Basic Express
Sistema operativo

Versión 2.0

Traducción española: Alicia Bernal, Revisión: Pablo Pompa
www.superrobotica.com

2.00.H

Índice de contenidos

Variables persistentes en BasicX

Un concepto totalmente exclusivo de BasicX son las variables persistentes. Estas son variables que, en lugar de residir en la memoria RAM, residen en la memoria EEPROM, lo que significa que las variables persistentes retienen sus valores también una vez que se desconecta la alimentación.

Las variables persistentes tienen propiedades similares a las variables normales. Pueden utilizarse como argumentos en expresiones, subrutinas o llamadas de funciones. Así mismo, pueden asignar de la misma manera que las otras variables.

Definición de las variables persistentes

¿Cómo se definen las variables persistentes? A modo de ejemplo, en primer lugar escribimos un programa utilizando todas las variables basadas en RAM:

```
Dim MidnightTemp As Single ' Variable basada en RAM

Sub LateNight()

    Dim Temp As Single

    Do
        ' ¿Estamos 60 segundos después de media noche?
        If (Timer < 60.0) Then

            ' Registrar temperatura y abandonar.
            Call ReadTemperatureSensor(Temp)
            MidnightTemp = Temp
            Exit Sub
        End If

        ' Comprobar cada 45 segundos.
        Call Sleep(45.0)
    Loop

End Sub
```

Una vez que el programa está funcionando normalmente, tomamos la siguiente variable y la cambiamos a una basada en RAM:

```
Dim MidnightTemp As Single
```

A una basada en EEPROM:

```
Dim MidnightTemp As New PersistentSingle
```

Tenga en cuenta que el resto de los programas no cambia en absoluto. Sólo cambia el tipo de variable. Cuando el compilador recompila el programa, se genera el código correcto para que todo se ejecute en un segundo plano.

Las variables persistentes, por si propia naturaleza, retienen sus valores después de la desconexión de la corriente y por lo tanto resultan ideales para elementos como puntos de referencia (*setpoints*), parámetros definidos por el usuario, valores de tiempos de desconexión, retardos de seguridad, retardos de inicio de motores y configuraciones de compresores. Así mismo, también son muy

convenientes para los registros de datos de los estados de alarma, temperaturas extremas y marcas de tiempos. Cuando se trata de variables persistentes el único límite es su imaginación.

Las variables persistentes pueden iniciarse a través de la red, o bien pueden ejecutarse en un programa local especial utilizado únicamente para la inicialización. Tras el proceso de inicialización, su programa principal seguirá ejecutándose de manera normal haciendo uso de las variables. Las variables persistentes pueden actualizarse en cualquier momento a través de la red o posteriormente a través de un programa independiente.

Sin embargo, existen algunas limitaciones que se deben tener en cuenta al utilizar variables persistentes:

Límites de ciclos de escritura

Existe un límite para el número de veces que se puede "escribir" o asignar una variable persistente. Normalmente, la memoria EEPROM dentro del chip de BasicX garantiza hasta 100.000 ciclos de escritura. Este número de ciclos podría parecer muy elevado pero sólo si actualiza sus parámetros muy de tarde en tarde. Por otro lado, si el programa de manera accidental envía un bucle de escritura a una variable persistente, este límite se podría exceder en tan sólo unos segundos.

No obstante, las lecturas pueden ser virtualmente ilimitadas. Podrá leer una variable persistente como si se tratase de cualquier otro tipo de variable, sin preocupaciones de superar el límite.

Tiempo de escritura

Una variable persistente tarda mucho más en escribirse que una variable basada en RAM. Cada byte tarda aproximadamente 4 milisegundos en escribirse. BasicX trata de servirse de este tiempo para aprovechar otras capacidades de multitarea u otros usos del procesador durante este tiempo. No obstante, dependiendo del programa este retardo puede ser importante.

Paso de parámetros

Si va a pasar una variable persistente como un argumento de un subprograma, deberá pasarla por valor. No es posible pasarla por referencia.

Declaraciones de nivel de módulo

Todas las variables persistentes deben ser declaradas en el código de nivel de módulo. No es posible utilizarlas como variables locales.

Errores comunes (Sólo BX-01)

Si escribe una variable persistente, el sistema operativo le permitirá leer la variable antes de finalizar la operación de escritura. En el siguiente ejemplo, se supone que X inicialmente es 1:

```
Dim X As New PersistentInteger, Y As PersistentInteger

' Valor inicial de X es 1.
X = 3
Y = X
```

En la última expresión, el valor de Y puede ser 1 en lugar de 3. Se necesita una solución para insertar al menos 4 ms de retardo después de escribir una variable persistente.

Este problema sólo afecta a los sistemas BX-01, y no a otros sistemas BasicX.

Clases de datos de bloque

Inicialización de las matrices

Un problema con las matrices convencionales es que no existe una manera sencilla de inicializarlas sin incurrir en un error de rendimiento. Los dialectos tradicionales de Basic utilizan enunciados de datos (DATA) para funciones similares, sin embargo utilizar los enunciados (DATA) no es sencillo ni son compatibles con Visual Basic.

