

# RobotBASIC Subroutines

<b>GoSub Subroutines</b>	<b>1</b>
<b>Call/Sub Subroutines</b>	<b>2</b>
<b>Global and Local Variable Scoping</b>	<b>2</b>
<b>By Value Parameter Passing</b>	<b>4</b>
<b>By Reference Parameter Passing</b>	<b>4</b>
<b>Passing a Value to a By-Reference Parameter</b>	<b>6</b>
<b>Optional Parameter Passing</b>	<b>6</b>
<b>Accessing Global Variables in a Call/Sub Subroutine</b>	<b>7</b>
<b>Returning Results From a Call/Sub Subroutine</b>	<b>8</b>
<b>Arrays are Always Global</b>	<b>9</b>
<b>Errors and Debugging While Inside a Call/Sub Subroutine</b>	<b>9</b>
<b>Advantages and Disadvantages of GoSub Subroutines</b>	<b>9</b>
<b>Advantages and Disadvantages of Call/Sub Subroutines</b>	<b>10</b>
<b>Recursion</b>	<b>10</b>
<b>Using RB Subroutines in Real Applications</b>	<b>11</b>
<b>Sonar Distance</b>	<b>11</b>
<b>Serial Communications</b>	<b>12</b>
<b>A Complex Real Application</b>	<b>13</b>



# RobotBASIC Subroutines

In RobotBASIC there are two kinds of subroutines

- GoSub subroutines
- Call/Sub subroutines

This document explains the differences between the two types. We will also explain when one is used in preference to the other. Both types have utility and advantages and disadvantages depending on the situation and application.

## GoSub Subroutines

---

The GoSub subroutine is basically just a chunk of code surrounded by a *label* and a **Return** statement. In all other aspects the subroutine is part of the main program. All subroutines must be set aside from the main area of the program so that they may not be run by the normal flow of the program.

A GoSub subroutine is invoked by the statement **GoSub Label** where **Label** is the name given to the subroutine. During the normal run of the program when the statement **GoSub Label** is encountered, the program flow is diverted to the area where **Label** is then execution continues from there onwards. When at any time later a **Return** statement is encountered the program flow returns back to the line just after the **GoSub Label** statement and keeps going from there.

In all other aspects it is as if the GoSub subroutine code were inserted right after the **GoSub Label** statement. There is no difference whatsoever. Consider this program:

### Program 1

```
Vs = 343.5  
t = 2 \ D = t*Vs \ Print D  
t = 5 \ D = t*Vs \ Print D  
End
```

Notice how the bold parts are exactly the same. Now imagine you wanted to change the formula to say  $0.5*t*Vs$ . In this case it is not so painful. You just modify two lines of code. But if this formula was to be applied in numerous places then every time you want to change the formula you have to go to all those places and change them. If you happen to miss one then there would be unpredictable and hard to debug little bugs.

Consider this program:

### Program 2

```
Vs = 343.5
t =2 \ GoSub Distance
t = 5 \ GoSub Distance
End
//-----
Distance:
  D = t*Vs \ Print D
Return
```

Notice how the subroutine is put after the **End** statement. This will ensure that the program flow will never unintentionally execute the subroutine since the program will end before it reaches that area of the program.

Using the subroutine, all you have to do to change the formula is change it in that one place and then all places that invoke the subroutine will be guaranteed to work.

## Call/Sub Subroutines

---

A Call/Sub subroutine is also a chunk of code that you can invoke when needed much like a GoSub subroutine. But there are many important differences between the two as well as some similarities.

Consider this program that will do the same action as Program 2:

### Program 3

```
Vs = 343.5
call Distance(2,Vs)
call Distance(5,Vs)
End
//-----
Sub Distance(t,Vs)
  D = t*Vs \ Print D
Return
```

Notice that a Call/Sub subroutine is designated with the statement **Sub** before the name of the subroutine. Also the name of the subroutine is no longer a label since there is no **:** after it. Also we invoke the subroutine with the **Call** statement rather than **GoSub**.

Another thing of note is that with the Call/Sub subroutine it is now possible to pass parameters. The **(t,Vs)** section after the name of the subroutine is the way we tell RB that the subroutine will have a parameter called **t** and another called **Vs**. So later when we invoke the subroutine as we did in the program and we give it the value 2 or 5 then in the body of the subroutine **t** will be assigned the value 2 or 5 before the subroutine is run.


We will discuss this in much more details later. But for now just note that this mechanism makes it a lot easier to use the subroutine than the GoSub version. In the GoSub version we had to assign the variable **t** before invoking the subroutine. With the Call/Sub version we just pass it a parameter. There is a lot more to it than that, but for now that is all that matters.

