# ZBasic

Application Note

## AN-210 Sharing Data between Tasks

### Introduction

Application note AN-209 discussed various methods of managing tasks in ZBasic. Unless everything is done by a single task, the various tasks in a ZBasic application also need to share data. The methods presented in this application note to share data between tasks are standard in multitasking operating systems but have been adapted to ZBasic.

### The Data Sharing Problem

The example program below (Problem.bas) illustrates the problem of sharing data between tasks. `Main()` starts a task that increments a global integer called count. The main routine then waits for a period of time and calls the same `counter()` routine as the other task. An extract of the code is shown below:

```
Private TaskStack(1 to 50) as Byte ' task stack
Private count as Integer            ' counter

' main routine
Public Sub Main()
   Dim time as Single
   time = Timer()

   count = 0
   CallTask "counter", TaskStack

   Call Sleep(20) ' change this value to see different results
   Call Counter()

   ' Wait for second task to finish if it hasn't already
   Call Sleep(1.0)
   Debug.Print "Final count is ";CStr(count);" after ";CStr(Fmt(Timer()-time,2));" seconds"
End Sub

' common counting routine
Private Sub counter()
   Dim I as Integer
   Dim tempCount as Integer
   For I= 1 to 10
       tempCount = count
       Call Sleep(2)
       If (tempCount <> count) Then
           Debug.print CStr(Register.TaskCurrent);" wrong value"
       End If
       count = tempCount + 1
   Next
End Sub
```

The example is contrived but it illustrates the main aspects of the problem. The call to `Sleep(2)` in the `counter()` routine is meant to simulate work or some other ZBasic function call that could allow another task to run. The result of running the above program is as follows:

```
Final count is 20 after 1.09 seconds
```

In this case the second task ran to completion before `Main()` had a chance to call the `counter()` routine. When the sleep time in `Main()` is reduced from 20 to 10 the following output is received. The first number is the address of the task stack and is used to simply differentiate between the two tasks.

```
212 wrong value
160 wrong value
212 wrong value
160 wrong value
212 wrong value
```

 Published January 2006

```
Final count is 14 after 1.09 seconds
```

In this second case the two tasks tried to update count at the same time and sometimes the value was overwritten with a lower value from the other task. You can try different delay times between the tasks and different delay times within the `counter()` routine to see various results. The net effect is the same - both tasks are trying to write to the same value causing a "multiple update" problem. The following sections describe various solutions to the problem of data sharing between tasks.

## *Single Writer, Many Readers*

A simple solution is to only allow one task to write the shared value and other tasks can only read the value. The other tasks read the last value written and may in fact read the same value multiple times until the "write" task updates the value. An example of using a single writer is shown in Application Note AN-208 (Using I2C with Devantech Ultrasonic Range Finders). The following amended code (SingleWriter.bas) illustrates this idea with unmodified code shown in gray.

```
Private TaskStack(1 to 50) as Byte  ' task stack
Private count as Integer            ' counter

' main routine
Public Sub Main()
   Dim time as Single
   time = Timer()

   count = 0
   CallTask "counter", TaskStack

   Call Sleep(0)
   Call Counter()

   ' Wait for second task to finish if it hasn't already
   Call Sleep(1.0)
   Debug.Print "Final count is ";CStr(count);" after ";CStr(Fmt(Timer()-time,2));" seconds"
End Sub

' common counting routine
Private Sub counter()
   Dim I as Integer
   Dim tempCount as Integer
   For I= 1 to 10
       tempCount = count
       Call Sleep(2)
       If Register.TaskCurrent = 160 Then
           ' write to count
           count = tempCount + 1
       Else
           ' reads count
           If (tempCount <> count) Then
               Debug.print "count changed"
           End If
       End If
   Next
End Sub
```

In this case task 160 (the one using TaskStack) is updating the value and the Main task (212) is only reading it. The value of 160 is hardcoded for simplicity in the example code. The following output shows there were five occasions when the value changed from one read to the next. This illustrates that when using multi-tasking you should not rely on execution order or timing between tasks as you can never tell when a task may run in relationship to others in the system.

```
212 count changed
212 count changed
212 count changed
212 count changed
212 count changed
Final count is 10 after 1.08 seconds
```

