handler is slightly different. It allows us to observe any REGISTERED database nodes for changes and be told when they change. The skeleton for this is…

Public Function OnMethodChange(NodeIndex As Integer, NodeName As String, NodeValue As String) As Boolean

End Function

To use this handler we need to register interest in some items we do this with the **RegisterMethodChange** command. Place the following line into the Sub Main we previously had that scanned the data.

```
RegisterMethodChange("Current Wavelength", True)  ' watch changes in this itemPublic Function OnMethodChange(NodeIndex As Integer, NodeName As String, NodeValue As String) As Boolean
LPrint(NodeName, NodeValue)
End Function
```

The timer event is useful to create a response at a time interval. The following example shows how to run this handler. The **Timer1Start** function has an optional **TimerInterval** parameter. If this is not provided the timer event is disabled. If the TimerInterval is provided then the timer is enabled and the value is the (approximate) time interval in milliseconds that the **OnADLTimer** event is triggered. The exact time since the Timer1Start command was issued is provided as a parameter in the handler.

```
Sub Main
  Timer1Start(500)   ' an optional timer interval in milliseconds activates the timer event, no value disables the timer
  For i = 1 To 30000
    DoEvents
  Next i
End Sub

' this event responds to the timer at the given interval (if specified)
Public Function OnADLTimer(ElapsedSeconds As Single) As Boolean
  cprint(ElapsedSeconds)
End Function
```

Our last two simpler handlers look for Status and Error messages from Cary applications programs. They work in conjunction with the ADL variable **IgnoreStsErr** to allow certain messages to be hidden or seen by the application.

```
Public Function OnCaryStatus(StatusValue As Integer, StatusString As String) As Boolean
  LPrint("Status: ",StatusString, IgnoreStsErr)        ' print all status and whether it is ignored by the UIF
End Function

Public Function OnCaryError(ErrorValue As Integer, ErrorString As String) As Boolean
  LPrint("Error: ",ErrorString, IgnoreStsErr)        ' print all errors and whether it is ignored by the UIF
End Function
```

Advanced Event Handling

The **OnData** event handler can sometimes get "tied down" by Cary and as such there is a more advanced (and preferred) way to collect using this event. To do this we need to select FULL EVENT operation. This is done with the keyword **EventMode**. Set it true to activate and set it false to deactivate event operations. In this mode there is no Sub Main program. You have a series of event handlers ONLY that you can fill in. The use of the **OnEventModeRun** handler allows us to create operational programs that don't wait for things to happen.

Public Function OnEventModeRun() As Boolean

End Function

This is the place that you put your startup code for your experiment. For example we get the collection running by putting all code here except the Wait Around bit. The following example illustrates how to lay out the code

```
' This bit is done when you press the ADL run button
Public Function OnEventModeRun() As Boolean
   StartPrint   ReportMasterHeading("Interactive Data Collection")
SetVal("Scan Start", 500)   SetVal("Scan Stop", 400)   SetVal("Scan Mode",
Wavelength_XMode)
   ' we need 1 ctm for the StartInst command to collect ""ordinary data"
   a$=NewCtm("Test Ctm")
  clearctm(a$)
  ' we need another  ctm for the OnData event to collect ""Modified data"
  c$=NewCtm("Another Ctm")
  clearctm(a$)
  b$=NewGraph("Test Graph")
  ShowCtm(b$, a$)
  ShowCtm(b$, c$)
  SetupInst
  startinst(a$)      ' start but do not wait
End Function

' This bit traps data from Cary and allows us to place it into our second
continuum
Public Function OnData(X As Single, Y As Single, YRear As Single) As Boolean
AddCtmData("AnotherCtm",X,Y+0.5)            ' for example we offset the data
End Function
```

Using this approach we could (for example) build up an ADL program that would allow us to collect all data, trap the end of the collection using the "Collecting" database node change event and prepare for another collection. This ADL is left to you to implement as an exercise.

Linking to other applications and libraries

Cary ADL has the ability to launch other applications. In DOS this was through the Spawn_Prog command. In the new environment it is through the SHELL command. For example to launch the windows calculator run the following.

```
Sub Main    X = Shell("Calc")                    ' run the calc program    AppActivate
X    SendKeys "% R"                    ' restore calc's main window    SendKeys
"30*2{+}10=",1       ' results in 70End Sub
```

Talking to the program is done with the SendKeys command. This allows (as its name indicates) to send key strokes to the application. No information can be obtained back from the application this way. See the ADL help about SendKeys for more information

DDE linkage

Cary ADL has the ability to link to other applications using DDE (Dynamic Data Exchange). This allows sending and receiving of information between compliant applications. Cary does not directly have DDE with its data streams as it did in OS/2 but through ADL it can send and receive via DDE.

The example in the help shows how to talk to Windows "Program manager" program.

```
Sub Main    ChanNum = DDEInitiate("PROGMAN","PROGMAN")    DDEExecute
ChanNum,"[CreateGroup(XXX)]"    DDETerminate ChanNumEnd Sub
```

OLE linkage

Please examine appendix J for some examples on this means of linking to external programs. It is for the very advanced users only.

External libraries

Libraries are better known as DLLs in windows parlance. These are (generally) collections of useful things that can be accessed from any running program. The trick is knowing how to access this information.
Libraries provide a public interface to the world. This allows programs to know what the relevant functions and subs are inside the library. Windows provides many DLLs in its operating system. Some of these are useful to us but most are of no relevance here. If you know how to create your own library using a programming language such as C++ or DELPHI then this section may interest you.

The syntax for linking to ADL is the **DECLARE** statement. This tells ADL the name of the DLL and the functions or Subs that you want to access inside it. Also the Declare statement describes the exact syntax for access. For example we can access the Windows User32 library to find the title of the active window. This uses the data type identifier "&" for Long integer (32 bits).

```
Declare Function GetActiveWindow& Lib "user32" ()Declare Function GetWindowTextLengthA&
Lib "user32"  (ByVal hwnd&)Declare Sub GetWindowTextA Lib "user32"  (ByVal hwnd&, ByVal
lpsz$, ByVal cbMax&)Function ActiveWindowTitle$()    ActiveWindow = GetActiveWindow()
TitleLen = GetWindowTextLengthA(ActiveWindow)    Title$ = Space$(TitleLen)                ' you
must fill up the string with characters to be replaced
   GetWindowTextA ActiveWindow,Title$,TitleLen+1    ActiveWindowTitle$ = Title$End
FunctionSub Main    Debug.Print ActiveWindowTitle$()End Sub
```

We also provide a program to link to the Cary 50 sipper via a special DLL. This program is available on our web site.

Prac Session

The following example programs are for development and/or conversion.

Simple reads example
Create an ADL dialog version of the simple reads program. The user enters the wavelength, SBW and averaging time and the number of samples to read. You may also want an initial sample label field.

Cell changer
Perform a simple multi-cell collect in Abs mode. Provide a means to enter the start and stop times in minutes. Provide a means to select which cells the user wants to use (1 to 6 only). Provide graphics scaling as entered by the user.

Wine example
The example here is a simple analysis. The user enters a wine label, two wavelengths (default 420 and 520 nm). Two calculations are performed on data read in Abs at the wavelengths. The color intensity is the sum of the Abs readings. The color hue is the ratio of the Abs at 420 to the Abs at 520. Compute and print a report of these in a table. Make the user determine the last sample read based upon response to a message prompt.

Beer example
The example is to analyze for Alpha and Beta acids in hop powders and pellets. ADLNew21.adl is the basis for this program. Translate and make it into an ADL Dialog-based program. If time permits, make this program into a two button Analyzer. The first button is "Parameters" and the second is "Collect". These buttons link to two like-named ADL programs that perform the two parts of the analysis. Use the system User variables as storage or create your own database nodes as storage.

Appendix A - Cary Constants

Cary has many pre-defined constants. Below is a list of ones that relate to the basic instrument and filing operations. There are many more that you can see by using the ADL Quick Reference.

```
CaryExtensionsVersion

File constants
Method_Files                    = 0
Data_Files                      = 1
Report_Files                    = 2
Template_Files                  = 3
Graphics_Files                  = 4
Batch_Files                     = 5
OS2Data_Files                   = 6
AsciiXY_Files                   = 7
DosData_Files                   = 8
BaseLine_Files                  = 9
ADL_Files                       = 10
Grams_Files                     = 11
Any_Files                       = 12

Y Mode constants                                X Mode constants
Abs_YMode                       = 0             Wavelength_XMode =
0
PercentT_YMode                  = 1             Angstrom_XMode   =
1
FactorAbs_YMode                 = 2             Wavenumber_XMode =
2
PercentR_YMode                  = 3             Minutes_XMode    =
3
RawT_YMode                      = 4             Angle_XMode      =
4
User_YMode                      = 5             Millimeter_XMode =
5
LogAbs_YMode                    = 6             Degrees_XMode    =
6
AbsoluteR_YMode                 = 7
KubelkaMunk_YMode               = 8
LogKM_YMode                     = 9
LogOneOnReflectance_YMode       = 10
Reflectance_YMode               = 11

Node constants
String_Node                     = 0
Real_Node                       = 1
Integer_Node                    = 2
Stepped_Real_Node               = 3
Stepped_Integer_Node            = 4
Choice_Node                     = 5
Boolean_Node                    = 6

Peak constants
Max_Peak                        = 1
Min_Peak                        = 2
Zero_Peak                       = 3

Instrument constants
Beam_Mode_Single_Front          = 0             Detector_UvVis  =
1
Beam_Mode_Single_Rear           = 1             Detector_Nir    =
2
Beam_Mode_Double_Fixed_Slit     = 2             Grating_UvVis   =
1
Beam_Mode_Double_Fixed_Energy   = 3     Grating_Nir     = 2
Beam_Double_Auto_Select         = 4             Slit_One_Third  =
1
Beam_Mode_Test_Mode             = 5             Slit_Full       =
0
Beam_Mode_Dual_Single           = 6             Source_Uv       =
1
Cal_Sig_Proc                    = 0             Source_Vis      =
2
Cal_UvVis_Wl                    = 1             Source_Third    =
3
Cal_Nir_Wl                      = 2
Cal_Nir_Mem_1                   = 3
```

```
Cal_Nir_Lin_1                   = 4
Cal_Nir_Lin_2                   = 5
Cal_Check_UvVis                 = 6
Cal_Check_Nir                   = 7
Cal_Pm_Zero_1                   = 8
Cal_Pm_Zero_2                   = 9
Cal_Nir_Mem_2                   = 10
```