## Global and Local Variable Scoping

---


Variable scoping is how we refer to the *extent of visibility* of a variable. When a variable is declared explicitly using the **Declare** statement or implicitly by assigning it a value then the variable becomes visible at the *level* of the code that declared the variable.

If the variable is declared during the main program execution then it is available at the global level. It becomes a *Global Variable*. That is it is a variable that is at the level of the main program.

 A GoSub subroutine has the same variable scoping as the calling code. That is a Gosub subroutine does not have variable scoping of its own. It inherits the scoping of the code that called it.

If you look at Program 2 you will notice that before we invoked the subroutine we declared **t** and **Vs** implicitly by assigning them values. When the subroutine started running it had the same scoping as the area of code that called it. Thus it had access to the **t** and **Vs** variables and could use them in the body of the subroutine to calculate the distance.

A Call/Sub subroutine is as if we have a new program. It does not know anything about variables declared implicitly or explicitly outside its body. It only has knowledge of variables declared within its body and the ones in the list of its parameters.

 A Call/Sub subroutine has *local* variable scoping. A Call/Sub subroutine is like a program on its own. It only has access to variables declared within its body of code and variables in its parameter list. But also see later how it can also have access to global variables.

Consider this program:

#### Program 4

```
A = 10 \ B = 20
GoSub Test1
Call Test2(A)
End
//-----
Test1:
  Print A
Return
//-----
Sub Test2(N)
  Print N
  Print A //causes an error
  //GoSub Test1
Return
```


Notice how the GoSub subroutine had access to the variable **A** since it is at the same variable scoping as the caller area (main program in this case).

Notice how the Call/Sub subroutine causes an error on the second line (first bold line). When it executed the first line there was no problem because the variable **N** is part of the variables list and the statement **Call Test2(A)** passed along the value of **A** to the subroutine and thus it got assigned to the **N** parameter and the **Print N** statement could therefore print the value of **A**. But it is not actually printing the value of **A**. It is in fact printing the value of **N** which just happens to have been assigned the value that was stored in **A** because the **Call** statement passed that along as a parameter.

On the second line we tried to print the variable **A**. Since there is nowhere in the body of the subroutine any declaration of **A**, the line causes an error.

Now, comment out the first bold line and uncomment the second bold line and rerun the program. There is going to be an error, but it is not in the body of the subroutine. The error will occur when the blue line is executed. Why is that?

The blue line worked earlier because when the **Test1** was invoked it was from within the main program and since **Test1** is a GoSub subroutine and thus has no scoping of its own, it had access to the **A** variable. The second time **Test1** was invoked it was from within the body of **Test2()** and when it tried to print **A** there was no **A** to print, since now the scoping is the scoping of the **Test2()** routine, which has no declared **A**.

 If a GoSub subroutine is called from within a Call/Sub subroutine it will of course have the same scoping as the Call/Sub subroutine, since a GoSub subroutine has no scoping of its own.

## By Value Parameter Passing

When defining a Call/Sub routine you also define a list of parameters to be passed to it. There may be as many as you need or none. Consider this:

### Program 5

```
A = 10 \ B = 20
Call Test1()
Call Test2(A,B+5,30)
//Call Test2(C,20,40) //causes an error
End
//-----
Sub Test1()
  Print "Hi there"
Return
//-----
Sub Test2(X,Y,Z)
  Print X+Y+Z
Return
```

In Program 5 you will notice the mechanism of parameter passing to a Call/Sub routine. **Test1()** has no parameters, but you still need to use the parenthesis in the definition as well as in the invoking statements.

Notice that when you call **Test2()** and pass parameters to it you are actually passing *expressions*. That is any valid expression that results in *value* can be used as the parameter passed to the subroutine.

What takes place is that RB will evaluate the expression in the position of the parameter and then *assign that value to the parameter*. That parameter becomes a valid variable in the local scoping of the routine. This is repeated for all the parameters on the list. So what is going on is that the *value* is passed to the subroutine.



By value parameter passing is when the value of the expression in the parameter position is evaluated and the value is passed to the parameter. This is regardless of the expression being just a variable it is always evaluated before passing it along. Uncomment the bold line and see how it causes an error. This is because the variable **C** does not exist and since it is going to be evaluated before passing along it will cause an error.

## By Reference Parameter Passing

Consider this program:

### Program 6

```
A= 10
Call Test(A)
Print A //prints 10
End
//-----
Sub Test(A)
  Print A //prints 10
  A = 70
  Print A //prints 70
Return
```

Even though the parameter of **Test()** in Program 6 has the same name as the variable in the main program, they are not related in any way. Remember that a Call/Sub subroutine has its own variable scoping which is not related to the variable scoping of the caller area. So the variable **A** in the caller area has no relationship to the variable **A** in the routine.