## *Using Semaphores*

In many cases only having one writer is too restrictive. Semaphores allow multiple tasks to access the same resource by serializing the access so that in effect there is only one writer at a time. Because ZBasic supports multiple tasks it is also provides a system library function for semaphores. The modification to the `counter()` routine (Semaphore.bas) and the global declaration of the semaphore is shown below.

```
Private sem as Boolean

' main routine
Public Sub Main()
…
End Sub

' common counting routine
Private Sub counter()
   Dim I as Integer
   Dim tempCount as Integer
   For I= 1 to 10
       ' wait for semaphore
       Do While (Not Semaphore(sem))
           Call Sleep(0)
       Loop
        ' update counter
       tempCount = count
       Call Sleep(2)
       If (tempCount <> count) Then
           Debug.print CStr(Register.TaskCurrent);" wrong value"
       End If
       count = tempCount + 1
       ' reset semaphore
       sem = False
   Next
End Sub
```

The required result is shown in the output below. The performance overhead of using the semaphore is not noticeable. Notice that the condition "`If (tempCount <> count) Then`" is never true and could be removed.

```
Final count is 20 after 1.09 seconds
```

In the code above the almost empty do while loop is required to wait for the semaphore. A sleep or delay is important to allow the other tasks to run and hopefully free up the semaphore.

Semaphores can be used to also enforce single-threaded access to other kinds of system resources such as an I2C channel or timer. However semaphores need to be used with care as it is possible to get into a deadlock (or deadly embrace) situation where two or more tasks are waiting for each other's semaphores. The "deadly embrace" terminology comes from fighting scorpions where neither can let go for fear of being stung.

## *Using Atomic ZBasic Operations*

Multitasking in all single processor machines is actually a façade implemented by the operating system or runtime. Under the covers each task that is ready to run is given some time to run and then another task is given the opportunity to run. Sophisticated implementations such as the Windows operating system allow for task prioritization and can "preempt" one task to run another or service an interrupt. In all cases the actual processor is only executing one instruction at a time from one task.

Within the ZBasic runtime a similar situation applies. The ZBasic virtual machine is only executing one ZBasic instruction at a time even though this may require multiple AVR processor instructions. We can make use of this fact to allow multiple writers of shared data providing that a particular write to a shared data item is an atomic ZBasic instruction. Here is an example (Atomic.bas) which shows multiple writers:

```
' common counting routine
Private Sub counter()
   Dim I as Integer
   Dim tempCount as Integer
   For I= 1 to 10
       ' update counter
       tempCount = count
       Call Sleep(2)
       If (tempCount <> count) Then
           Debug.print CStr(Register.TaskCurrent);" wrong value"
       End If
```

```
            count = count + 1
        Next
    End Sub
```

Below is the output from this program.

```
160 wrong value
212 wrong value
160 wrong value
212 wrong value
160 wrong value
212 wrong value
160 wrong value
212 wrong value
160 wrong value
Final count is 20 after 1.08 seconds
```

Even though the program reports that the value has not changed, the answer is correct. This is because the real access happens in the modified line "`count = count + 1`". It works because incrementing the count variable is actually only a single atomic instruction in ZBasic as can be seen from this extract of the ZBasic listing file:

```
                              count = count + 1
00b7 e5d200       INCA_W          00d2
```

The atomic ZBasic operations which write to shared data are:

- Write to any data local or global variable or array element except strings
- Increment or decrement of any variable or array element
- Most system library functions except InputCapture and GetADC

## *Using a Critical Section*

In computer science terminology a critical section is a segment of code that is executed without interruption from other tasks thus forming an atomic operation. A critical section can be used to allow a task to have sole access to data and might be useful for multiple updates to a complex data structure.

In ZBasic critical sections are implemented using the System Library functions `LockTask()` and `UnlockTask()`. Note that there are restrictions on how long a task can remain locked. In particular for our example there is a call to `Sleep(2)` in the `counter()` routine which will unlock the task and allow another task to run. Hence in our case wrapping the critical code that gets and sets the count variable with calls to LockTask() and UnlockTask() does not provide a solution to the data sharing problem presented in this application note. The circumstances of your application may be different. For completeness the changes to `counter()` for `LockTask()`and `UnlockTask()` are given in CriticalSection.bas in the attached zip file.

## *Using Queues*

Queues are another data construct that can be used to control shared access to a resource. Multiple producers can put data onto the queue and a single consumer takes data off the queue. This works in a multi-tasking environment only if the runtime can guarantee that a producer can put a complete packet on the queue in a single operation. Fortunately in ZBasic the system library functions PutQueue(), PutQueueStr, and PutQueueByte() are all atomic as far as the user program is concerned and a semaphore is not needed. An example of using a queue between two tasks is shown in Application Note AN-204 (Input Capture and Multi-tasking for IR Remote Controls).