Appendix B - Database node names

(For the current set of node names in any application, use the ADL Quick Reference)

X Interval	Last transmission value	NIRDetectorCal[5]
X Start	last PMV gain index	NIRDetectorCal[6]
Beam mode	last Slit width	NIR interval
Old Collecting	last rear T value	NIRPeakWls[1]
Detector	Model	NIRPeakWls[2]
Detector changeover	NIR averaging	NIRPeakWls[3]
Filter	NIRDetectorCal[1]	NIR reference level
Gain	NIRDetectorCal[2]	NIR slit width
Grating	NIRDetectorCal[3]	NIRWavelengthCalA
Grating changeover	NIRDetectorCal[4]	NIRWavelengthCalB
NIRWavelengthCalC	sig_procc_calib_offset	Source
NIR Wavenumber changeover	sig_procc_calib_gain[1]	Source
PGA Gain indices front on/off UV	sig_procc_calib_gain[2]	Source
PGA Gain indices rear on/off Vis	sig_procc_calib_gain[3]	Source
PGA Gain indices dark on/off third	sig_procc_calib_gain[4]	Source
PGA mode	sig_procc_calib_gain[5]	Status
Shutter front	sig_procc_calib_gain[6]	Target SNR
Shutters open/closed front averaging	sig_procc_calib_gain[7]	UVVIS
Shutters open/closed rear interval	sig_procc_calib_gain[8]	UVVIS
Shutter rear	Slit height	UVVISPeakWls[1]
UVVISPeakWls[2]	Version slit	Busy
UVVISPeakWls[3]	Current Wavelength	Resetting
UVVIS reference level	Scan mode	Collecting
UVVIS slit width	Scan start	Offline
UVVISWavelengthCalA	Scan stop	last_RD_value
UVVISWavelengthCalB	Scan points	Last_D2Peak_Wavelength
UVVISWavelengthCalC	Calibrate mode	Last_Time
UVVIS Wavenumber	UVVIS wavenumber interval	Instrument_Time
Version main	NIR wavenumber interval	Lamp1 Time
Version mono	Goto Wavelength	Lamp2 Time
Lamp3 Time	NIR Scan rate	Simulate
Last_Read_Time	Common Scan rate	Operator
Current Source	Common Interval	Method Owner
Current Filter	Common Slit Width	Report Owner
Current Grating	UVVIS Energy	Batch
Current Detector	NIR Energy	Comment
UVVIS SAT	Common Energy	X mode
NIR SAT	Current Energy	Y mode
Common SAT	Beam Interchange	Y mode User
UVVIS Scan rate	Simulate File	Y mode T
Y mode %R	Zero Mode	Collect Mode
Y mode Abs*f	Basline Mode	Abs Factor
Y mode %T	Current Method	Current X
Y mode Abs	Current Batch	Current Y
Y mode Log 1/R	Current Method Date	Online
Y mode Log KM	Current Method Time	Running Sequence
Y mode KM	Method Modified	Report Graph Scale
Y mode Absolute R	Method ReadOnly	Report AutoPrint
Y mode Log Abs	Report ReadOnly	Report Results

Y mode R	User Collect	Report Graph
Report Parameters	Accy Measure2	Accy Measure12
Report Company Logo	Accy Measure3	Accy Measure13
Report Data Form	Accy Measure4	Accy Measure14
Y min	Accy Measure5	Accy Measure15
Y max	Accy Measure6	Accy Measure16
Idle 50	Accy Measure7	Accy Measure17
SPS Online	Accy Measure8	Accy Measure18
SPS On	Accy Measure9	Accy Measure19
Accy Measure0	Accy Measure10	Accy Measure20
Accy Measure1	Accy Measure11	Accy Measure21
Accy Measure22	Accy Measure32	Accy Measure42
Accy Measure23	Accy Measure33	Accy Measure43
Accy Measure24	Accy Measure34	Accy Measure44
Accy Measure25	Accy Measure35	Accy Measure45
Accy Measure26	Accy Measure36	Accy Measure46
Accy Measure27	Accy Measure37	Accy Measure47
Accy Measure28	Accy Measure38	Accy Measure48
Accy Measure29	Accy Measure39	Accy Measure49
Accy Measure30	Accy Measure40	Accy Measure50
Accy Measure31	Accy Measure41	Accy Measure51
Accy Measure52	Accy Measure62	Accy Measure72
Accy Measure53	Accy Measure63	Accy Measure73
Accy Measure54	Accy Measure64	Accy Measure74
Accy Measure55	Accy Measure65	Accy Measure75
Accy Measure56	Accy Measure66	Accy Measure76
Accy Measure57	Accy Measure67	Accy Measure77
Accy Measure58	Accy Measure68	Accy Measure78
Accy Measure59	Accy Measure69	Accy Measure79
Accy Measure60	Accy Measure70	Accy Measure80
Accy Measure61	Accy Measure71	Accy Measure81
Accy Condition	Cell5	Cell15
Status Display	Cell6	Cell16
Cell Changer	Cell7	Cell17
Multi Cell Zero	Cell8	Cell18
Max Cells	Cell9	Fibre
Last Valid Cell Zero	Cell10	Multi Fibre
Cell1	Cell11	Fibre1
Cell2	Cell12	Fibre2
Cell3	Cell13	Fibre3
Cell4	Cell14	Fibre4
Fibre5	Current Angle	Rack fitted1
Fibre6	Temperature Monitor	Rack fitted2
Fibre7	Current Block Temp	Rack fitted3
Multi cell baseline	Current Probe 1 Temp	Rack fitted4

Multi fibre baseline	Current Probe 2 Temp	Rack fitted5
Current Cell	Current Probe 3 Temp	Prompt on Stop
Cell in Use	Current Probe 4 Temp	Preserve current database
Current Fibre	Current Time	Autoconvert
Current Distance	Current Rack	AutoConvert Separator
Current RBA	Current Tube	Method Only Loaded
Terminated	Absolute Common Interval	Mail BCC
Model Name	Attached to Comms	Mail Body
Suppress Instrument Error	Bypass Dialogs	Peak Threshold
Hold Save On The Fly	Smart ReReads	Peak Style
Fast Save Rate	Mail Host	Peak Font Name
Slow Save Rate	Mail Port	Peak Font Style
Source Button Type	Mail Subject	Peak Font Size
Lamps on/off	Mail To	Peak Font Height
Absolute UVVIS interval	Mail From	Peak Y Decimals
Absolute NIR Interval	Mail CC	Peak Label Mode
Max Peak	All Peaks	Run ADL Enabled
ADL Path	RSA Fill Time	RSA Delay Time
ASCII Convert Mode	MM Per Cell	Steps Per MM

Appendix C - String conversion format specifiers and operations

Formatting

Number format specifiers.
User defined number formats can contain up to four sections separated by ';':

- form - format for non-negative expr, '-'format for negative expr, empty and null expr return ""
- form;negform - negform: format for negative expr
- form;negform;zeroform - zeroform: format for zero expr
- form;negform;zeroform;nullform - nullform: format for empty or null expr

Parameter Description

digit, don't include leading/trailing zero digits (all the digits left of decimal point are returned)
e.g. Format(19,"###") returns "19"
e.g. Format(19,"#") returns "19"

0 digit, include leading/trailing zero digits
e.g. Format(19,"000") returns "019"
e.g. Format(19,"0") returns "19"

. decimal, insert localized decimal point
e.g. Format(19.9,"###.00") returns "19.90"
e.g. Format(19.9,"###.##") returns "19.9"

, thousands, insert localized thousand separator every 3 digits
"xxx," or "xxx,." mean divide expr by 1000 prior to formatting
two adjacent commas ",," means divide expr by 1000 again
e.g. Format(1900000,"0,,") returns "2"
e.g. Format(1900000,"0,,.0") returns "1.9"

% percent, insert %, multiply expr by 100 prior to formatting

: insert localized time separator

/ insert localized date separator

E+ e+ E- e- use exponential notation, insert E (or e) and the signed exponent
e.g. Format(1000,"0.00E+00") returns "1.00E+03"
e.g. Format(.001,"0.00E+00") returns "1.00E-03"

- + $ ( ) space insert literal char
e.g. Format(10,"$#") returns "$10"

\c insert character c
e.g. Format(19,"\####\#") returns "#19#"

"text" insert literal text
e.g. Format(19,""""##""###""##""") returns "##19##"

Example:

Sub Main

```
    Debug.Print Format$(2.145,"#.00") ’2.15
End Sub
```

Date Format Specifiers.
The following date formats may be used with the Format function. Date formats may be combined to create the user defined date format. User defined date formats may not be combined with other user defined formats or predefined formats.

