ZBasic Application Note

AN-209 Task Management

Introduction

The multi-tasking capability of ZBasic is a very powerful feature, allowing a much cleaner implementation of interfaces to devices that need periodic service. The ZBasic System Library includes several functions for task management including `ResumeTask()`, `ExitTask()` and `StatusTask()`. These routines are sufficient for simple task management but some applications may require more advanced capabilities. This application note reviews the basic task management concepts and then describes some advanced techniques.

Determining Task Status

Every task in ZBasic, including `Main()`, comprises a task routine and a task stack. The task stack is partitioned into two sub-components: the task control block and the actual stack space for the task. The `Main()` task is created automatically when a program begins to run and its task stack consists of all User RAM that is not otherwise allocated. In contrast, all other tasks are created under program control by using the `CallTask` statement. The code below demonstrates a simple application with two tasks.

```
Private taskStack1(1 to 50) as Byte

Sub Main()
  CallTask "Task1", taskStack1
  Do
    Call Delay(1.0)
    Debug.Print "In Main(), Task1 status is "; CStr(StatusTask(taskStack1))
  Loop
End Sub

Private Sub Task1()
  Do
    Call Delay(2.0)
    Debug.Print "In Task1(), Main status is "; _
        CStr(StatusTask(CByteArray(Register.TaskMain)))
  Loop
End Sub
```

This code illustrates several points related to task management. In `Main()`, the status of `Task1()` is displayed and vice versa. Note, particularly, that the array holding the stack for `Task1()` is passed directly to the `StatusTask()` function which requires a byte array as its parameter. Because all arrays are passed by reference in ZBasic, `StatusTask()` receives the address in RAM of the task stack as its parameter.

In contrast, since the task stack for `Main()` is not explicitly defined like it is for other tasks, a slightly different approach must be used. The built-in system variable `Register.TaskMain` is an unsigned integer value giving the address of the task stack of `Main()`. This integral value can be converted to the required “address of a byte array” type by using the `CByteArray()` function. It should be noted that `StatusTask()` can also be invoked with no parameter provided. In that case, the status of the `Main()` task will be returned.

In this simple example, the value displayed for the status of both tasks always will be 1 meaning that the task is currently waiting for a delay to expire. If the delays are reduced sufficiently, e.g. to 0.0, the status will always be 0 meaning that the task is ready to run.

The page for `StatusTask()` in the System Library Reference Manual describes other status values.

Terminating a Task

In some applications, you may want to terminate a task, either unconditionally or only in certain circumstances. For example, you might want to terminate a task that is awaiting completion of an `InputCapture()` if too much time has elapsed. The sample code below illustrates one way that this can be done.

```
Private taskStack1(1 to 50) as Byte
Private pulseData(1 to 20) as UnsignedInteger
Private icTimeoutTick as Long

Sub Main()
    CallTask "Task1", taskStack1
    Do
        Call Delay(0.1)

       ' see if the input capture has completed
       If (StatusTask(taskStack1) = TaskHalted) Then
           Debug.Print "input capture completed"
           Exit Do
       ElseIf (Register.RTCTick > icTimeoutTick) Then
           ' timeout threshold reached, terminate the task
           Call ExitTask(taskStack1)
           Debug.Print "input capture timed out"
           Exit Do
       End If
   Loop
End Sub

Private Sub Task1()
   ' set the timeout threshold for two seconds and begin capture
   icTimeoutTick = Register.RTCTick + (2 * 512)
   Call InputCapture(pulseData, UBound(pulseData), 0)
End Sub
```

In this code, a timeout threshold is set just before the InputCapture begins. The threshold represents a future RTC tick value beyond which the InputCapture should be considered to have timed out. In a loop, the main task examines the status of the input capture task. If the task has not yet completed, the current RTC value is compared to the timeout threshold and when the threshold is exceeded, the InputCapture task is terminated.

It should be noted that when an InputCapture is terminated prematurely, the content of the pulse width array is indeterminate. There is no way to determine how many entries in the array, if any, contain valid data.