Also notice that despite **A** being assigned a value in the body of the routine it had no effect on the **A** in the caller area. This is very important and is the real power of *by value parameter passing*. This is why *local*

*scoping is very useful.* The subroutine cannot inadvertently affect variables in the caller area. But what if we did want the routine to affect the value of the variable **A** in the caller area?

Sometimes it is useful to give the routine the ability to change a parameter and also reflect that change on the variable in the caller area. This is *one way of being able to pass back values to the caller area.* It is called *by reference parameter passing.*

Consider this program:

### Program 7

```
A= 10
Call Test(A)
Print A //prints 70
End
//-----
Sub Test(&B)
    Print B //prints 10
    B = 70
    Print B //prints 70
Return
```

Notice in Program 7 the only difference from Program 6 is the blue stuff. But now the bold line will print a different result than the previous program. Why? Notice that despite the parameter being a different name it still was able to change the value of the variable **A** in the caller area.

The prefix **&** in front of the parameter name **B** indicates to RB that the parameter is to be a *by reference parameter*. When we call the routine and pass it a variable in the position of a by-reference parameter that variable is not evaluated first. Instead it is passed as it is and in the body of the subroutine the parameter becomes the same variable, despite being of a different name (*it becomes an alias to the passed variable*).

So in Program 7 when **Call Test(A)** was executed **A** becomes **B** inside the body of the routine. Whatever is done to **B** is also done to **A** (they are the same variable). So printing **B** is like printing **A**, assigning a value to **B** is just as if **A** was assigned the value. They are now for all intents and purposes the same variable. This is what it means to use by-reference parameter passing.



**B** is now an *alias* for **A**.

Consider this program:

### Program 8

```
Call Test(A)
Print A //prints 70
End
//-----
Sub Test(&B)
    Print B //prints nothing (just cr/lf)
    B = 70
    Print B //prints 70
Return
```

Program 8 is Program 7 but with the first line (**A = 10**) removed. Now when we call **Test()** there is no variable **A** that has been declared. Yet there is no error. This is because the variable **A** is not evaluated, it is just passed along to the routine as a variable to be used. But in the first line in the routine (blue line) we try to print **B** which is now **A**, however, **A** has never been defined or assigned a value. So what happens?

RobotBASIC is not a strict variable typing language, though you can make it so if you wish using the **Declare** statement. Thus, when you pass a non-existing variable as a by-reference parameter, RB will create it for you and will define it as a string variable and assign it an empty string (""). Thus in our situation because **A** is going to be passed to the subroutine and does not exist RB will make it an empty string and since **B** is **A**, when we execute the blue line nothing is printed (just line feed).

The rest is as before and when **B** is assigned the value 70, so will **A** and thus when we print **A** on the second line of the main program we will see 70 printed.

Now, remove the **&** from the definition of the subroutine (bold line) and run the program. You will get an error saying that **A** is not a defined variable because now **A** is going to be passed by value to be assigned to **B** and thus will be evaluated, and since it does not exist, you will get an error.

## Passing a Value to a By-Reference Parameter

If you pass a variable to a by reference parameter the variable will be used within the body of the subroutine (it is aliased). However, you can also pass an *expression* instead of just a variable. In this case the expression will be evaluated and the parameter will become a by-value one despite the **&** prefix. If you pass an expression instead of a variable then the value is passed along and since there is no variable to be aliased then the parameter becomes as if it is a by-value one.

There is an exception to this. If the *first* element of the expression is a variable, this variable will be used to be a by reference variable to be aliased inside the subroutine. But the *expression* value will still be assigned to the parameter as an initial value. Notice the second call to the subroutine in the main program. After this call **A** will be 70 since it has been assigned the value 70 in the subroutine.

### Program 9

```
A = 10
Call Test(30*A) //notice we are passing an expression
Print A //prints 10
Call Test(A*20) //passing an expression but there is a valid variable as the
                //first part of the expression
Print A //prints 70
End
//-----
Sub Test(&B)
  Print B
  B = 70
  Print B //prints 70
Return
```

The program outputs the following

```
300 //this is the result of the expression
70
10 //notice how A is still 10
200 //this is the result of the expression
70
70 //notice how A is now 70 not 10
```

## Optional Parameter Passing

When you call a Call/Sub subroutine that has a list of parameters you can optionally omit any or all of the parameters. Any parameters that are not passed a value will not be defined in the body of the subroutines regardless of whether they are by-value or by-reference.