Parameter	Description
:	insert localized time separator
/	insert localized date separator
c	insert ddddd ttttt, insert date only if t=0, insert time only if d=0
d	insert day number without leading zero
dd	insert day number with leading zero
ddd	insert abbreviated day name
dddd	insert full day name
ddddd	insert date according to Short Date format
dddddd	insert date according to Long Date format
w	insert day of week number
ww	insert week of year number
m	insert month number without leading zero insert minute number without leading zero (if follows h or hh)
mm	insert month number with leading zero insert minute number with leading zero (if follows h or hh)
mmm	insert abbreviated month name
mmmm	insert full month name
q	insert quarter number
y	insert day of year number
yy	insert year number (two digits)
yyyy	insert year number (four digits, no leading zeros)
h	insert hour number without leading zero
hh	insert hour number with leading zero
n	insert minute number without leading zero
nn	insert minute number with leading zero
s	insert second number without leading zero

ss insert second number with leading zero

ttttt insert time according to time format

AM/PM use 12 hour clock and insert AM (hours 0 to 11) and PM (12 to 23)

am/pm use 12 hour clock and insert am (hours 0 to 11) and pm (12 to 23)

A/P use 12 hour clock and insert A (hours 0 to 11) and P (12 to 23)

a/p use 12 hour clock and insert a (hours 0 to 11) and p (12 to 23)

AMPM use 12 hour clock and insert localized AM/PM strings

\c insert character c

"text" insert literal text

Text format specifiers
The following text formats may be used with the Format function. Text formats may be combined to create the user defined text format. User defined text formats may not be combined with other user defined formats or predefined formats.

User defined text formats can contain one or two sections separated by ';':

- form - format for all strings
- form;nullform - nullform: format for null strings

Parameter Description

@ char placeholder, insert char or space

& char placeholder, insert char or nothing

< all chars lowercase

> all chars uppercase

! fill placeholder from left-to-right (default is right-to-left)

\c insert character c

"text" insert literal text

Example:

```
Sub Main
   Debug.Print Format("123","ab@c")  ' 12ab3c"
   Debug.Print Format("123","!ab@c") ' ab1c23"
End Sub
```

Special format specifiers are as follows.

Currency Same as user defined number format "$#,##0.00;($#,##0.00)"
Not locale dependent at this time.

Fixed Same as user defined number format "0.00".

Standard Same as user defined number format "#,##0.00".

Percent Same as user defined number format "0.00%".

Scientific Same as user defined number format "0.00E+00".

Yes/No Return "No" if zero, else return "Yes".

True/False Return "True" if zero, else return "False".

On/Off Return "On" if zero, else return "Off".

Example:

Debug.Print Format$(2.145,"Standard") ’ 2.15

String Operations

The following are some of the more useful commands that can be done with strings.

MID$-Function: This allows extraction of a section of a string. You give the string and ask for the start position in the string (1 is the first character) and number of characters to return.
MID$(“Varian”,2,3) This allows extraction of a section of a string, beginning at the start. You give the string and ask for the number of characters to return.
ari
MID$(“Varian”,1,3) returns “Var“
MID$(“Varian”,5,1) returns “a“

LEFT$- Function: This allows extraction of a section of a string, beginning at the start. You give the string and ask for the number of characters to return.
Left$(“UV-Lampe”,2) returns “UV“
Left$(“UV-Lampe”,3) returns “UV-“

RIGHT$- Function: This allows extraction of a section of a string from its end. You give the string and ask for the number of characters at the end to return.
right$(“UV-Lamp”,4) returns “Lamp“
right$(“UV-Lamp”,5) returns “-Lamp“

UCASE$- Function: This changes all text in the string to uppercase
UCASE$(“Varian Cary 50”) returns “VARIAN CARY 50”

LCASE$-Function: Converts the string to lower case
UCASE$(“Varian Cary 50”) returns “varian cary 50”

LEN-Function: This returns the number of characters in the string
len(“UV-Lamp”) returns 7
len(“UV”) returns 2

CHR$-Function: This returns the character equivalent for an ASCII number code (see the ASCII table below)
chr$(181) returns µ
chr$(177) returns ±
chr$(186)+”C” returns ºC
chr$(136) returns ‰

ASCII table of characters- font is Courier New (The default font used in reports)

033 !	034 "	035 #	036 %	038 &
037 %	038 &	039 ’	040)	042 *
041)	042 *	043 +	044 -	046 .
045 -	046 .	047 /	048 1	050 2
049 1	050 2	051 3	052 5	054 6
053 5	054 6	055 7	056 9	058 :
057 9	058 :	059 ;	060 =	062 >
061 =	062 >	063 ?	064 A	066 B
065 A	066 B	067 C	068 E	070 F
069 E	070 F	071 G	072 I	074 J
073 I	074 J	075 K	076 M	078 N

077 M	078 N	079 O	080 Q	082 R
081 Q	082 R	083 S	084 U	086 V
085 U	086 V	087 W	088 Y	090 Z
089 Y	090 Z	091 [	092]	094 ^
093]	094 ^	095 _	096 a	098 b
097 a	098 b	099 c	100 e	102 f
101 e	102 f	103 g	104 i	106 j
105 i	106 j	107 k	108 m	110 n
109 m	110 n	111 o	112 q	114 r
113 q	114 r	115 s	116 u	118 v
117 u	118 v	119 w	120 y	122 z
121 y	122 z	123 {	124 }	126 ~
125 }	126 ~	127	128	130 ,
129	130 ,	131 ƒ	132 …	134 †
133 …	134 †	135 ‡	136 ‰	138 Š
137 ‰	138 Š	139 ‹	140	142 Ž
141	142 Ž	143	144 ‘	146 ’
145 ‘	146 ’	147 “	148 •	150 –
149 •	150 –	151 —	152 ™	154 š
153 ™	154 š	155 ›	156	158 ž
157	158 ž	159 Ÿ	160 ¡	162 ¢
161 ¡	162 ¢	163 £	164 ¥	166 ¦
165 ¥	166 ¦	167 §	168 ©	170 ª
169 ©	170 ª	171 «	172 -	174 ®
173 -	174 ®	175 ¯	176 ±	178 ²
177 ±	178 ²	179 ³	180 µ	182 ¶
181 µ	182 ¶	183 ·	184 ¹	186 º
185 ¹	186 º	187 »	188 ½	190 ¾
189 ½	190 ¾	191 ¿	192 Á	194 Â
193 Á	194 Â	195 Ã	196 Å	198 Æ
197 Å	198 Æ	199 Ç	200 É	202 Ê
201 É	202 Ê	203 Ë	204 Í	206 Î
205 Í	206 Î	207 Ï	208 Ñ	210 Ò
209 Ñ	210 Ò	211 Ó	212 Õ	214 Ö
213 Õ	214 Ö	215 ×	216 Ù	218 Ú
217 Ù	218 Ú	219 Û	220 Ý	222 Þ
221 Ý	222 Þ	223 ß	224 á	226 â
225 á	226 â	227 ã	228 å	230 æ
229 å	230 æ	231 ç	232 é	234 ê
233 é	234 ê	235 ë	236 í	238 î
237 í	238 î	239 ï	240 ñ	242 ò
241 ñ	242 ò	243 ó	244 õ	246 ö
245 õ	246 ö	247 ÷	248 ù	250 ú
249 ù	250 ú	251 û	252 ý	254 þ

Appendix D - Database node format specifiers

For the database, the format specifiers are in “developer software language” rather than ADL format. Format specifiers have the following form:

"%" [index ":"] ["-"] [width] ["." prec] type

A format specifier begins with a % character. After the % come the following, in this order:

An optional argument index specifier, [index ":"]
An optional left justification indicator, ["-"]
An optional width specifier, [width]
An optional precision specifier, ["." prec]
The conversion type character, type

The following table summarizes the possible values for type:

d Decimal. The argument must be an integer value. The value is converted to a string of decimal digits. If the format string contains a precision specifier, it indicates that the resulting string must contain at least the specified number of digits; if the value has less digits, the resulting string is left-padded with zeros.

e Scientific. The argument must be a floating-point value. The value is converted to a string of the form "-d.ddd...E+ddd". The resulting string starts with a minus sign if the number is negative. One digit always precedes the decimal point.

The total number of digits in the resulting string (including the one before the decimal point) is given by the precision specifier in the format string--a default precision of 15 is assumed if no precision specifier is present. The "E" exponent character in the resulting string is always followed by a plus or minus sign and at least three digits.

f Fixed. The argument must be a floating-point value. The value is converted to a string of the form "-ddd.ddd...". The resulting string starts with a minus sign if the number is negative.

The number of digits after the decimal point is given by the precision specifier in the format string--a default of 2 decimal digits is assumed if no precision specifier is present.

g General. The argument must be a floating-point value. The value is converted to the shortest possible decimal string using fixed or scientific format. The number of significant digits in the resulting string is given by the precision specifier in the format string--a default precision of 15 is assumed if no precision specifier is present.

Trailing zeros are removed from the resulting string, and a decimal point appears only if necessary. The resulting string uses fixed point format if the number of digits to the left of the decimal point in the value is less than or equal to the specified precision, and if the value is greater than or equal to 0.00001. Otherwise the resulting string uses scientific format.

n Number. The argument must be a floating-point value. The value is converted to a string of the form "-d,ddd,ddd.ddd...". The "n" format corresponds to the "f" format, except that the resulting string contains thousand separators.

m Money. The argument must be a floating-point value. The value is converted to a string that represents a currency amount. The conversion is controlled by the CurrencyString, CurrencyFormat, NegCurrFormat, ThousandSeparator, DecimalSeparator, and CurrencyDecimals global variables, all of which are initialized from the Currency Format in the International section of the Windows Control Panel. If the format string contains a precision specifier, it overrides the value given by the CurrencyDecimals global variable.

p Pointer. The argument must be a pointer value. The value is converted to a string of the form "XXXX:YYYY" where XXXX and YYYY are the segment and offset parts of the pointer expressed as four hexadecimal digits.

s String. The argument must be a character, a string, or a PChar value. The string or character is inserted in place of the format specifier. The precision specifier, if present in the format string, specifies the maximum length of the resulting string. If the argument is a string that is longer than this maximum, the string is truncated.

x Hexadecimal. The argument must be an integer value. The value is converted to a string of hexadecimal digits. If the format string contains a precision specifier, it indicates that the resulting string must contain at least the specified number of digits; if the value has fewer digits, the resulting string is left-padded with zeros.