BasicX intenta resolver este problema proporcionando al sistema unas clases de datos de bloque (DATA BLOCK) definidos por el sistema, que son los equivalentes de las matrices inicializadas y almacenadas en la memoria EEPROM:

1-Clases de matrices dimensionales (sólo bytes):

```
ByteVectorData
ByteVectorDataRW
```

2- Clases de matrices dimensionales:

```
[Byte | Integer | Long | Single] TableData [ RW ]
```

Las clases de datos de bloque pueden ser de sólo lectura o de lectura-escritura. Si el nombre de la clase tiene un sufijo RW, es de lectura-escritura. De lo contrario, es sólo de escritura.

El prefijo del nombre de la clase determina el tipo de dato del objeto. Las clases VectorData son siempre de tipo Byte. Las clases TableData pueden ser de tipo Byte, Integer, Long o Single.

Todos los objetos de los datos de bloque deben declararse en el nivel de módulo. A continuación, se incluyen algunos ejemplos de declaraciones de objeto:

```
    Dim B As New ByteVectorData         ' 1D matriz tipo byte, sólo
lectura.
    Public BRW As New ByteVectorDataRW  ' 1D matriz tipo byte,
                                          lectura-escritura.
    Private S As New SingleTableData    ' 2D matriz tipo float, sólo
lectura.
```

Método fuente

El método fuente (Source) define el fichero de datos desde el que un objeto obtiene sus datos. El fichero se lee en el momento de compilación, y a continuación se carga en la memoria EEPROM al mismo tiempo que se descarga el programa BasicX Código de ejemplo:

```
    Call B.Source("ByteVector.txt")
    Call BRW.Source("C:\Temperatures.dat") ' Se puede usar la ruta de
acceso
                                             Completo del PC
    Call S.Source("CalibrationCurve.dat")
```

El método fuente (Source) debe llamarse antes de leer o escribir en los datos internos del objeto. El comando de Source debe ser un literal de una sola cadena.

Cada fichero de datos de bloques se trata de un fichero ASCII normal de texto que contiene una lista de números. Los ficheros VectorData deberían contener 1 número por fila. Los ficheros TableData pueden contener entre 1 y 253 números por fila, y cada fila debe contener el mismo número de entradas. Los números están separados bien por una coma o por delimitadores de espacio.

Un objeto de datos de bloque se trata de manera similar a una matriz, en las que las dimensiones de la matriz están determinadas de manera implícita por el fichero fuente. Para objetos de una dimensión (1D), el número de filas determina las dimensiones de las matrices. Para objetos de dos dimensiones (2D), el número de columnas y de filas determina las dimensiones de las matrices.

Advertencia – si intenta leer o escribir en la propiedad Value de un objeto de datos de bloque antes de llamar al método fuente (Source), los resultados estarán sin definir (consulte los detalles acerca de la propiedad Value).

Propiedad Value (Valor)

La propiedad Value (Valor) le permite leer los datos internos almacenados en un objeto de datos de bloque. En el caso de los objetos de lectura-escritura, también es posible escribir en esta propiedad. La propiedad Value es una propiedad por defecto, lo que significa que es referida de manera implícita.

Un objeto de datos de bloque de una dimensión (1D) se trata de la misma manera que una matriz de una dimensión (1D), en la que el índice corresponde con el número de filas. La numeración de las filas empieza en 1. En el siguiente ejemplo, V(1) es el primer elemento de la matriz:

```
' En el nivel de módulo.
Dim V As New ByteVectorDataRW  ' El objeto es de lectura-escritura.
Dim B As Byte

    [...]

' Leer el primer elemento.
B = V(1)

' Escribir el tercer elemento.
V(3) = B + 5
```

Un objeto de datos de bloque de dos dimensiones (2D) se trata de la misma manera que una matriz de dos dimensiones (2D), en la que el índice corresponde al número de columnas y el segundo índice al número de filas. La numeración de las columnas y filas empieza en 1. En el siguiente ejemplo, VRW(1, 1) es el primer elemento de la matriz:

```
Dim VRW As New LongTableData  ' El objeto es sólo de lectura.
Dim L As Long

    [...]

' Leer columna 2, fila 3.
L = VRW(2, 3)
```

Advertencia – Un objeto de datos de bloque es similar a una variable persistente en lo que respecta a las limitaciones de los ciclos de escritura y al tiempo que se tarda en escribir en el objeto (consulte la sección de las variables persistentes si desea obtener más detalles).

Propiedad DataAddress

La propiedad DataAddress devuelve el principio de la dirección EEPROM de los datos internos del objeto. La propiedad DataAddress puede utilizarse también como el parámetro de dirección de las llamadas del sistema GetEEPROM y PutEEPROM.

DataAddress es un tipo Long sólo de lectura. Código de ejemplo:

```
Dim T As New IntegerTableData, Addr As Long
Dim A1 As Integer, A2 As Integer

    [...]

' Esto devuelve el principio de la dirección EEPROM de los datos
internos del objeto.
Addr = T.DataAddress

' Estas dos expresiones son formas equivalentes de lecturas del primer
elemento de T.
A1 = T(1, 1)
Call GetEEPROM(T.DataAddress, A2, 2)
' En este punto, A1 y A2 deberían ser iguales.
```

Red de BasicX (sólo BX-01)

Hardware de red

La red que integra BasicX es una de las características más potentes del sistema. Sus múltiples chips pueden interconectarse en las redes, estando limitadas no sólo por cuestiones físicas como la longitud del cable, sino por cuestiones electrónicas. El esquema de direcciones de red de BasicX permite hasta 65.000 nodos por red. Cada nodo de la red tiene el mismo acceso y prioridad que cualquier otro nodo de la red, permitiendo una comunicación de punto a punto desde cualquier nodo y en cualquier momento.