Consider this program:

### Program 10

```
Call Test(10)
Call Test(,,A,10) \ print A
Call Test(,30,,6)
Call Test(,,3)
end
//-----
Sub Test(X,&Y,&Z,W)
  if !vType(X) then X = 0
  if !vType(Y) then Y = 0
```



```

if !vType(Z) then Z = 0
if !vType(W) then W = 0
print X;Y;Z;W
Y = 10 \ Z = 20
Return

```

This is the output of the program

```


10      0      0      0
0       0              10
20
0       30      0      6
0       0       3      0

```

Notice the use of the **vType()** function to ascertain if a parameter is defined or not. This function will return a 0 (false) if the variable is not a defined one. This will occur if the parameter has been omitted and no value or variable (if by-reference) was passed to it.

With the **vType()** function *you can assign missed parameters default values*. Alternatively you can use **vType()** to do other actions like for instance skipping the processing related to the missing parameter.

Study Program 10 and its output and see if you can figure out the reason the output is as it is.

 When designing Call/Sub subroutines for use by other people always make sure to use **vType()** to check if the parameter has been passed a value (or variable if by reference) and make sure that default values are assigned, or skip using the parameter since that would cause an error because the parameter would not exist as a variable.

## Accessing Global Variables in a Call/Sub Subroutine

Remember that a Call/Sub routine has its own variable space and its variables cannot be seen outside it and it cannot see other variables outside it. But in certain situations it would be nice to access variables created in the main program or to create variables that can be used by the main program even after the subroutine is finished or even by subsequent calls to the subroutine. We can do this with the use of the **\_** prefix. By prefixing the variable name with **\_**, RB will know that the global variable is the one that is needed not a local variable with the same name.

Consider this program:

### Program 11

```

A = 10
Call Test()
Print A;B //after the call to the subroutine B will exist since the routine
          //created it. Also notice how A is not modified
End
//-----
Sub Test()
  A = 20
  Print A; _A //notice the fact that the Global variable is called _A in the
              //subroutine and that is not the same as A
  A = 40 \ _A = 50 \ _B = 4 //notice how _B is being created in the global scope
Return

```


This is the output of the program

```

20      10 //notice how the _A is the value from the main program i.e. Global var
50      4  //notice how B now exists and how A has been modified

```

It is preferred to use by-reference parameters to pass along any variables that the subroutine needs. However, if you do need a global variable that has not been passed as a parameter you can use this mechanism.

 Global variables can be used inside a Call/Sub subroutine as *static variables*. Static variables are ones that maintain their values between different invocations of the subroutine. Once a Call/Sub subroutine terminates, its variable space is erased and discarded. So next time the routine is called it will start with a new set of variables. If you need to maintain certain variables between calls to keep some values that need to be used by subsequent calls then the only way to do it is by using global variables where their values will remain and can be used by the next invocation of the routine.


## Returning Results From a Call/Sub Subroutine

There are four mechanisms to return values from a Call/Sub subroutine to the caller area:

1. Using by-reference parameters and assign them the return values.
2. Using the `_` prefix to assign the return values to global variables.
3. Using the **Return Expression** mechanism and using the **XXXX\_\_Result** variable (see below)
4. Using Arrays (see later).

You have already seen the first two mechanisms. The third mechanism is also quite handy and when used in combination with the first method you can return any combination of values back to the caller area. The second method is effective but you should try to use the first and third methods before resorting to the second method.

When returning from a Call/Sub subroutine you use the **Return** statement. However, instead of just returning you can also say **Return Expression**. Where Expression is optional and can be any valid expression. For example you can say **Return sin(x)**.

 In RB, a conditional expression is an expression like any other and can be used in any place an expression can be used. It always results in a 1 (true) or a 0 (false) and therefore can be used just as a number (which it is; 0 or 1). Thus you can say `X = 4*(A > B)`, or `Return (x==5)`.

If you do use an expression to be returned by the **Return** statement, what actually happens is that the subroutine will return to the caller area but with a new variable now available in the variable scoping of the area and this new variable has the value of the calculated expression. The variable is called **XXXX\_\_Result** where XXXX is the name of the subroutine.

Even if you use **Return** alone without an expression the variable **XXXX\_\_Result** will still be available but with an empty string as its value. This is best illustrated with an example:

### Program 12

```
Call Test(A)
Print "-",Test__Result,"-"; A;B
Call Test(A)
Print Test__Result
End
//-----
Sub Test(&N)
  If IsNumber(N) then return N*40
  N = 40 \ _B = 30
Return
```

The output is

```
-- 40      30
1200
```

Notice that the program uses three methods for returning a value. The by-reference variable **N** is assigned the variable **A**, which becomes set to the value 40 if it does not have an already assigned numerical value. Also the global variable **B** is created inside the subroutine by using the `_` prefix to create it and assign it the

value 30; hence the first line of output shown above. Also notice that since the return expression is not given, the return result is an empty string.