Appendix E - Comparison DOS/OS2/WinADL

FITF	OS/2	WIN 95
	ABS_TO_T	
		AccWaitNotBusy
	ACOS	
	ACTION	
ACTION_DO		
	ACTIVE_BOX	ActiveGraph
	ACTIVE_SCAN	ActiveTrace
	ADD_BOX	NewGraph
	ADD_ITEM	
	ADD_SCAN	ShowCtm
	ADL_COMPILE	
		AddCtmData
	ADJUST_SCANPARAMS	
ADS_ALIGN	ADS_ALIGN	AlignSps
ADS_MIX	ADS_MIX	MixSps
ADS_PARK	ADS_PARK	ParkSps
ADS_RACKTYPE	ADS_RACKTYPE	
ADS_RINSE	ADS_RINSE	RinseSps
ADS_SAMPLE	ADS_SAMPLE	
	ADS_SENDSTRING	
	ADS_STATUS	StatusSps
ADS_STIR	ADS_STIR	StirSps
	ADS_STIR_HEIGHT	
	ADS_STOP	
	ADS_WIGGLE	WiggleSps
ALARM	ALARM	Alarm
*AND		SaxBasic
	ANTILN	
APPEND	APPEND	Append
	APPEND_ITEM	
APP_PROMPT		
APP_UPDATE_CONC		
ARAMPACC	ARAMPACC	ARampAcc
ARCHIVE	ARCHIVE	Archive
ARCHIVE_REPORT	ARCHIVE_REPORT	ArchiveReport
AREAB	AREAB	AreaB
AREAZ	AREAZ	AreaZ
	ASIN	
	ATAN	
AUTO_RUN		n/a
	AUTO_REPORT	
	AUTO_REPORT_DLG	
AUTO_SCALE	AUTO_SCALE	AutoScale PressAutoScaleBtn
AUTO_STORE		
	AUTO_STORE_NAME	
	AUTO_STORE_REPORT	
AUTODILUTE_SAMPLE		n/a
AUTOSCALE		
	AV	
BASELINE#	BASELINE#	n/a
BEEP	BEEP	SaxBasic
*BEGIN		Dlg fn in SaxBasic
BLANK_STATUS_FIELD		StatusStr
	BOX_CONTROL	
	BREAK	
BUSYACC		BusyAcc
	C_NOTIFYADL	
CALCULATE_CONC		
CALIBRATE	CALIBRATE	CalibrateInst
	CALL	SaxBasic
	CALL_HOOKS	
*CASE		SaxBasic

CHECK_FOR_LABELS		
CHECK_FOR_TAGS		
CHECK_RECAL		
CHECK_USED_CELLS		
		CheckAbort
CHOOSE_NEXT_SAMP LE		
	CHDIR	SaxBasic
CHR$	CHR$	SaxBasic
	CLASS_OF	
CLEANUP_AFTER_ADS _RUN		
	CLEAR_COLLECTED_CTMS	
CLEARABORT		SetAbort
CLEARCATTAGS		
		ClearStatusDisplay
		ClearLines
	CLICK	PressAutoArrangeBtn PressAutoScaleBtn PressCalcBtn PressClearGraphBtn PressClearReportBtn PressFullGraphBtn PressGotoBtn PressPrintBtn PressRecalcBtn PressSetupBtn PressZeroBtn PressZoomOutBtn PressStartBtn PressStopBtn PressAttachBtn PressUserdataFormBtn PressPeakLabelBtn
	CLOSE	Quit
	CNC_CLOSE_DOWN	
	CNC_PRE_CALC	
	CNC_PRE_READ	
	CNC_PRE_SEQUENCE	
	CNC_POST_READ	
	CNC_POST_SEQUENCE	
	CNC_POST_USER_BUTTON	
	CNC_STARTUP	
	CNC_USER_BUTTON	
	CNC_USER_ZERO	
COLLECT	COLLECT	StartInst,Collect
COLLECT#	COLLECT#	n/a
COLOUROPTIONS		
COMMAND		n/a (SaxBasic uses)
COMMENCE_CELL_CO LLECT		
COMMKEY_ACTION		
	COMPILE	
	COMPILE_AND_SAVE_WS	
	COMPILE_APPLICATION_C ODE	
COMPILE_CALC		
COMPILE_INIT_ARUN		
COMPILE_RUN_FLUSH		
	CONVERTSCAN	Fconvert
CONDITIONACC	CONDITIONACC	ConditionAcc
COPY#		
	COPY_FROM_CLIPBOARD	ReportCopyToClipBoar d,ReportPasteFromClip Board,GraphCopyToCli pBoard,GraphPasteFro mClipBoard
	COPY_TO_CLIPBOARD	
COS	COS	SaxBasic
CREATECURVE	CREATECURVE	CreateCurve

	CREATE_STDPLOT	
		SetCtmXLabel
		SetCtmYLabel
CURRENT#		n/a
CURSOR		PressCursorBtn
CURSOR_DRIVE		
CURVEFIT	CURVEFIT	CurveFit
CURVEFITDATA	CURVEFITDATA	CurveFitData
DATA_VIEW		
	DB_ABSCISSA	
	DB_ADD_PARAM	AddNode
	DB_EDIT_PARAM	
	DB_FORMAT	SetNodeFormat
	DB_INFO	
	DB_ONLINE	
	DB_ORDINATE	
	DB_PARAMS	
	DB_RUN_DLG	
	DB_TEXT	
	DE_ENCRYPT_ADL	
*DEFINE		Sax Basic Sub
	DEL_USER	
DEPOSIT	DEPOSIT	SetVal
DEPOSIT_ATTR		
DEPOSIT_FIELD		
DEPOSIT_LIMS	DEPOSIT_LIMS	SetNodeLimits
DEPOSIT_STR		SetVal
DEPOSIT_SYSIND		SetVal
DEPOSITCHOICE		SetVal
DERIVn#	DERIVn#	Deriv
DILUTE_IF_REQUIRED		
DILUTOR_RACK		n/a
*DIM		SaxBasic
	DIR	SaxBasic Dir$
	DIRC	
	DIRECT_PRINT_FILE	
	DISPLAY	
DISPLAY_AND_GO_COMMAND		
DISPLAY_COMMAND		
DISPLAY_GRAPH		
DISPLAY_INDEX		
	DISPLAY_OPTIONS	
	DISPLAY_OPTIONS_BASE	
DISPLAY_PAGE		
DISPLAY_TABLE		
DISPLAY_TEXT		
*DO		
DOS		Shell
DO_ADS_BLANK_AND_STANDARDS		
DO_AUTO_REPORT		
DO_BASELINE		DoBaseline
DO_CALIBRATE_STANDARDS		
DO_PAGE_ENTRY		
DO_PAGE_EXIT		
DO_PROCESS_STANDARDS		
DO_STORE_ACTIONS		
DRIVEACC	DRIVEACC	DriveAcc
DUMP_GRAPHICS		
DUMP_GRAPHICS_TIME&TEMP		
DUMPSCR	DUMPSCR	
EDIT		
*ELSE		SaxBasic
EMPTY_RSA	EMPTY_RSA	EmptyRsa

*END		SaxBasic End (end Sub)
*ENDDEF		
	END_COLLECT	
END_MULTI_CELL_COL LECT		
ENTER_PAGE		
ERASEDISPLAY		
ERROR		
ERRORSTR	ERRORSTR	ErrorStr
	ERRPRINT	
	EVAL	
EXECUTE	EXECUTE	Execute
	EXECUTEC	
EXIT	EXIT	SaxBasic - Quit
EXP	EXP	SaxBasic
EXPERTOPERATIONS		
EXTRACT	EXTRACT	Extract
F9_COLLECT		n/a
FCLOSE	FCLOSE	SaxBasic Close
FCONVERT		Fconvert
FEND		
FIELDEXIT		
FILE_CURRENT_AND_ TAG		
	FILEIN	
	FILEINC	
FILL_RSA	FILL_RSA	FillRsa
FINISH_AUTO_RUN		
	FIND_FILES	SaxBasic Dir
	FIND_FILESC	
	FINDX	
FINPUT		Sax Basic
	FLAT	
FLUSHLAST		
FOPEN	FOPEN	SaxBasic Open
	FOPENC	
*FOR		Sax Basic
	FORCENAME	
	FORMAT	Sax Basic Format$
FORMAT_DATA_VAL		n/a
FORMSACTION		
FPRINT	FPRINT	Sax Basic Print
	FRAME_CONTROLS	
	FRAME_DISPLAY	
	FRAME_MENU	
	FRAME_RUN	
	FREAD	Sax Basic Input Sax Basic Input$
	FREADB	
	FREADLN	Sax Basic Line Input
FREECODE		
FREEDATA		
FREEDICTIONARY		
	FSEEK	Sax Basic Seek
	FWRITE	Sax Basic Print Sax Basic Write
	FWRITEB	
	FWRITELN	
	GARBAGE_COLLECT	
GET	GET	GetVal
GET_ACTFACT		
GET_ADS_TABLE_RO W_VARS		
	GET_APP_PARAM	
GET_CELL_FACTOR		
GET_COLLECT_CHOIC		