Físicamente, la red de BasicX está configurada con topología de bus. Es posible implementar una configuración de estrella con un hub de BasicX. Con la configuración de bus estándar los chips de BasicX tienen una conexión de tipo margarita.

Para distancias cortas dentro de una placa de circuitos, dentro de un chasis normal, o para un espacio limitado como el interior de un robot o un vehículo, los chips de BasicX pueden ser interconectados utilizando un único cable o una resistencia por cada chip.

Para distancias largas, los chips de BasicX han sido fabricados para ser interconectados a través de un chip transmisor RS-485 estándar. Dependiendo del tipo de transmisor utilizado, una red BasicX puede ampliarse a cientos de metros y miles de nodos. Normalmente es posible interconectar hasta 32 sistemas BasicX en red con distancias de hasta 300 metros, sin repetidores. Así mismo, también es necesario utilizar un cable de tierra, que puede obtenerse del cable de tierra de cualquier componente del equipo, o a través de un cable exclusivo de tierra. Muchos usuarios deciden utilizar un sistema de cuatro cables: dos de ellos para alimentar y servir de tierra y el resto para la red. Se recomienda el uso de un cable de Categoría 5 (CAT5).

Software de red

Desde el punto de vista de un programador, es posible enviar o recibir cualquier tipo de variable de BasicX a través de la red. Es más, los datos pueden recibirse desde sistemas que nunca hayan sido configurados para enviar datos. ¿Cómo es esto posible? Efectivamente esto suena un poco extraño, pero este concepto funciona y de manera muy potente. La red de un chip de BasicX se constituye como un sistema completamente independiente de la ejecución de los códigos; por lo tanto se ejecuta como si se tratase de una tarea independiente. Incluso si se ha detenido el programa de Basic, la red puede seguir enviando y recibiendo datos del chip del programa BasicX detenido.

A continuación, se incluye un ejemplo a modo de demostración:

```
' Programa ChipA, dirección de nodo 99.
Dim Counter As Integer
Dim OkToCount As Boolean
'-----------------------------------------------------
Sub Main()
    OkToCount = False
    Counter = 0
    Do
        If OkToCount Then
            Counter = Counter + 1
        End If
    Loop
End Sub
```

El programa ChipA no es muy complicado. Tenga en cuenta que no hay código en este programa para enviar datos o recibir comandos. De hecho, el contador no contabilizará sin algún tipo de ayuda, dado que OkToCount comienza como falso (false). Cuando el chip de BasicX se descarga con código, el programador tiene que definir si el chip requiere o no red. Por lo tanto, si el programador especifica una dirección de un nodo para el chip, la conexión de red arrancará antes de que se inicie el programa principal. En este caso, supongamos que 99 es la dirección del nodo del chip de BasicX Chip que ejecuta ChipA.

Una vez que está compilado un programa BasicX, se generará un fichero de mapa. Este fichero de mapa muestra la ubicación de todas las variables estáticas (nivel de módulo) en la memoria RAM. Todas las variables persistentes están también incluidas, con sus ubicaciones en la memoria EEPROM. El fichero de mapa tiene el nombre *ProgramName*.mpx. El fichero de mapa es necesario para que otros chips puedan conectarse con los datos cuando sea necesario. En nuestro ejemplo, el fichero de mapa tiene el siguiente aspecto:

```
Public Const ChipA_Counter As Integer = 400 ' &H190
Public Const ChipA_OkToCount As Integer = 402 ' &H192
```

La convención de los nombres de ficheros es la siguiente: "ModuleName_VariableName". El nombre del módulo va seguido del nombre de la variable, con un guión bajo como separador. El nombre del módulo puede o no ser el mismo que el nombre del programa.

Si el fichero de mapa está incluido en el proyecto de otro programa, el proyecto sabrá ahora dónde se encuentra CounterA y OkToCount en el programa ChipA. Si edita y cambia el código fuente de ChipA, el fichero de mapas cambiará también, manteniendo cada programa actualizado con las ubicaciones más recientes de las variables. Por supuesto, el "otro" programa debe ser también recompilado si las variables se cambian de ubicación, se añaden o eliminan. Lo único que debe recordar es el nombre del módulo en el equipo remoto, el nombre de la variable del módulo, y la dirección de la placa del chip remoto.

Ahora ya ha llegado el momento de crear el programa ChipB. Este programa leerá y escribirá datos desde el programa en ejecución ChipA.

```
' Programa ChipB, dirección de nodo 86.
Dim MyCounter As Integer
Dim CountStart As Boolean
Dim Result As Byte
'-----------------------------------------------------
Sub Main()

    ' Se puede iniciar el contador remoto.
    CountStart = True

    ' Configure los datos remotos para iniciar el contador.
    Call PutNetwork(99, ChipA_OkToCount, CountStart, Result)

    Do
        Call GetNetwork(99, ChipA_Counter, MyCounter, Result)

        ' Detener el contador por encima de 1000.
        If Mycounter > 1000 Then
            CountStart = False
            Call PutNetwork( _
                99, ChipA_OkToCount, CountStart, Result)
            Exit Do
        End If
    Loop
End sub
```

Este programa inicia el recuento en el programa ChipA al enviar un booleano a la ubicación OkToCount. En este punto, el programa ChipB empieza a comprobar el funcionamiento del contador, y cuando es superior a 1000, hará que el contador se detenga y establecerá OkToCount como valor falso (false). Debido a la duración de los dos sistemas, una vez que ha finalizado el programa ChipB, el contador en ChipA será muy superior a 1000.