On the second call to the subroutine **A** is already assigned a numerical value and so the conditional statement will return from the subroutine with the expression you see. Hence the second output line.

## Arrays are Always Global

There is one exception to the local scoping in Call/Sub subroutines. Arrays are always global regardless. Any arrays created in the main program are also available to the Call/Sub routine and vice versa.

You can also use arrays as another mechanism for returning values from a Call/Sub subroutine. When a subroutine creates an array and fills it with data it becomes available in the calling area. Alternatively the calling area can create an array that will be visible to the subroutine to fill it with data or read data from it.



Arrays can be a very effective way of passing lots of data between the main program and a Call/Sub subroutine or between different Call/Sub routines.

### Program 13

```
Dim A[4] //create an array but it is still accessible to the subroutine
A[0] = 20
Call Test()
Print A[0];A[1]; B[1] //prints 20 60 3
End
//-----
Sub Test()
  A[1] = A[0]*3 //array not created here is still accessible here
  Data B;2,3 //create an array but it is still accessible to caller area
Return
```

## Errors and Debugging While Inside a Call/Sub Subroutine

Normally, when a programming error causes the program to halt, you can use the *View Variables Table* menu option (*Ctrl+B*) under the *Run* menu to view the list of variables in the current program state. These are normally the global variables. However, if the error occurs during the execution of code in a Call/Sub subroutine then the variables in the variables table will be the local scoping variables of the subroutine.

If you need to see the variables in the global variables list you will need to use a debugging session to halt the program at controlled locations and use stepping and so forth.

## Advantages and Disadvantages of GoSub Subroutines

GoSub subroutines are simple to use due to their lack of variable scoping. They can be used without all the complications of passing parameters and worrying about global variables and so forth. However, this can also be a disadvantage. If you use variables in a GoSub subroutine that are the same as variables used in other parts of the main program then you can have hard to debug errors due to variable clashing. You will need to be very careful with variable naming. For instance if you are using the variable **I** in the main program and then you call a subroutine that uses **I** too (e.g. for a counter in a For-Loop) then the subroutine will change the value of **I** and the main program will be inadvertently affected.

Another disadvantage of GoSub routines is that if you use them to perform a certain function that needs to be passed parameters you will have to set all the parameters before calling the routine in separate statements, which is not as clean as with Call/Sub routines.

With GoSub subroutines it is not easy to create recursion (see later) due to lack of local variable scoping, which complicates the design of the recursive parameter nesting. It can still be accomplished but with a lot of work.

One advantage GoSub subroutines have over Call/Sub subroutines is the ability to use an expression instead of a label when calling the routine. Look at this program:

#### Program 14

```
a = "Test"
for i=1 to 2
  GoSub a+i
Next
End
//-----
Test1:
  print 1
Return
//-----
Test2:
  print 2
Return
```

## Advantages and Disadvantages of Call/Sub Subroutines

The local variable scoping of Call/Sub subroutines can complicate some aspects of programming, however, the advantages gained by variable encapsulation quickly outweigh the occasional inconvenience.

The mechanisms of by-value and by-reference parameter passing are versatile and make programming the routine much less complicated than with GoSub subroutines. The **Return Expression** mechanism is also quite useful.

In the next few sections the versatility and convenience of using Call/Sub routines will be illustrated with practical examples.

## Recursion

Recursion is a very powerful concept in computer science. With GoSub subroutines it is hard to accomplish recursion due to lack of local variable scoping. It can be accomplished, but you have to implement your own stack pushing and popping of variables, which complicates the programming quite a lot.

Let's consider a simple recursion example of calculating the factorial of a number using Call/Sub routine:

#### Program 15

```
Print factorial(5) //show result using RB's inbuilt function for verification
Call myFactorial(5)
Print myFactorial__Result
End
//-----
Sub myFactorial(n)
  If n < 2 then return 1
  Call myFactorial(n-1) //recursion
Return n*myFactorial__Result
```

Here is the same but using a GoSub routine:

### Program 16

```
Print factorial(5) //show result using RB's inbuilt function for verification
n = 5 \ GoSub myFactorial
Print result
End
//-----
myFactorial:
  if !vType(result) then result = 1.0 \ n2 = n
  if n2 = 0 then return
  result = result * n2 \ n2--
  GoSub myFactorial //recursion
return
```

Notice how much more difficult it is to do the programming with the GoSub version and also it is not intuitive, which is one of the advantages of recursion. Moreover, this was a simple example – with a slightly more complicated recursion algorithm you would need to push and pop many variables not just one as we did above. In the above we used **result** as an accumulator and stored the original number in **n2** so as to be manipulated without changing the original number, so we did not need to push and pop explicitly.