E		
GET_LABEL_AND_COR RS		
	GET_MONITORED_CHAN	
GET_NOMINAL		
GET_PAGE		
	GET_PROP	
	GET_RUN_PATH	
GET_SINGLE_RESULT		
GET_TUBES_PER_RAC K		n/a
	GET_USER	
GET_X1_X2		
GET_X_INC		
GET_ZPLANE@COLOU R		
		GetAcc
GETABSMODE		GetAbsMode
GETDATE		SaxBasic Date
		GetFormatted
		GetNodeIndex
GETORDMODE		GetOrdMode
	GETSEQ	
		GetSps
		GetValByIndex
GETVARS		
GO_GRAPHICS_PAGE_ IF_REQD		
		GoBusyAcc
	GOTO	
	GOTO_SAMPLE	MoveToSps
		GoToCellAcc
GOTO_PAGE		
GOTO_REQUIRED_TE MP		
GOTO_SAMPLE		
GOTO_TEMP		
GOTOXY		
GRAPHICS	GRAPHICS	
GRAPH_CLEAR	GRAPH_CLEAR	GraphClear
	HANDLE_INTV_GET	
	HANDLE_NIRINTV_GET	
	HANDLE_XMAX_GET	
	HANDLE_XMIN_GET	
	HANDLE_INTV_SET	
	HANDLE_NIRINTV_SET	
	HANDLE_NIRRATE_SET	
	HANDLE_NIRSAT_SET	
	HANDLE_RATE_SET	
	HANDLE_SAT_SET	
	HANDLE_XMAX_SET	
	HANDLE_XMIN_SET	
HEADER	HEADER	Header
HEADERNAME		
HEADERSTR		HeaderStr
HPLOT#		
HPLOTAXIS		
HPLOTCLOSE		
HPLOTSTART		
HPXAXISLABEL		
HPYAXISLABEL		
	HW	
*IF		Sax Basic If
	INCLUDES_ITEM	
	INIT_ADS_VARS	
INIT_ARUN		
INIT_MULTI_CELL_TEM P		

INIT_PIPE_DRAW		DrawLine
	INNER_RECT	
INPUT	INPUT	Cinput,Vinput
INPUTCR	INPUTCR	
INPUTESC	INPUTESC	
	INSPECT	
	INST_BUSY	BusyInst
	INST_STATUS	AddStatusEntry
		InstGet
		InstSet
		InstWaitNotBusy
		InstWaitNotScanning
INT$	INT$	Sax Basic InStr
IS_DILUTION_RACK_THERE		
	IS_BYTES	
	IS_INDEXED	
	IS_KIND_OF	Sax Basic IsNum etc
	IS_OPEN	
	KEY_AT	
KEYOP		
KEYPRESSED		KeyCode
	KEYS	
KEYTYPE		
	KM	
LABEL	LABEL	
	LAMPS_OFF	
	LAMPS_ON	
LEN	LEN	Sax Basic
LIMITACC	LIMITACC	LimitAcc
	LIMITSEQ	
	LINE_DRAW	DrawLine
	LIST_APPEND	
	LIST_FIELD	
	LIST_FILLER	
	LIST_USER_STATUS	
LN	LN	Log
	LOADFILE	
	LOADFILEC	
LOADHEADER	LOADHEADER	LoadHeader
LOADMETHOD	LOADMETHOD	OpenAs
	LOAD_WORK_FILE	
LOG	LOG	Log10
	LONG_AT	
	LONG_TO_C	
LPRINT	LPRINT	Lprint
MAIN_ARUN		
	MAKE_CTM_NAME	
	MAKE_USER	
	MAX	
MEASUREACC	MEASUREACC	MeasureAcc
	MENU_ATTRIB	
	MENU_REMOVE	
MID#	MID#	MidC
MID$	MID$	Sax Basic
	MIN	UVMin
*MINUTES		n/a
	MKDIR	Sax Basic
	MLEPRINT	
		MonitorAcc
MOVE_PROBE_TO_POSITION		MoveToSps
	MSG_BOX	SaxBasic MsgBox
MULTI_CELL_TEMP_COLLECT		
MULTI_CELL_TIME_COLLECT		
MULTI_ZERO		PressZeroBtn

	NAMED_MEASUREACC	
NATURAL_EXP	NATURAL_EXP	Power
	NEW	
	NEW_SIZED	
		NewCtm
*NEXT		Sax Basic
NORMAL_AUTO_RUN		
NORMAL_COLLECT	NORMAL_COLLECT	n/a
	NORMALIZE	
*NOT		Sax Basic
	NUM_BOXES	NumGraphs
	NUM_SCANS	NumTraces
	ONLINE_DB	
*OR		Sax Basic
PAGE		
PAGE_CHANGE		
PAGE_SWAP_INIT		
PAGE_TOGGLE		
PARK_ADS_PROBE		
PAGEEXIT		
PIPEDRAW		
PLOT#	PLOT#	Plot
	PLOT_CONTROL	
PLOT_EXACT#		
PLOT_SYMBOL	PLOT_SYMBOL	PlotSymbol
PLOTOFF	PLOTOFF	
PLOTON	PLOTON	
POST_LOAD_METHOD		
		PositionGraph
PREVIOUS#		n/a
PRE_PAGE_ENTRY		
		PressAutoArrangeButto n
	PREVIOUS_READ	
PRINT	PRINT	Cprint
	PRINT_GRAPH	
	PRINT_REPORT	
PRINT_WEIGHT_VOL_ REP		
	PRINTER_SELECTION	
PRINTFILE	PRINTFILE	
PRINTHEADER	PRINTHEADER	PrintHeader
PROGRAM		
	PRUNE_CONSTS	
	PTS	
	PUT_LONG_AT	
	PUT_WORD_AT	
PUTVARS		
QUICK_STOP		
	RAISE_ERROR	Sax Basic Error
RAMP_DISTANCE	RAMP_DISTANCE	RampDistance
RAMP_TEMP	RAMP_TEMP	RampTemp
RAMP_VASRA	RAMP_VASRA	RampVasra
RAMPACC	RAMPACC	RampAcc
READ	READ	Read
	REAL_TO_C	
READ_TEMP	READ_TEMP	ReadTemp
RECALL	RECALL	GetVal
RECALL(TAG_DATA)		
RECALL_FIELD		
RECALL_FORMATTED		GetFormatted
RECALL_HIGH_LIM		GetNodeLimits
RECALL_IND		GetVal
RECALL_LOW_LIM		GetNodeLimits
RECALL_PROMPT		
RECALL_STR		GetVal
RECALL_SYSIND		GetVal
RECALL_XY_USER_RE		

S		
RECALLCHOICE		GetVal
RECALLCHOICE_IND		GetVal
RECOVER		
	RECT_BOTTOM	
	RECT_HEIGHT	
	RECT_LEFT	
	RECT_RIGHT	
	RECT_TOP	
	RECT_WIDTH	
	REMOVE_BOX	RemoveGraph
	REMOVE_DISPLAYED_CTM S	
	REMOVE_ITEM	
	REMOVE_SCAN	RemoveCtm
	REMOVE_SPACES	
	REMOVEHOURGLASS	
	REMP_PROP	
*REPEAT		
REPLICATE_LOAD		
REPORT_REPLICATES		
		ReportAddText
RESET_ADS		ResetSPS
RESET_SLIDE	RESET_SLIDE	ResetSlide
RESET_SOLUTION_NO		
RESET_VASRA	RESET_VASRA	ResetVasra
	RESETINST	
	RESET_GOTORANGE	
	RESIZE	SizeGraph
RESULT#		n/a
RETRIEVE	RETRIEVE	
	RETRY_ERROR	
RETURNKEY		
RINSE_TEST		
	RMDIR	Sax Basic
	ROUND	Sax Basic Rnd
	RUN_DLG	
	RUN_DLG_2	
	RUNFUNCTION	
SAVE		
	SAVE_FILE	SaveAs
	SAVE_FILEC	
SAVEMETHOD	SAVEMETHOD	
	SAVE_REPORT	ArchiveReport
	SAVE_WORK	
	SAVE_WS	
	SCAN#	
	SCAN_IN_BOX	
		ScanningInst
SCREEN_DUMP		
	SCROLL_MAX	
	SCROLL_MIN	
	SCROLL_REVERSE	
	SCROLL_SET	
	SD	
*SECONDS		n/a
	SELECT_DOS_CTM	
	SELECT_FILE	ShowOpenDlg
	SELECT_FILEC	
SELECT_RACK_AND_T UBE		
	SELECT_USER	
		SelectGraph
		SelectTrace
SET	SET	SetVal
	SET_BMAP_SIZE	
SET_CELL_NO		
SET_CURRENT_PAGE		

	SET_HOOK	
	SET_PLOT_FLAG	
	SET_PLOT_FLAG_ADL_VAR	
	SET_PROP	
	SET_RPG_DATABASE	
SET_SOLUTION_NO		
	SET_SYMBOL	
	SET_SYMBOL_SIZE	
	SET_USER	
SET_USER_STATUS	SET_USER_STATUS	
SETABORT		SetAbort
SETACC	SETACC	SetAcc
SETCOLOR		
SETCURRHEAD		
SETHEADER	SETHEADER	SetHeader
SETHEADERSTR		SetHeaderStr
SETRBA	SETRBA	SetRba
	SETSEQ	
		SetSps
SETUP		
SETUP_PLOT_AXES	SETUP_PLOT_AXES	
SETUP_SN_NO		
SETUPINST	SETUPINST	SetupInst
		SetValByIndex
	SGSMOOTH	Smooth
	SHORTEN	
	SHOW_DEBUG	Debugger
		ShowCtm
	SHOWHOURGLASS	
SHOWREPORT		
	SHOW_WINDOW	ShowAddPictureDlg ShowAddTextDlg ShowAxesScalesDlg ShowCalculatorDlg ShowCursorDlg ShowGraphPreferences Dlg ShowOpenDlg ShowSaveDlg ShowTracePreferences Dlg ShowScanParamsDlg ShowPeakInfoDlg ShowUserDataFormDlg ShowNodeDlg
SHUT_DOWN		
SIN	SIN	Sax Basic
SIREN	SIREN	
	SIZEOF	
SLOPE	SLOPE	
	SMOOTH	Smooth
SMOOTHn#	SMOOTHn	Smooth
SN#		n/a
SN_COLLECT		n/a
SN_COLLECT_INIT		
SN_COLLECT_REPORT ING		
SPAWN_PROG	SPAWN_PROG	Shell
		SpsBusy
		SpsWaitNotBusy
SQRT	SQRT	Sqr
	START#	
	START_COLLECT	StartInst
	START_CONTROL	
STARTCYCLETIMER		
STARTEXTERN		n/a
STARTPRINT		StartPrint
START_UP		
STATUS	STATUS	