¿Qué hemos aprendido con este sencillo experimento? En unas breves líneas de código, tenemos dos chips comunicados entre sí e intercambiando información. Como ya se ha explicado anteriormente, el programa ChipA no tiene código que enviar o recibir de ningún otro chip. De hecho, cientos de chips podrían obtener el valor del controlador y activar/desactivar el proceso de recuento. El programa ChipA no tiene que saber quiénes son o si tienen algún código que procesar para sus peticiones.

A continuación, encontrará un ejemplo de un sencillo sistema de riego. Programa IRRG:

```
' Programa IRRG, dirección de nodo 10.
Dim WaterOn As Boolean
'-----------------------------------------------------
Sub Main()

    Dim Hour As Byte, Minute As Byte, Second As Single

    Do
        Call GetTime(Hour, Minute, Second)

        If (Hour = 10) and (Minute = 30) Then

            WaterOn = True

            ' Encender válvula.
            Call PutPin(5, bxOutputHigh)
```

```
            Do
                Call GetTime(Hour, Minute, Second)
                If (Minute = 45) Then
                    Exit Do
                End If
            Loop

            WaterON = False

            ' Apagar válvula.
            Call PutPin(5, bxOutputLow)
        End If
    Loop

End Sub
```

En este sencillo ejemplo, se muestra el uso de un reloj de tiempo real para registrar la hora del día. A las 10:30 AM de cada día, este programa abrirá una válvula y la cerrará 15 minutos después. Establecerá el valor WaterOn como verdadero (true) cuando la válvula esté activa. De esta manera, lo único que necesitará será otro chip para comprobar el estado de la válvula de agua es la siguiente expresión:

```
Dim RemoteWaterOn As Boolean
Call GetNetwork(10, IRRG_WaterOn, RemoteWaterOn, Result)
```

Y después comprobará el valor de RemoteWaterOn, reflejando el estado actual de la válvula.

Tenga en cuenta que el programa IRRG no tiene código para enviar la variable WaterOn a nadie. Se ejecuta en un segundo plano, detrás de IRRG. Lo increíble de este concepto es que se puede concentrar en realizar el trabajo en el controlador sin tener que preocuparse de los detalles de red. Por supuesto, el "maestro" (master) necesita conocer más detalles sobre la red como de todos los aspectos.

Seguiremos el esquema de este programa de riego para demostrar algunos de los conceptos de red. A continuación, crearemos un sistema más complejo para poder modificar las horas, los días y la duración del riego. Si utilizamos variables persistentes, el sistema seguirá activo después del corte de alimentación, manteniendo la programación intacta. Realizaremos la demostración de una válvula y dejaremos que el usuario amplíe el sistema.

```
' Programa IRRG2, dirección de nodo 10.
Dim WaterDay(1 To 7) As New PersistentBoolean ' True = regar hoy.
Dim WaterHour As New PersistentByte
Dim WaterMinute As New PersistentByte
Dim WaterLength As New PersistentByte
Dim WaterOn As Boolean
'--------------------------------------------------
Sub Main()

    Dim Hour As Byte, Minute As Byte, Second As Single
    Dim I As Byte, DayOfWeek As Integer

    WaterOn = False
    Do
        Call GetTime(Hour, Minute, Second)
        DayOfWeek = CInt(GetDayOfWeek)

        If (WaterDay(DayOfWeek) And _
            (Hour = WaterHour) And _
            (Minute = WaterMinute)) Then
```

```
                WaterOn = True
                Call PutPin(5, bxOutputHigh)
                For I = 1 To WaterLength
                    Call Sleep(60.0) ' Detenerse durante 60 segundos.
                Next
                WaterOn = False
                Call PutPin(5, bxOutputLow)
            End If
        Loop
    End Sub
```

Ahora ya hemos creado un controlador más complejo, pero en este caso el maestro sigue pudiendo decir si el riego está activo a través de los mismos comandos sencillos:

```
Dim RemoteWaterOn As Boolean
Call GetNetwork(10, IRRG2_WaterOn, RemoteWaterOn, Result)
```

No importa lo complejo que sea nuestro controlador de riego, mientras utilicemos la misma variable WaterOn, cualquier miembro de la red podrá comprobar el estado del sistema.

Los ejemplos que aparecen anteriormente son aplicaciones sencillas de maestro/esclavo (master/slave). Sin embargo, dado que un sistema BasicX puede enviar y recibir datos de cualquier otro sistema BasicX, también se pueden crear aplicaciones de igual a igual (peer-to-peer). Escribir estos tipos de programas es más complejo, ya que requieren un nivel de interconectividad y compatibilidad mayores. Por lo tanto, el factor de flexibilidad se hace más importante a medida que los sistemas se hacen cada vez más complejos, en los que se pueden integrar motores, servos y otros operadores controlados como respuesta a registros de temperatura, presión o sensores similares.