Notice how when using the Call/Sub subroutine we did not have to worry about the original value being changed by the subroutine due to local variable scoping.

## Using RB Subroutines in Real Applications

So far we have used very simple and somewhat artificial examples to illustrate the points being discussed in regards to the various aspects of Call/Sub subroutines. We now want to see them in use in real life situations and see how the various properties are of utility depending on the situation.

### Sonar Distance

Imagine you have an electronic device that sends out an ultrasound wave and starts a stopwatch timer. It then activates a microphone and starts listening to the ultrasound signal to come back. When the signal comes back it stops the timer and notes the time it took for the signal to go out and be reflected back. This is the **round trip time** duration it took for the sound wave to go out, encounter and object, be reflected and to arrive back at the microphone.

From basic physics we know that **Distance = Speed \*time**. Thus if we know the speed of a sound wave and the time it took to go and come back, then we know the distance to the object and back. In other words, we know the distance to the object since that is half the distance there and back.

The speed of sound however, also depends on the ambient temperature of the air the sound is traveling through. The formula is:  $V_s = 331.3 * \text{Sqrt}(1 + T_c/273.15)$

Where **Vs** is the speed of sound in meters per second and **Tc** is the temperature in degrees centigrade.

If we know the round trip time **t** in microseconds and we want the distance to be in millimeters then the formula is

$$D = (0.5 * t / 1e6) * 331.3 * \text{Sqrt}(1 + T_c / 273.15) * 1000$$

→  $D = 5e-4 * 331.3 * \text{Sqrt}(1 + T_c / 273.15)$  D is in mm, TC is in °C and t is microseconds.

So now if we have a device that gives us the time **t** in microseconds we can calculate the distance to the detected object using the above formula.

It would be nice if we had a subroutine that would do that for us. Also, if we do not know the temperature we want the subroutine to assume a standard ambient temperature of 22.2 °C.

Here is one possibility:

### Program 17

```
call mmDistance(30,,d) \ print d, " mm/sec"
call mmDistance() \ print mmDistance__Result. " mm/sec"
call mmDistance(,2000) \ print mmDistance__Result, " mm"
call mmDistance(,2000,S) \ print S, " mm";mmDistance__Result, " mm"
End
//-----
Sub mmDistance(Tc,t,&D)
  if !vType(Tc) then Tc = 22.2 //default temp if not given
  if !vType(t) then t = 2e6 //default round trip time if not given 2 secs
  D = 5e-4* t * 331.3 * Sqrt(1+Tc/273.15)
Return D
```

The program above demonstrates how versatile the subroutine is.

- We can obtain the result using a by-reference variable.
- If we do not pass a by-reference variable to be filled with the result the fact that the routine returns the result in the **mmDistance\_\_Result** variable comes in handy.
- If we do not give it a temperature then it will assume a default temperature by detecting that the parameter is not defined and giving it a default value if it is not.
- If we do not pass a time value the routine assumes a 2 seconds time value. This can be useful for obtaining the speed of sound in mm/sec as we did in the second line (bold one) in the program. The one above it also gives us the speed of sound in mm/sec but at 30 °C.
- You can see from the program the versatility and power of such a subroutine.
- Notice the use of **vType()** to be able to initialize any optional parameters to default values.
- Notice the fact that we used two ways to return a value to the caller area.
- Notice that even the by-reference parameter is still optional.

## Serial Communications

Imagine you have a system where you can use the serial port to send out a byte command and a byte parameter for the command and then wait for data to come back over the serial port from the external process (e.g. communicating with a microcontroller). Say the data that comes back are 5 bytes.

The wait for the 5-byte response has to be done with a time out. If the bytes do not arrive we want to be alerted to the fact. If the bytes arrive then we want to use the first three bytes to update the values of three global variables (1 byte each) and the 4<sup>th</sup> and 5<sup>th</sup> bytes have to be combined into a 16-bit number using the MSBByte First method. This 16-bit value we want to be returned in a by-reference value.

Consider this subroutine

```
Sub SendCommand(C,P,&V,&x,&s)
  SerialOut C,P
  SerBytesIn 5,s,x
  If x < 5 then return false //not enough bytes received
  _Bumpers = GetStrByte(s,1)
  _Feelers = GetStrByte(s,2)
  _Sensors = GetStrByte(s,3)
  V = (GetStrByte(s,4) << 8)+GetStrByte(s,5)
Return true
```

- In the above subroutine C and P are not optional because if they are not assigned the subroutine will generate an error. Do you know how to make them optional?
- Notice how the subroutine returns true or false in the **SendCommand\_\_Result** variable to indicate if there was a timeout before all 5 bytes were received.
- Notice how the global variables are set with the results from the 5 bytes.
- Notice how the 16-bit value is returned in a by-reference variable.