	STATUSCOLOR	
		StatusHardware
		StatusResult
		StastusSequence
STATUSSTR	STATUSSTR	StatusStr
*STEP		Sax Basic
STEP_KEY		
	STRING_TO_C	
STOP	STOP	
STOPACC	STOPACC	StopAcc
STOPEXTERN		n/a
STOPINST		StopInst
STOPPRESSED		StopPressed
	STOPSEQ	
STOP_SN_COLLECT		
STR$	STR$	Str$
	SUM	
	SUMMARIZE_ERROR	
	SUMSQRDIF	
	T_TO_ABS	
	T_TO_PERCENT	
TABLE	TABLE	Table
TABLEDATA	TABLEDATA	TableData
	TABLEDEST	
TABLEFMT	TABLEFMT	TableFmt
TABLEHDR	TABLEHDR	TableHdr
	TAN	Sax Basic
	TEM_GET_ABS	
	TEM_GET_VAL	
	TEM_GOTO	
	TEM_RAMP	
	TEM_WAIT	
TEMP_OFF		
TEMP_ON		
TEMP_PRINT		
	TEXT	
*THEN		Sax Basic
	TIME	Sax Basic
	TIME_INFO	
	TIMER_START	
TIME_START_END		
TIME#		n/a
	TIMER_START	
	TIMER_STOP	
	TIME_WAIT	Sax Basic Wait
*TO		Sax Basic
	TRACE_OFF	
	TRACE_ON	
TURNSTATUSOFF		n/a
TURNSTATUSON		n/a
	TYPECHECK	
*UNTIL		
	UNLIST_USER_STATUS	
UPDATECOLOR		
UPDATE_CALC		
UPDATE_CURR_ABS		
UPDATE_DATA		
UPDATE_SCREEN		
UPDATE_STATUS		
USER1$		
USER2$		
USER_COLLECT		n/a
	USER_ERROR	
USER_RESULT	USER_RESULT	UserResult
	USER_STARTUP	
	USER_STOP	
	UV_ARCHIVE	
	UV_CHECK_ABS_MODE	

	UV_CHECK_ABS_MODES_ MATCH	
	UV_CLEAR_ALL	
	UV_CLEAR_REPORT	
	UV_CREATE_BUTTON	
	UV_CREATE_CHECKBOX	
	UV_CREATE_CHOICEBOX	
	UV_DELETE_CARY_FILES	
	UV_CREATE_EDITOR	
	UV_CREATE_EDITOR_REA DONLY	
	UV_CREATE_ENTRYFIELD	
	UV_CREATE_GROUPBOX	
	UV_CREATE_LIST_BOX	
	UV_CREATE_PLOT_WINDO W	
	UV_CREATE_RADIO_BUTT ON	
	UV_CREATE_STATIC_FIEL D	
	UV_ERROR_PROCESSOR	
	UV_GET_INST_ORD_STRIN G	
	UV_GET_LONG_STRING	
	UV_GET_UV- SCAN_PARAMETERS	
	UV_GET_WAVELENGTH_R ANGE	
	UV_MAKE_ENTRYFIELD_R EADONLY	
	UV_MAKE_STRETCHABLE	
	UV_PRINT_REPORT	
	UV_RESET_STATUS_SETTI NGS	
	UV_RETRIEVE_FILES_INTO NE_BOX	
	UV_RETRIEVE_WAVELENG TH_SCANS	
	UV_SAVE_STATUS_SETTIN GS	
	UV_SET_LONG_STRING	
	UV_SET_UV_SCAN_PARAM ETERS	
	UV_TABLEHDR_	
	UV_TIMESTAMP	
	UV_UNPACK_CARY_NAME	
	UV_UPDATE_SCAN_INFO	
VAL	VAL	Val
	VALUE_AT	
	VALUES	
	VT__ACCESSORY_EVENT	
	VT__ACCESSORY_RESPON SE	
	VT__ADL_STARTUP	
	VT__AUTO_REPORT_SCAN	
	VT__BULID_ADVANCED_TO OLS	
	VT__BULID_BASE_CLASSE S	
	VT__BUILD_BASE_FUNCTI ONS	
	VT__BUILD_BASE_RUNTIM E	
	VT__BUILD_BASE_TOOLS	
	VT__BUILD_CARY45	
	VT__BUILD_CARY45_APPS	
	VT__BUILD_CARY45_BASE	
	VT__BUS_ACTION	
	VT__C45_STARTUP	

	VT__C45_STARTUP_TIMER_ACTION	
	VT__C45_STOP_ACTION	
	VT__CALL_HOOKS	
	VT__CFM_ACTION	
	VT__COLLECT_ENDED	
	VT__COLLECT_STARTED	
	VT__CREATE_STDPLOT	
	VT__DEBUG	
	VT__ENSURE_UNIQUE_ABSCISSA	
	VT__FLUSH_AUTO_REPORT	
	VT__HANDLE_ACY_DISPLAY	
	VT__HANDLE_DSB	
	VT__INIT_AUTO_REPORT	
	VT__INT_ACTION	
	VT__INT_MENU	
	VT__INT_MENU_ATTR	
	VT__LOAD_WORK	
	VT__PLOTOFF_COLL_END	
	VT__PLOTOFF_COLL_START	
	VT__PLOTOFF_SEQ_END	
	VT__PLT_ACTION	
	VT__RAISE_ADL_ERROR	
	VT__RAISE_APP_ERROR	
	VT__RESTORE_APP_STATE	
	VT__SAVE_APP_STATE	
	VT__SEQUENCE_ENDED	
	VT__SEQUENCE_STARTED	
	VT__SETUP_BOX_ARRAY	
	VT__SIREN_BUTTON_ACTION	
	VT__SIREN_TIMER_ACTION	
	VT__STARTUP	
	VT__STDOUT_FRAME_ACTION	
	VT__STOP	
	VT__SUMMARIZE_ERROR_INFO	
	WAITENDCOLLECT	
WAITENDCYCLE		
WAITENDCYCLETIME		
WAITENDTEMP		
*WHILE		Sax Basic
	WIN_CENTRE	
	WIN_CLOSE	
	WIN_CLR	
	WIN_ENABLE	
	WIN_ID	
	WIN_FONT	
	WIN_HEIGHT	
	WIN_PARENT	
	WIN_POSN	
	WIN_RECT	
	WIN_RSRC	
	WIN_SHOW	
	WIN_STRETCH	
	WIN_STYLE	
	WIN_TEXT	
	WIN_VALUE	
	WIN_WIDTH	
	WN	
	WORD_AT	
WT_VOL_CORRECT		

	ZERO_BUTTON	
ZEROINST	ZEROINST	ZeroInst
ZEROTEST	ZEROTEST	ZeroTest
ZERO_READ		
ZOOM		PressZoomOutButton
[		
]		
{		
}		
#		
$		
%QUOTE%		
(		
)		
*		
+		CtmOp
,		MatOp
-		
.		
/		
:		
<		
<=		
<>		
=		
>		
>=		

Appendix F - Technical references on VBA

The VBA Developer's Handbook

Authors: Ken Getz and Mike Gilbert
Publisher: Publisher: Sybex Inc.
ISBN: 0-7821-1951-4

Beginning Access 95 VBA Programming

Focusing on a sample application (provided), it explains the concepts and techniques you'll need to come to grips with VBA.
Authors: Robert Smith, David Sussman
Publisher: Wrox Press
ISBN: 1874416648

Microsoft Office & Visual Basic for Applications

Developer
Each month Microsoft Office & Visual Basic for Applications Developer contains advanced how-to articles and tips columns on developing Office and Office-compatible applications using VBA.

And Microsoft Visual Basic

Visual Basic Programmer's Journal

Visual Basic Programmer's Journal is the world's leading development magazine, delivering critical, hands-on articles for programmers using Visual Basic and other VB tools. VBPJ is published monthly by Fawcette Technical Publications.

Inside Visual Basic

The Cobb Group's monthly journal of tips and techniques for Visual Basic. Free trial copy available

Mastering Visual Basic 5

Beginners will gain a firm footing in Visual Basic with a complete introduction to the language and IDE. Others will take their skills to the next level with coverage of a wide range of increasingly advanced topics presented through scores of practical examples.
Author: Evangelous Petroutsos
Publisher: SYBEX Inc.
ISBN: 0-7821-1984-0

Visual Basic 5 from the Ground Up

Takes you from elementary programming skills to state-of-the-art techniques that give you the professional edge.
Author: Gary Cornell
Publisher: Osborne/McGraw-Hill
ISBN: 0-07-882349-8

Visual Basic Programmer's Reference

The vital VB, VBScript, and VBA information you need - all

in one handy reference.
Author: Dan Rahmel>
Publisher: Osborne/McGraw-Hill
ISBN: 0-07-882458-3

Appendix G - Accessory Codes

The following information details the capabilities of the Accessory Controller Board which is optional for the Cary 1, 3, 4, 5, 100, 300, 400 and 500 instruments.
The first table lists the codes that various Cary accessories use. The second table contains the details of the full functionality of the Accessory Controller Board.
The following abbreviations and codes are used:
RSA= Routine Sampler Accessory
PL3= 9 pin socket on the Accessory Controller Board inside the sample compartment
PL4= 15 pin socket on the Accessory Controller Board inside the sample compartment
PL5= 25 pin socket on the Accessory Controller Board on the outside of the instrument
PL6= 25 pin socket on the Accessory Controller Board inside the sample compartment
anlg = analog
dig=digital
i/p=input
o/p=output
Deg C= degrees Celsius °C
MV=millivolts
MM = millimetres
stpr=stepper motor
gnd = ground

Device code	**Use**	**Available on Cary 50***
0	Sample transport (Steps per mm = 15.75 Cary 400/500, 9.84375 Cary 100/300, 16.667 Cary 50, MM per cell = 14 Cary 400/500, 11 Cary 100/300, 10.5 Cary 50)	Yes
2	RBA(1.8 degrees per step), Vasra (2 degrees per step)	No
16	Block temperature (RSA, Temperature controller, 100 counts per degree)	No
32	Set temperature (RSA, Temperature controller, 100 counts per degree)	No
48	RSA empty button (0=pushed)	No
49	RSA fill button (0=pushed)	No
52	Remote Read Switch	Yes
18	Temperature probe 2	Yes
19	Temperature probe 1	Yes
20	Temperature probe -external connection probe 2	No
21	Temperature probe -external connection probe 1	No

- all of these devices are available on the Cary 1, 3, 4, 5, 10, 300, 400 and 500 instruments, the Cary 50 has only some of the devices.