Broadcasting y groupcasting

Puede utilizar los sistemas BasicX trasmitir (o “broadcast”) los datos a todos los nodos de la red de manera simultánea. Así mismo, también puede dividir los nodos en grupos, y trasmitir por grupos (o "groupcast"). Puede tener hasta 254 grupos en una misma red.

Las direcciones de red son enteros sin signo de 16 bits. Las direcciones especiales están reservadas para las trasmisiones de tipo broadcasting y groupcasting.

Broadcasting – si la dirección de red es 65 535 (&HFFFF), entonces el mensaje se trata de un “broadcast”, y se dirige a todos los nodos de manera simultánea.

Groupcasting – si el byte de nivel superior de la dirección de red es 255 (&HFF) y el byte de nivel inferior es distinto de 255, entonces es mensaje se trata de un “groupcast”, en el que el byte de nivel inferior es la dirección del grupo. Es decir, se trasmite un paquete con una dirección de red de &HFFxx al grupo "xx".

Los modos Broadcasting y groupcasting imponen restricciones en la dirección de red de cada sistema BasicX. Por ejemplo, es obvio que un nodo no puede tener la dirección 65 535, ya que esa dirección está reservada para la transmisión “broadcasting”.

Las reglas son las siguientes:

(1) El byte de nivel superior de cada dirección no puede ser 255 (&HFF), lo que implica que todas las direcciones de los nodos están restringido al rango 0 - 65 279 (&HFEFF).

(2) La dirección de grupo de cada nodo tiene el rango 0 - 254 (&HFE).

Capacidad de multitarea en BasicX

Una de las características más potentes de BasicX es su capacidad para ejecutar múltiples tareas de manera simultánea. Esta capacidad de multitarea permite simplificar programas complejos al dividirlos en piezas más pequeñas y más manejables. Cada tarea puede bien cooperar con otras tareas o trabajar de manera independiente. Es como tener múltiples procesadores funcionando al mismo tiempo.

Problemas de tiempo

Obviamente, existen algunas limitaciones: la primera que en realidad sólo hay un procesador en el sistema. El uso compartido de los recursos del procesador añade un cierto retardo que puede desacelerar un programa.

Aún así, no es extraño que un sistema integrado pase la mayor parte de su tiempo esperando que el usuario introduzca algo o esperando a una determinada hora del día. Es más, un sistema puede dedicarse exclusivamente a registrar datos a intervalos poco frecuentes. En estos casos, las ventajas de desarrollar códigos más rápidos y de un mantenimiento más sencillo puede contrarrestar los efectos y limitaciones de la multitarea.

Pila de tareas

Los programas con multitarea generalmente necesitan más memoria RAM que los programas con una tarea sencilla. Si utiliza dos o más tareas, cada una de ellas (que no sea el programa principal) necesita una pila explícita concreta. El programa principal utiliza una pila implícita asignada automáticamente por el compilador. Cada pila de tareas se trata de una matriz de bytes que deben estar en el código de nivel de módulo (estático).

¿Cómo debe ser de grande la pila de tareas? Esta pregunta no tiene fácil respuesta. Las pilas se utilizan para procesar el código ejecutable, como por ejemplo llamadas a subprogramas, expresiones matemáticas y transferencias de datos. En general, es difícil predecir la memoria necesaria para estos procesos, especialmente si utiliza el método recursivo.

Una tarea sencilla sin variables locales, ecuaciones y comparaciones sencillas, y sin llamadas a subprogramas podría funcionar con una pila de tan sólo 32 bytes. Otras tareas más complejas podrían necesitar más de 1000 bytes de espacio de pila.

Cómo calcular la utilización de la pila

Debe determinar de manera empírica la utilización de la pila. Una técnica es comenzar con una pila de gran tamaño que esté precargada con los datos ya conocidos. Después de ejecutar el programa, revise los datos de la pila para comprobar los recursos utilizados. (Se da por sentado que la tarea no devuelve los mismos datos con los que empezó).

Para hacer este cálculo, deberá tomar este resultado, añadir un 20 % a ese número y volver a ejecutar de nuevo el programa. Si el programa funciona correctamente, reduzca el margen un 10 %, que le permitirá tener margen por si el programa se ejecuta de manera diferente.

Estas son sólo pautas de acción, por supuesto, y los requisitos de la pila dependen en gran medida del diseño del programa. Por ejemplo, si el comportamiento del programa es extremadamente sensible al tráfico o a las interrupciones de la red, el espacio necesario de la pila puede depender de factores externos.

Afortunadamente, las pilas de tareas son activas sólo cuando la tarea está activa. Estas *no* son las pilas de procesador principal y por lo tanto no se verán afectadas *directamente* por las interrupciones, por la red u otros factores del tráfico de red del procesador principal.

Con una buena planificación, el chip de BasicX puede controlar programas con literalmente cientos de tareas. Las únicas limitaciones son la potencia total de procesado y la memoria para las variables y pilas. A modo de prueba, se ha creado un programa con 1000 tareas para poner a prueba el motor multitarea. El programa funcionó – no excesivamente rápido pero funcionó.