Resuming a Task

Depending on your requirements, you may want to cause a task to resume instead of terminating it. The sample code below is a slightly modified version of the previous example that does just that. Note how a Boolean flag is employed to allow the task to be aware that it resumed prematurely.

```
Private taskStack1(1 to 50) as Byte
Private pulseData(1 to 20) as UnsignedInteger
Private icTimeoutTick as Long
Private icTimeout as Boolean

Sub Main()
  CallTask "Task1", taskStack1
  Do
```

```
    Call Delay(0.1)

    ' see if the input capture has completed
    If (StatusTask(taskStack1) = 2) And (Register.RTCTick > icTimeoutTick) Then
      ' timeout threshold reached, indicate a timeout and resume the task
      icTimeout = true
      Call ResumeTask(taskStack1)
    End If
  Loop
End Sub

Private Sub Task1()
  Do
    Debug.Print "starting input capture"

    ' set the timeout threshold for two seconds and begin capture
    icTimeout = false
    icTimeoutTick = Register.RTCTick + (2 * 512)
    Call InputCapture(pulseData, UBound(pulseData), 0)
    Debug.Print "input capture "; IIf(icTimeout, "timed out", "completed")
  Loop
End Sub
```

It is important to note that calling `ResumeTask()` does not cause the target task to begin executing immediately. Rather, it removes all conditions that are preventing it from running thus rendering it ready to run at the next opportunity. It will execute again when it comes up in the normal task rotation sequence.

Advanced Task Management

As noted earlier, each task has an associated Task Control Block - a data area that occupies the first few bytes of the task stack. When a task is activated its task control block is initialized and then inserted into a circular linked list immediately following the task control block of the then-current task. When a task terminates, either normally or by the use of `ExitTask()`, the task control block is removed from the circular linked list. The table below gives some information on the structure of the task control block.

Task Control Block Elements

Offset	Length	Description
0	1	Task status. See `StatusTask()` for details.
1	2	Remaining time to sleep (in RTC ticks).
3	2	Address of next task control block.
5	6	Task context: IP, BP, SP (valid only when not the current task).
11	1	Task control flags (used internally).
12	2	Address of the byte following the end of the task's stack.

N.B. this information is considered implementation detail and is subject to change as necessary.

Caution: directly modifying the task control block (other than the sleep counter) will probably cause your program to malfunction.

When it comes time to switch tasks, the system examines the tasks on the task list in order beginning with the task following the current task. The first task that is found that is ready to run will execute next.

You may navigate the task control block ring beginning with any valid task control block. The built-in registers `Register.TaskCurrent` and `Register.TaskMain` are useful for this purpose as is the memory address of any active task stack. The code fragment below shows how to display the status of all existing tasks.

```
Dim task as UnsignedInteger

' begin with the current task
task = Register.TaskCurrent
Do
  ' display task status
  Debug.Print "Task: "; CStrHex(task);
  Debug.Print "  Status: "; CStr(StatusTask(CByteArray(task)));
  Debug.Print "  Stack Size: "; CStr(RamPeekWord(task + 12) - task)

  ' get the pointer to the next task in the linked list
  task = RamPeekWord(task + 3)

  ' stop when back at the current task
Loop While (task <> Register.TaskCurrent)
```

On occasion, it may be useful to extend or shorten the sleep time of sleeping task. This may be done by directly modifying the "time to sleep" value in the task control block. The units of this value are RTC ticks, approximately 1.95mS. Note that changing the sleep time of a task that is not currently sleeping has no effect. There currently is no way to force a task to sleep.

The execution context of a task includes three 2-byte values: the instruction pointer (IP), the base pointer (BP) and the stack pointer (SP). Of these, only the SP has any particular value to end-users. It may be useful in determining stack usage of various tasks. Note, however, that the execution context of a task control block is not updated until a particular task is swapped out. For the currently executing task, `Register.UserSP` gives the stack pointer value.

Author

Don Kinzer is the founder and CEO of Elba Corporation. He has extensive experience in both hardware and software aspects of microprocessors, microcontrollers and general-purpose computers. Don can be contacted via email at dkinzer@zbasic.net.