Device Description	Hardware connection	Physical range	ADL device code	Software range
Stpr0 dig i/o	PL4-4,9,2,10,3	0 to +10000. steps	0	0 to +10000. steps
Stpr1 dig i/o	PL4-13,14,7,15,8	“”	1	“”
Stpr2 dig i/o	PL3-1,6,2,7,3	“”	2	“”
Stpr0 opto	PL4-5	0, 1=home		
Stpr1 opto	PL4-13	“”		
Stpr2 opto	PL3-5	1, 0=home		
Anlgin0 anlg i/p	PL6-14	-10.240 to +10.235 V	16	-10240 to +10235 mV
Anlgin1 anlg i/p	PL6-2	“”	17	“”
Anlgin2 anlg i/p	PL6-15	“”	18	“”
Anlgin3 anlg i/p	PL6-3	“”	19	“”
Anlgin4 anlg i/p	PL5-2	“”	20	“”
Anlgin5 anlg i/p	PL5-3	“”	21	“”
Anlgin6 anlg i/p	PL5-4	“”	22	“”
Anlgin7 anlg i/p	PL5-5	“”	23	“”
Anlgout0 anlg o/p	PL6-5	-10.240 to +10.235 V	32	-10240 to +10235 mV
Anlgout1 anlg o/p	PL6-18	“”	33	“”
Anlgout2 anlg o/p	PL6-6,PL5-10	“”	34	“”
Anlgout3 anlg o/p	PL5-9	“”	35	“”
Digin0 dig i/p	PL6-22 (RSA Empty)	open, gnd	48	=0 gnd, =1 open
Digin1 dig i/p	PL6-9 (RSA Fill)	“”	49	“”
Digin2 dig i/p	PL6-21	“”	50	“”
Digin3 dig i/p	PL11-21	“”	51	“”
Digin4 dig i/p	PL11-9 (remote read switch, use with PL11-19)	“”	52	“”
Digout0 dig o/p	PL5-13	5v, 0v	64	
Digout1 dig o/p	PL5-14	“”	65	
Digout2 dig o/p	PL5-15	“”	66	
Digout3 dig o/p	PL5-11	“”	67	
Digout4 dig o/p	n.c.	“”	68	
Digout5 dig o/p	PL6-6	“”	69	
Digout6 dig o/p	PL6-20	“”	70	
Digout7 dig o/p	PL6-7	“”	71	
DC motor dc o/p	PL5-19,20 PL6-10, 23	REV,SHORT,FWD	80	=0 SHORT, <0 REV, >0 FWD
AC motor ac o/p	PL5-23,24 PL6-12, 25	REV,OFF,FWD	81	=0 OFF, <0 REV, >0 FWD

Digital output		
stepper ramping	-9449 to +9449 STEPS/MIN	
ramping	-5.50 to +5.50 steps/DPA	-5500 to +5500 STEPS/1000_DPA
Analog output		
ramping	-1.000 to +1.000 mV/sec	-10000 to + 10000 MV*10/SEC
ramping	-33.33 to + 33.33 mV/DPA	-3333 to + 3333 MV/100_DPA

Appendix H – Cary Instrument Database Controls

Primary instrument controls using the database

The following database nodes are the lowest level control for the Cary instrument. Examination of these using GetNodeLimits will provide information about available ranges for relevant models. Also the ADL Quick reference (and Appendix A) contains the constants for all choices, e.g. Beam modes of Beam_Mode_Single_Front, Beam_Mode_Double_Fixed_Slit etc.

"Beam mode"
"Gain"
"UVVIS reference level"
"NIR reference level"
"UVVIS slit width"
"NIR slit width"
"UVVIS interval" (uses negative range)
"NIR interval" (uses negative range)
"UVVIS averaging"
"NIR averaging"
"Slit height"
"PGA Gain indices front"
"PGA Gain indices rear"
"PGA Gain indices dark"
"PGA mode"
"Grating changeover"
"Grating"
"Goto wavelength"
"Source changeover"
"Source"
"Source on/off UV"
"Source on/off Vis"
"Source on/off third"
"Detector changeover"
"Detector"
"Filter"
"Shutters open/closed front"
"Shutters open/closed rear"
"Target SNR"
"UVVIS wavenumber interval"
"NIR wavenumber interval"
"Scan start"
"Scan stop"
"Scan mode"

Set these controls and follow with a SetupInst command to set Cary to the desired locations. Note that where the control has no hardware, setting it has no effect. E.g. Cary 50 has no slit control so setting the value has no effect.
The items mentioned such as "Absolute UVVIS Interval" , "Scan Rate" are application interface specific entries and as such have no direct effect on the instrument. Note that "Scan Rate" is a combination of the Scan Interval ("UVVIS Interval", "NIR Interval") and the averaging time ("UVVIS averaging", "NIR averaging"). This is only an approximation to the speed of scanning.

Note that the collection uses the "Scan Start", "Scan Stop" and "Scan Mode" items to operate with the "Collect" and StartInst commands.
Note that there is a translation step between "X Mode" and "Scan Mode" for user interfacing. Please use "X Mode" with the "Collect" function and use the previously mentioned xxx_XMode constants.

Note that the Interval nodes have negative values. That is because Cary instruments scan wavelengths from high to low. Therefore the change from one wavelength to the next is negative.

Secondary instrument controls

(these update the primary controls automatically)
"Common Slit Width" - updates "UVVIS Slit Width" and "NIR Slit Width"
"Common Interval" updates "UVVIS Interval" and "NIR Interval"
"Common SAT" updates "UVVIS Averaging" and "NIR Averaging"
"Common Energy" Updates "UVVIS Energy" and "NIR Energy"
"UVVIS Energy" and "NIR Energy" update "Gain"

These database items allow Cary 500 particularly to find a common range for entry values that suits both its UVVIS and NIR gratings, accounting for the 4 to 1 dispersion factor between these gratings. If you need independent control for Cary 500 then use the primary controls.

Appendix I - Continuum Description Overview

We created a special data storage unit called a continuum because it gives us great power and flexibility to manipulate data collected from the Cary instrument. It is ideally suited to spectroscopy. As you are no doubt aware we can turn the raw data into Absorbance data in one simple logarithm operation. In other instrument systems each piece of data must be processed individually to produce the complete scan. Therefore they need to be fully aware of the detail of how the data was collected. In our system you don't need any details of how it was collected. Simply process it as you wish. If the "X" position of the data happens to lie between two collected data points then we will automatically interpolate between the data points to give you the expected value at the point you desire.

Internal Structure of Continua

For your information the data in a continuum is divided into two discrete sections. There is a header section that describes relevant information as to how and when the data was collected or has been processed as well as your description of the data and changes. Then there is the actual data storage section.

Internal Structure of Continua

The 30 numbered "slots" which were provided by DOS continua have been extended in WinADL with up to 50 slots available for general storage of numbers still accessed via the HEADER and SETHEADER commands. As well we can also use the audit log as a storage area accessed via the HEADERSTR and SETHEADERSTR operations. The SETHEADER command allows you set any individual slot with a number. The HEADER command allows you to extract any individual number. The slots that are allocated from DOS days are as follows

Xmode	= 0	in Winuv Ymode = 1
in Winuv Cycles	= 2	not in Winuv
SAT	= 3	in Winuv
Slit width	= 4	in Winuv
Interval	= 5	in Winuv
Scan rate	= 6	in Winuv
Steps per cell	= 7	in Winuv
Baseline mode	= 8	in Winuv
Xmin	= 9	in Winuv
XMax	= 10	in Winuv
Reserved 1	= 11	
Reserved 2	= 12	
Reserved 3	= 13	
Reserved 4	= 14	
Reserved 5	= 15	
Reserved 6	= 16	
Detector change wl	= 17	in Winuv
SNR mode	= 18	in Winuv
Target SNR	= 19	in Winuv
Slit height	= 20	in Winuv
Independent NIR	= 21	not in Winuv
NIR slit	= 22	in Winuv
NIR SAT	= 23	in Winuv
NIR interval	= 24	in Winuv
NIR scan rate	= 25	in Winuv
Method number	= 26	not in Winuv
Factor	= 27	not in Winuv
Beam mode	= 28	in Winuv
Beam interchange	= 29	in Winuv

Internal Structure of Continua

The Data

The data section contains its own information describing the starting "X" value, the number of data points in this section, and the distance between data points. This data grouping is called a "region". Where the data has a different spacing from another area or is disjoint in the "X" direction and is still part of the same collection of data we internally represent it as a separate region. Where the data is not part of the same collection but is related to the overall process of data handling we would save the data as separate arrays of collections. We allow for many arrays of data collections and many regions of data within collections. Our only limit is memory. More information about the continuum structure may be permitted upon request

Appendix J – Classes and Objects (examples)

Classes

It is possible to make a Cary object with all the details about interactions, SETUPINST commands and others to be hidden from your general program. However the notation used may be confusing to traditional ADL programmers. The following example is in the ADL help. It illustrates how to make a simple Class that allows us to open and read a text file.