Conmutación de tareas

Las tareas comparten los recursos del sistema por orden de llegada, excepto para aquellas tareas activadas por interrupciones de hardware. Todas las tareas tienen la misma prioridad, aunque si la tarea tiene una importancia crítica, puede bloquearse (ver procedimiento LockTask). Así la tarea puede decidir por si misma cuándo abandonar el control del procesador.

En condiciones normales, las tareas pueden conmutarse con cada pulsación del reloj. La frecuencia de las pulsaciones es 512 Hz, lo cual implica que si tiene 16 tareas ejecutándose a la vez en un bucle infinito, cada tarea se ejecutaría 32 veces por segundo. Las tareas bien diseñadas y con un buen esquema de cooperación traspasan su tiempo extra de procesamiento a otras tareas si están en un bucle a la espera de entrada de datos o eventos.

En el siguiente ejemplo, una tarea está esperando que el usuario presione un botón:

```
Do
    ' Leer el botón y salir si se presiona.
    If GetPin(5) = 0 then
        Exit Do
    End If

    ' Permitir que se ejecute la siguiente tarea.
    Call Sleep(0.0)
Loop
```

En esta aplicación en concreto, el botón pasa la mayor parte del tiempo sin hacer nada, lo cual implica que la CPU estaría desperdiciando tiempo sin necesidad si no tenemos esto en cuenta. En circunstancias ideales, la tarea debería comprobar el estado del botón, y después cambiar rápidamente a la siguiente tarea, que es la función de la llamada a *Sleep* (modo de espera)

Aunque el propósito de Sleep es principalmente para los retardos de tiempo; así mismo también se duplica como un mecanismo para permitir de manera explícita que se ejecute la siguiente tarea, a la vez que mantiene la tarea actual en la lista de tareas en ejecución. En concreto, puede llamar a la tarea Sleep con un retardo de tiempo cero para permitir la ejecución de otra tarea, pero volver inmediatamente si no hay lista ninguna otra tarea. En este caso un retardo cero podría volver en microsegundos.

Las tareas pueden eliminarse de la lista de ejecución por otras razones que no sea Sleep (modo de espera). Un ejemplo es enviar datos a un puerto serie. Si la cola de salida está completa y el programa intenta colocar otro carácter en la cola, entonces la tarea seguirá en modo de espera hasta que la cola se libere.

Así mismo, la red también puede hacer que una tarea entre en modo de espera al enviar un mensaje de red. En este caso la tarea entra en modo de espera hasta que se reciba una respuesta o se desconecta.

Tareas para interfaz de usuarios

Como ya se ha indicado anteriormente, la capacidad de multitarea puede ser ideal para implementar interfaces de usuarios. La capacidad multitarea facilita la combinación funciones predecibles y de fácil programación con eventos impredecibles como pulsadores, movimientos de joystick u otras interacciones humanas.

Hagamos un ejemplo con un termostato. El termostato compara la temperatura con un punto de referencia. Cuando la temperatura es inferior a esa referencia, la calefacción se encenderá. Si la temperatura es superior, entonces se apagará.

La función del termostato puede colocarse con su propia tarea sin tenerse que preocuparse en absoluto por interfaces de usuarios como botones o displays.

Ejemplo de termostato:

```
Dim Temperature As Single
Dim Setpoint As Single
Dim Burner As Boolean
'-------------------------------------------------
Sub HeaterControlTask()

    Do
        ' Comprobar sólo cada 30 segundos.
        Call Sleep(30.0)

        ' ¿Necesitamos calor?
        If Temperature < Setpoint Then

            ' Sí – encender calefacción.
            Burner = True

            Do
                ' Comprobar de nuevo 30 segundo.
                Call Sleep(30.0)

                ' ¿Es la temperatura dos grados inferior al punto de
                referencia?
                If Temperature >= Setpoint + 2.0 Then

                    ' Apagar la calefacción, salir y esperar a que
                    descienda la temperatura.
                    Burner = False
                    Exit Do
                End If
            Loop

        End If

    Loop
```

```
End Sub
```

Sintaxis de las tareas

Observe que la tarea HeaterControlTask tiene el mismo aspecto que un procedimiento normal. Es más, puede de hecho llamarse como un procedimiento y puede tratarse como una tarea en virtud de la forma de llamada. Esta cuestión de sintaxis se analizará más adelante al tratar el programa principal.

En este documento se utiliza la convención con la que los nombres de las tareas tienen la forma *Name*Task, en la que "Task" se usa como un sufijo. El compilador no requiere el sufijo – sólo es para facilitar la lectura del código.

A continuación crearemos un termostato más complejo y añadiremos tiempo de reajustes durante el día.

```
Dim MorningSetpoint As Single
Dim EveningSetpoint As Single
'-----------------------------------------------------
Sub SetbacksTask()

    Dim Hour As Byte, Minute As Byte, Second As Single

    MorningSetpoint = 75.0  ' Valores de referencia (mañana).
    EveningSetpoint = 65.0  ' Valores de referencia (tarde)
    Do
        ' Comprobar el reloj cada 30 segundos.
        Call Sleep(30.0)
        Call GetTime(Hour, Minute, Second)
        If (Hour = 5) And (Minute = 0) Then ' 5:00 AM?
            Setpoint = MorningSetpoint
        ElseIf (Hour = 19) And (Minute = 30) Then ' 7:30 PM?
            Setpoint = EveningSetpoint
        End If
    Loop
End Sub
```

Observe que el procedimiento del termostato no ha cambiado en absoluto. Las dos tareas comparten la variable de nivel de módulo llamado Setpoint (punto de referencia). Esto funciona dado que SetbacksTask es un productor de Setpoint y HeaterControlTask es un consumidor de Setpoint. (Es posible que surjan dificultades cuando haya múltiples productores de una variable; no obstante se tratará este tema en el apartado de los semáforos.)