- The last two parameters are by-reference ones to return the number of bytes received and the received buffer. This allows some versatility if the caller area wants to examine the raw data.
- The routine expects 5 bytes back. Do you know how to make the number of bytes to receive back a parameter of the subroutine?

## A Complex Real Application

---

In this final section we will present a complete and interesting application. However, it is a slightly complex one. Nevertheless, it will demonstrate fully the use of Call/Sub subroutines and their power. Also, you will notice how GoSub subroutines also come in handy. You will see the use of by-reference and by-value parameters and also why using global variables can also be quite necessary.

The best way to appreciate the following discussion is to run the program before you read on. The program simulates reading an accelerometer module. For Example using a microcontroller over a serial port as described in the book *A Hardware Interfacing And Control Protocol*. While in the book we use an actual system to do the reading of the accelerometer, here we will just *simulate* the process.

We will use the Keyboard arrow keys to let the user Pitch and Roll to simulate doing the same with a real device. The data is then used to calculate tilt angles and these are used to plot an artificial horizon AH (or attitude indicator AI) instrument like ones on airplanes.

The accelerometer being simulated is a device that returns values for the X, Y and Z axes in analog voltage levels in reference to a reference voltage level. To read the accelerations on each axis we need to talk to the device and obtain readings for the three axes. However, the first time we do this we need to also read the reference voltage. The actual g-force on an axis is calculated by subtracting the reference voltage from its reading and then multiplying the result with 0.0022.

Once the g-forces on the three axes are calculated the tilt angle of the x-axis or the y-axis is calculated using the **atan2()** function to calculate the angle between the axis and the Z-axis by taking the axis g-force against the Z-axis g-force

Once the angles are calculated the AH can be plotted using slightly complicated geometry calculations that do not need to concern us. We will just use the subroutine to do the plotting. We don't care about the details of how it works.

Study the subroutine **Accelerometer()**. Notice how it uses by-reference parameters to return the values of the g-forces. But also notice how the **&V** parameter is used. The first time we call the subroutine we pass to it the variable **vRef** which has in it the value 0. This causes the routine to do a voltage reading. But it will also set the value back into **vRef**. So the next time the routine is invoked **V** will no longer be 0 and no further reading of the voltage is performed.

Notice how the routine uses the two GoSub subroutines **ReadVoltage** and **ReadAxes** to *simulate* the action of reading the actual device. These GoSub routines will act within the variable scoping of the **Accelerometer()** routine and thus will make the variables available for the routine.

However, notice how the **ReadAxes** routine uses the global variables X,Y, and Z since they are prefixed with **\_**. We need these global variables as *static* variables to hold the value of the current simulated readings. Since we use arrows to increment or decrement the variables we need to maintain their values between subsequent calls to **Accelerometer()**. But every time we call accelerometer we start with a fresh set of variables. So to maintain the values between calls to the Call/Sub subroutine we use global variables, which will be maintained because they are global and thus are used as *static variables*.

Notice the **DisplayAttitude()** subroutine is used with the default settings by not passing parameters to it other than the pitch and roll values.

Notice how we use the **Flip On** and **Flip** commands to activate the RB back-buffer screen graphics to create flicker free animation. Comment out the **Flip On** statement and see what happens.

Notice how using Gosub subroutines along with Call/Sub subroutines is a very useful mechanism. The use of Global variables as *static variables* between calls to a Call/Sub routine can be indispensable.