In the example code below (Queue.bas) an additional task has been added to receive commands from the other tasks via a queue. In this case the command is simply a request to increment the count.

```
' private data for command queue
Private Const incCommand as Byte = &H80
Private cmdQueue(1 to 20) as Byte
Private cmdProcessorStack(1 to 80) as Byte

' routine to process commands
Private Sub cmdProcessor()
   Dim cmd as Byte
   Do
       Call GetQueue(cmdQueue, cmd, 1) ' wait for command to process
       If cmd = incCommand Then
           ' do counter increment here
```

```
            count = count + 1
        End If
    Loop
End Sub

Private TaskStack(1 to 50) as Byte ' task stack
Private count as Integer           ' counter

' main routine
Public Sub Main()
   Dim time as Single
   time = Timer()

   ' start command processor task
   Call OpenQueue(cmdQueue, Sizeof(cmdQueue))
   CallTask "cmdProcessor", cmdProcessorStack

   count = 0
   CallTask "counter", TaskStack

   Call Sleep(0)
   Call Counter()

   ' Wait for second task to finish if it hasn't already
   Call Sleep(1.0)
   Debug.Print "Final count is ";CStr(count);" after ";CStr(Fmt(Timer()-time,2));" seconds"
End Sub

' common counting routine
Private Sub counter()
   Dim I as Integer
   Dim tempCount as Integer
   For I= 1 to 10
       ' update counter
       tempCount = count
       Call Sleep(2)
       If (tempCount <> count) Then
           Debug.print CStr(Register.TaskCurrent);" wrong value"
       End If
       Call PutQueueByte(cmdQueue, incCommand)
   Next
End Sub
```

Note that queues in the opposite direction to get the value of count are not needed in this case as this is a single writer, multiple reader scenario as described previously. In the output below the value is changing quickly and each task never quite has the latest value updated by the queue consumer task. The cost of using the flexibility of queues is additional memory and slower performance.

```
312 wrong value260
 wrong value312 wrong value

260 wrong value312 wrong value

260 wrong value312
 wrong value260
 wrong value
312 wrong value
260 wrong value312
 wrong value
260 wrong value312
 wrong value
260 wrong value312
 wrong value260 wrong value

312 wrong value260 wrong value

Final count is 20 after 1.17 seconds
```

At first glance you might think that I have garbled the output from the program but it is correct and illustrates another instance of a shared resource multitasking problem. The line

"`Debug.print CStr(Register.TaskCurrent);" wrong value"`" in the program above actually gets broken down into three calls to the ZBasic runtime; one for each of the two parts of the debug string separated by a semicolon and one for the <CR><LF> added at the end. Here is the generated annotated ZBasic code:

```
PSHA_W          0x0072 (114)          ' Register.TaskCurrent
SCALL           CVTS_W
SCALL           OUTSTR                ' Output String
PSHI_S          " wrong value"
SCALL           OUTSTR                ' Output String
SCALL           OUTEOL                ' Output <CR><LF>
```

It would appear that by the time the " wrong value" string is queued on the output queue for the serial port, it is time for another task to run and it also outputs some text. At some later point the original task runs again and finally outputs the <CR><LF>. The problem is that the context of a single output message is lost when using Debug.Print.

There are several solutions to this problem including using a semaphore or using one output string by concatenating everything including the <CR><LF> together into one string as follows:

```
Debug.Print CStr(Register.TaskCurrent) & " wrong value" & Chr(&H0d) & Chr(&H0a);
```

The output is now correct and readable as shown below.

```
312 wrong value
260 wrong value
312 wrong value
260 wrong value
312 wrong value
260 wrong value
260 wrong value
312 wrong value
312 wrong value
260 wrong value
…
260 wrong value
Final count is 20 after 1.18 seconds
```

## *Summary*

Multi-tasking is a powerful feature of ZBasic that demands some additional thought by the programmer on the use and sharing of data between tasks. This application note explains the problem of data sharing when using multiple tasks in ZBasic and describes various solutions. The best solution depends on the circumstances of your particular application. One writer, multiple readers is the simplest but least flexible. Semaphores require a task to wait before being able to share a resource whereas queues can be used by multiple producers without waiting at the expense of additional memory and slower performance. Locking a task in memory may also be a viable alternative to using a semaphore.

## *Author*

Mike Perks is a professional software engineer who became interested in microcontrollers a few years ago. Mike has written a number of articles, projects and application notes related to ZBasic, BasicX and AVR microcontrollers. Mike is also the owner of Oak Micros which specializes in AVR-based devices including his own ZX-based products. You may contact Mike at mikep@oakmicros.com or visit his website http://oakmicros.com.