Ahora para la interfaz de usuario – añadiremos cuatro botones llamados Up (Subir), Down (Bajar), Morning (Mañana) y Evening (Tarde):

```
Const UpButton As Byte = 36
Const DownButton As Byte = 37
Const MorningButton As Byte = 38
Const EveningButton As Byte = 39

Dim DisplayTemperature As Single
'-----------------------------------------------------
Sub ButtonHandlerTask()
```

```
    DisplayTemperature = 72.0  ' Display por defecto

    Do
        ' Permitir que se ejecuten otras tareas.
        Call Sleep(0.0)

        ' Leer botones y actuar consecuentemente.
        If GetButton(UpButton) Then
            DisplayTemperature = DisplayTemperature + 1.0
            Call UpdateTemperatureDisplay
        ElseIf GetButton(DownButton) Then
            DisplayTemperature = DisplayTemperature - 1.0
            Call UpdateTemperatureDisplay
        ElseIf GetButton(MorningButton) Then
            MorningSetpoint = DisplayTemperature
        ElseIf GetButton(EveningButton) Then
            EveningSetpoint = DisplayTemperature
        End If
    Loop

End Sub
```

Aún no se ha cambiado la tarea HeaterControlTask original ni las tareas de ajuste (SetbacksTask). Ahora tenemos un termostato con una interfaz de usuario sencilla, reajustes según la hora del día, y control de calefacción en poco más de una página de código. Ahora veremos el programa principal que es el que inicia todo el proceso:

```
' Asignar espacio a cada tarea.
Dim StackHeaterControl(1 To 32) As Byte
Dim StackSetbacks(1 To 32) As Byte
Dim StackButtonHandler(1 To 32) As Byte
'-----------------------------------------------------
Sub Main()

    CallTask "HeaterControlTask", StackHeaterControl

    CallTask "SetbacksTask", StackSetbacks

    CallTask "ButtonHandlerTask", StackButtonHandler

    ' No hacer nada y asignar todo el tiempo a las otras tareas.
    Do
        Call Sleep(120.0)
    Loop

End Sub
```

Como ya se ha mencionado anteriormente, necesitamos algún método para poder distinguir entre procedimientos y tareas. Conseguimos esto con la instrucción CallTask, que es como se consigue tratar HeaterControlTask, SetbacksTask y ButtonHandlerTask como tareas reales en lugar de como procedimientos normales.

Observe que un procedimiento tratado como una tarea no puede tener parámetros.

Cómo compartir datos con semáforos

Las variables compartidas pueden tener productores y consumidores. Por un lado, un *productor* se trata de una tarea que modifica los contenidos de una variable. Este puede ser la parte izquierda de la expresión de una asignación. También puede ser una llamada a un subprograma que modifica un argumento. Por otro lado, un *consumidor* se trata de una tarea que utiliza una variable compartida en la parte derecha de la expresión de una asignación o como un argumento para un subprograma.

En otras palabras, un productor escribe en una variable, y un consumidor lee la variable.

¿Por qué se usan semáforos?

Considere esta línea de código:

```
Setpoint = Setpoint + 1.0
```

Este código es tanto un productor (*Setpoint* =) como un consumidor (*Setpoint + 1.0*). Esta línea de código aparentemente inocua crea conflictos de multitareas que pueden causar un funcionamiento erróneo de un programa. ¿Por qué? Pues porque el compilador rompe el código en cuatro (o más) partes:

1. Obtener la variable llamada Setpoint y colocarla en la pila
2. Obtener la constante 1.0 y colocarla en la pila
3. Añadir los dos elementos en la pila y dejar la respuesta en la pila
4. Transferir el valor en la pila a la variable llamada Setpoint

¿Qué ocurre si otra tarea cambia el Setpoint durante los pasos 2, 3 ó 4 que aparecen anteriormente? Cuando la tarea original vuelve a ejecutarse, toma la ahora la copia obsoleta del Setpoint de la pila y la transfiere a la ubicación de memoria del Setpoint, sobrescribiendo su nuevo valor. Los esfuerzos de la otra tarea son inútiles.

¿Cómo podemos evitar esto? Pues utilizando un *semáforo*. El semáforo es un concepto antiguo tomados de los ferrocarriles – la finalidad de los semáforos es evitar que dos trenes ocupen la misma sección de la vía al mismo tiempo, por razones más que obvias. En computación, los semáforos pueden utilizarse para evitar que dos tareas utilicen la misma variable de manera simultánea.

Reglas para el uso de semáforos

Un semáforo es un indicador – un indicador físico, como en el caso de los ferrocarriles. Este indicador se eleva (verdadero/true) cuando alguien tiene el uso del semáforo, y desciende (falso/false) si está libre. Las reglas son las siguientes:

1. Para tomar un semáforo, éste debe estar sin utilizar (falso/false).
2. Debe tomarlo y marcarlo como utilizado (verdadero/true) en una operación indivisible.
3. Cuando haya terminado de usar un semáforo, debe marcarlo como no utilizado (falso/false), ya que de lo contrario el semáforo estaría marcado para un uso indefinido.