### Program 18

```

Main:
  vRef = 0 \ Flip On //ensure flicker free animation
  print "Use Up/Dn and Left/Right arrows on the keyboard to Pitch and Roll"
  while true
    Call Accelerometer(vRef,gX,gY,gZ) //obtain the g-forces
    Call DisplayAttitude(atan2(gX,gZ)-pi(.5),atan2(gY,gZ)-pi(.5)) //display AH
    Flip //show the screen
  Wend
End
//-----
//-----
Sub Accelerometer(&V,&X,&Y,&Z)
  X = 0 \ Y = 0 \ Z = 0 \ Success = true //init the values
  if V==0 then GoSub ReadVoltage //read the voltage if needed
  If Success then GoSub ReadAxes //read axes voltages
  if !Success then return false //if the comm. fails
  X = (X-V)*.0022 \ Y = (Y-V)*.0022 \ Z = (Z-V)*.0022 //calc g-forces
Return Success
//-----
ReadVoltage:
  //simulate reading the voltage. For now just return the mid level
  V = 2047 \ Success = true //success is needed when we do real reading
Return
//-----
ReadAxes:
  //simulate reading Axis values. For now just use the Arrow keyboard keys to
  //simulate tilting we will use global variables to keep track of the values
  //between calls to the subroutine. That is static variables
  if !vType(_X) then _X = 2047 //initialize the static variables
  if !vType(_Y) then _Y = 2047
  if !vType(_Z) then _Z = 2502
  n = 5
  if keydown(kc_UArrow) then _X = Limit(_X-n,4095,0)
  if keydown(kc_DArrow) then _X = Limit(_X+n,4095,0)
  if keydown(kc_LArrow) then _Y = Limit(_Y+n,4095,0)
  if keydown(kc_RArrow) then _Y = Limit(_Y-n,4095,0)
  X = _X \ Y = _Y \ Z = _Z
  Success = true //success is needed when we do real readings
Return
//-----
//-----
sub DisplayAttitude(Pitch,Roll,Cx,Cy,r,LW,CW)
  if !vType(Pitch) then Pitch = 0 //initialize all default values
  if !vType(Roll) then Roll = 0
  if !vType(r) then r = 100
  if !vType(Cx) then Cx = 400
  if !vType(Cy) then Cy = 300
  if !vType(LW) then LW = 2
  if !vType(CW) then CW = 10
  //horizon
  T = -Roll-Pitch \ TT = -Roll+Pitch+pi()
  x1 = cartx(r,T) \ y1 = carty(r,T)
  x2 = cartx(r,TT) \ y2 = carty(r,TT)
  x3 = (x2+x1)/2 \ y3 = (y2+y1)/2
  Circle Cx-r,Cy-r,Cx+r, Cy+r

```



```

line x1+Cx,y1+Cy,x2+Cx,y2+Cy,LW,red
//ground and sky
for i=-3 to 3 step 6
  T1 = -Roll-Pitch+dtor(i) \ TT1 = -Roll+Pitch+pi()-dtor(i)
  x1 = cartx(r,T1) \ y1 = carty(r,T1)
  x2 = cartx(r,TT1) \ y2 = carty(r,TT1)
  x4 = (x2+x1)/2 \ y4 = (y2+y1)/2
  j = brown
  if i < 0 then j= lightcyan
  floodfill Cx+x4,Cy+y4,j
next
//ground texture arrays
if !vType(_DAI_Flag)
  dim DAI_b[0]
  data DAI_b;5,10,20,40,60
  dim DAI_a[0]
  data DAI_a;0,dtor(30),-dtor(180),-dtor(40),dtor(10),-dtor(140)
  _DAI_Flag = true
endif
//horizontal ground texture
for i=0 to 4
  T1 = -Roll-Pitch+dtor(DAI_b[i]) \ TT1 = -Roll+Pitch+pi()-dtor(DAI_b[i])
  x1 = cartx(r,T1) \ y1 = carty(r,T1)
  x2 = cartx(r,TT1) \ y2 = carty(r,TT1)
  line x1+Cx,y1+Cy,x2+Cx,y2+Cy
next
//diagonal ground texture
j=dtor(20) \ i=T+j
repeat
  x1 = cartx(r,i) \ y1 = carty(r,i)
  line Cx+x3,Cy+y3,Cx+x1,Cy+y1
  i += j
until abs(i) > abs(TT-j+.2)
Arc Cx-r,Cy-r,Cx+r, Cy+r,,CW,gray //instrument rim
//roll gradations
for k=0 to maxdim(DAI_a)-1 step 3
  i = -Roll+DAI_a[k] \ j=DAI_a[k+1]
  TW = CW/2
  if k >=3 then TW = 2
  rr1 = r+TW \ rr2 = r-TW
  repeat
    x1 = cartx(rr1,i) \ y1 = carty(rr1,i)
    x2 = cartx(rr2,i) \ y2 = carty(rr2,i)
    line Cx+x1,Cy+y1,Cx+x2,Cy+y2,2,white
    i -= j
  until i < -Roll+DAI_a[k+2]-.2
next
//roll or bank indicator
for j=-2 to 2 step 4
  i = -dtor(90-j)
  x1 = cartx(r+CW/2,i) \ y1 = carty(r+CW/2,i)
  x2 = cartx(r-CW/2,i) \ y2 = carty(r-CW/2,i)
  line Cx+x1,Cy+y1,Cx+x2,Cy+y2,3,red
next
//small airplane
rr = r/10
circlewh Cx-2,Cy-2,4,4,white
line Cx,Cy,Cx,Cy+rr-1,2,white
Arc Cx-rr,Cy-rr,Cx+rr,Cy+rr,pi(),pi(),2,white
Line Cx-rr,Cy,Cx-4*rr,Cy,2,white
Line Cx+rr,Cy,Cx+4*rr,Cy,2,white
Return

```