Firstly we create a class file/module. To do this we follow the instructions given as the first few comments. Choose File…New Module…Class Module
Then Edit…Properties…Name=File.
Then the rest of the text.

```
'File.CLS'File|New Module|Class Module'Edit|Properties|Name=FileOption ExplicitDim FN As
IntegerPublic Sub Attach(FileName As String)    FN = FreeFile    Open FileName For Input As #FN
End Sub
Public Sub Detach()
   If FN <> 0 Then Close #FN
   FN = 0
End Sub
Public Function ReadLine() As String
   Line Input #FN,ReadLine
End Function

Private Sub Class_Initialize()
   Debug.Print "Class_Initialize"
End Sub

Private Sub Class_Terminate()
   Debug.Print "Class_Terminate"
   Detach
End Sub
```

The key points to note are the two private Subs Class_Initialize and Class_Terminate. These are required to allow your class to integrate successfully. The other three public Subs are your own routines that will be accessed from the class.
Think of these Subs as Children residing inside the class. To find them we must identify the class then the children.

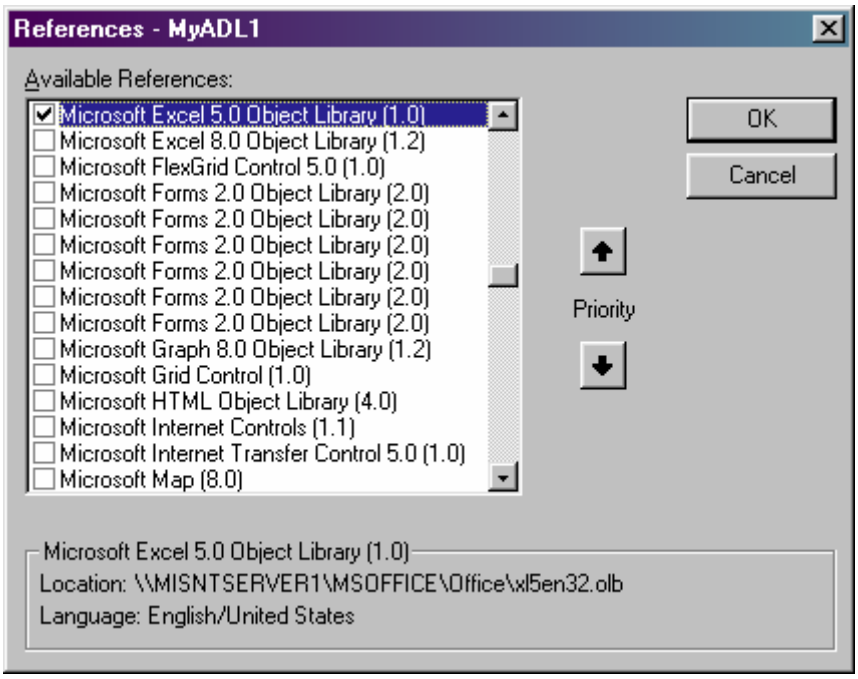
The use of this new class module is shown in the next piece of code. There is a USES clause to allow your second module to know that it will reference your description of the class. To use it we need to make a new variable (called File) which is a complete class having all the properties listed in the Class module. The line "Dim File As New File " does this bit.
Then we use our new variable with what is termed DOT notation. I.e. to get to the Attach Sub we previously created we use File.Attach. and to read from the file we use File.ReadLn.

```
'A.WWB
'#Uses "File.CLS"
Sub Main
   Dim File As New File
   File.Attach "C:\AUTOEXEC.BAT"
```

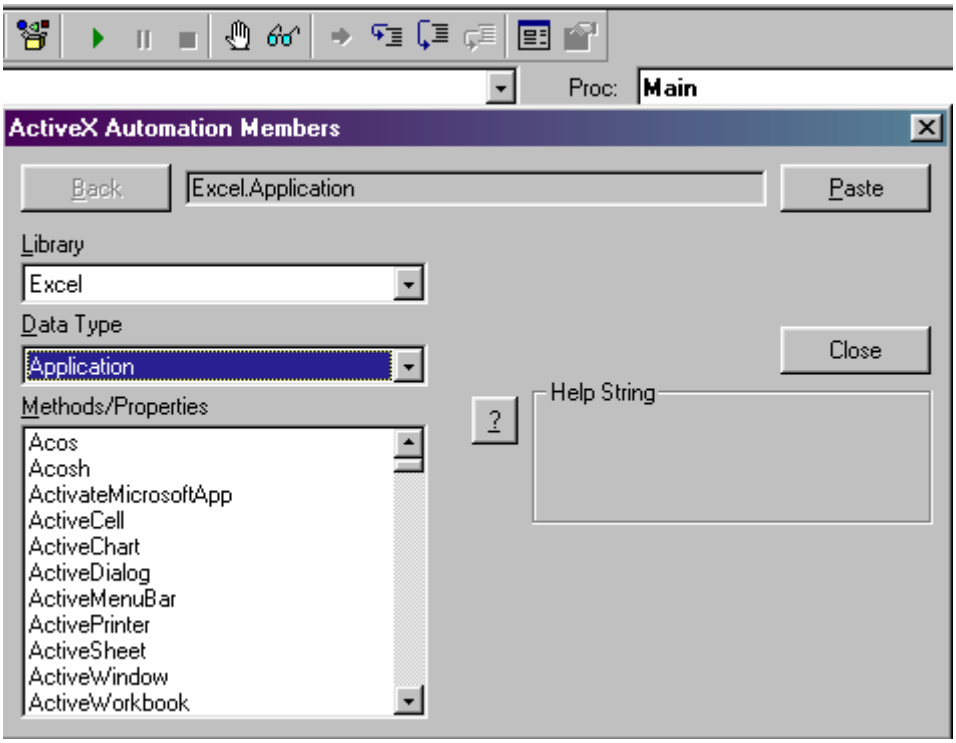
Debug.Print File.ReadLine
End Sub

OLE

Also WinADL allows us to communicate to third party applications via OLE (pronounced olay). This can let us start up compliant applications such as Word or Excel from Cary applications, then send information to the former programs as the data is being collected. To use this feature you will need to be familiar with the "Edit...References" section and the dialog seen from that selection. This will bring up a list of OLE compliant applications within your own computer. You need to select the ones that you need to link to , e.g. Microsoft Excel 5.0 Object Library for communications to Excel (if installed).



When you do this and then select the Browser button (see top left corner of the picture below) you get all the OLE references to Excel to allow you to know the syntax for communication to Excel.



Although this is a lot of information, there is an easier way to establish communication via OLE to Excel. This is found by using the Macro facility of Excel to record a sequence of actions that you want your ADL program to do. For example, make a new sheet, title some rows, enter some data, perform some calculations. When you use the Macro recording of this sequence Excel writes a VBA script file that can be copied into your ADL program. The Excel help will give you great assistance in this area. The following example shows how to access Microsoft Excel, send it a continuum which Excel will place into two columns, headed by the name and modes. The example also uses ReadCtm to get an array of the exact data inside the continuum. The salient points are in the ExcelWrite Sub.
Dim xl As ObjectSet xl=CreateObject("excel.application")
These two lines form the link to Excel. Following this, we set properties inside our object using the DOT notation of Classes. E.g. xl.Visible=true

```
Dim XYValues() As SingleDim CtmData() As SingleDim Count As IntegerDim XMin As SingleDim
XMax As SingleSub ExcelWriteDim xl As ObjectSet xl=CreateObject("excel.application")
With xl
   .Visible = True
   .Application.DisplayAlerts = False
   .Workbooks.Add
   .ActiveCell.FormulaR1C1 = "Wavelength(nm)"
   .Range("B1").Select
   .ActiveCell.FormulaR1C1 = "Abs"
   .Range("A2").Select
   .Range("a2:b"+Trim$(Str$(Count+2))).Value = CtmData
  End With
  Set xl = Nothing
End Sub
Function GetDataIn As BooleanDim Intv As SingleDim i As Integer  If ActiveTrace <> "" Then
GetCtmXRange(ActiveTrace , XMin, XMax)    If MsgBox("Do you want all the trace data ",vbYesNo) =
vbYes Then
     Intv = 1        ' initial interval = 1/2
     Do
```

```
      intv = intv / 2
      ReDim XYValues(1+Abs(XMax-XMin)/Intv)
      Count = ReadCtm(ActiveTrace,XMin, XMax, XYValues)
      Debug.Print Str$(Count)+":"+Str$(LBound(XYValues))+">>"+Str$(UBound(XYValues))
    Loop Until Count*2 < Abs(XMax-XMin)/Intv
  Else
    Intv = Abs(VInput("Enter the interval",,1))
    ReDim XYValues(1+2*Abs(XMax-XMin)/Intv)
    For i = 0 To Abs(XMax-XMin)*2/Intv Step 2
      XYValues(i) = XMax-(I/2)*Intv
      Debug.Print i;XYValues(i)
      XYValues(i+1)=extract(ActiveTrace,XYValues(i))
    Next i
    Count = Abs(XMax-XMin)/Intv
  End If
  ReDim CtmData(Count,1)     ' convert xyvalues to 2d array
  i = 0
  While i <= count
    CtmData(i,0) = xyvalues(i*2)
    CtmData(i,1) = xyvalues(i*2+1)
    i = i + 1
  Wend
  GetDataIn = True
 Else
  GetDataIn = False
 End If
End Function

Sub Main
If GetDataIn Then
 ExcelWrite
Else
 MsgBox"Please load a trace to export to Excel"
End If
End Sub
```