Siguiendo estas reglas, indicará a otras tareas que es propietario del semáforo, y que quien quiera que desee utilizarlo deberá esperar hasta que haya terminado de utilizarlo.

¿Cómo se implementan los semáforos?

La función booleana de un semáforo se obtiene a través del sistema operativo. Se deshabilitan las interrupciones mientras la función está ejecutándose. En el siguiente ejemplo, aparece la función escrita en Basic:

```
Function Semaphore(ByRef Flag As Boolean) As Boolean

    ' ¿Está disponible el indicador?
    If Not Flag Then

        ' Tomar posesión del indicador.
        Flag = True

        ' Informar a los demás de que lo tenemos.
        Semaphore = True
    Else
        ' Otra persona tiene el semáforo.
        Semaphore = False
    End If

End Function
```

Aplicaciones de los semáforos

El siguiente ejemplo utiliza un semáforo para proteger el punto de referencia (Setpoint). Le recomendamos que revise el ejemplo anterior del termostato:

```
Dim SetpointSemaphore As Boolean
'-----------------------------------------------------
Sub Task1()
    Do
        ' ¿Es nuestro?
        If Semaphore(SetpointSemaphore) Then
            ' Establecer cero el punto más bajo.
            If Setpoint > 0.0 Then
                Setpoint = Setpoint - 1.0
            End If
            ' Permitir a otros utilizar el semáforo.
            SetpointSemaphore = False
        End If
    Loop
End Sub
'-----------------------------------------------------
Sub Task2()
    Do
        ' ¿Es nuestro?
        If Semaphore(SetpointSemaphore) Then
            ' Establecer 90 como máximo.
            If Setpoint < 90.0 Then
                Setpoint = Setpoint + 1.0
            End If
            ' Permitir a otros utilizar el semáforo.
            SetpointSemaphore = False
        End If
    Loop
End Sub
```

Aunque utilizar semáforos es bastante más complicado que simplemente utilizar una variable, un semáforo se constituye como una forma sencilla de proteger datos normales. Una vez que se empieza a utilizarlos, los encontrará de un valor incalculable.

Puede utilizar semáforos para proteger bloques de variables, no simplemente una variable. Si tiene una matriz escrita o leída por su programa, entonces podrá utilizar un semáforo para proteger toda la matriz.

Los semáforos pueden proteger puertos de entrada/salida (I/O) o fuentes de datos de los puertos serie. Supongamos que desea enviar un mensaje de texto a través de un puerto serie para visualizar un dispositivo. No desea que su mensaje se interpole con los caracteres de otra tarea. Quiere que el mensaje finalice antes de que otras tareas envíen sus mensajes. Para solucionar este problema, puede crear un semáforo de puerto serie para bloquear el puerto para otras tareas hasta que la primera tarea acabe de utilizarlo.

Hay casos en los que no tiene datos que proteger y que simplemente tiene que disparar un evento. Un ejemplo es una condición de una alarma. Puede tener cinco tareas que podrían activar una alarma y no quiere perder ninguna de ellas. Un semáforo propiamente dicho podría ser la alarma. Cuando la tarea de la alarma termine de procesar el aviso de alarma, podría liberar el semáforo para que otra tarea pudiera establecer la condición de alarma más tarde.

Advertencia – Si utiliza múltiples semáforos, debería siempre utilizarlos en el mismo orden. Se puede producir una condición conocida como *interbloqueo* cuando una tarea establece un semáforo, una segunda tarea establece un semáforo diferente, y cada tarea espera que la otra libere su semáforo respectivo. Ninguna de las tareas puede liberar nada dado que ambas están esperando que la otra libere el semáforo. El programa se detiene completamente si no hay otras tareas.

Se produce una situación análoga cuando en una carretera de dos carriles hay un coche esperando para hacer un giro en el tráfico denso, y un segundo coche se encuentra en el carril contrario y desea hacer lo mismo pero está detrás del primer coche. Si hay demasiados coches, el tráfico se detendrá completamente ya que ningún coche podrá hacer el giro.

¿Cómo utilizar colas para compartir datos?

Un potente concepto integrado para compartir datos en el sistema BasicX es la *cola*. Una cola es un área de almacenamiento en el que los datos salen de un lado y entran en otro. Un caso análogo podría ser una cola de un banco – alguien entra en el banco, se pone en la fila y de manera gradual se va moviendo hasta ponerse en el primer puesto de la cola. Otro término para la cola es FIFO (First In First Out—Primero en entrar, Primero en salir).

Comunicaciones por puerto serie

Las colas son particularmente útiles como buffer de datos en comunicaciones a través de puerto serie. Cuando abre un puerto serie, en primer lugar se abre dos colas – una de entrada y otra de salida.

Los datos de entrada llegan al puerto y se colocan en la cola de entrada, y el programa procesa los datos en el momento que pueda. Si el programa está ocupado haciendo cualquier otra cosa, la cola simplemente se seguirá llenando hasta que el programa empiece a leer los datos. Mientras que la cola no se desborde, no se perderán los datos.

Del mismo modo, el programa puede poner los datos en la cola de salida y dedicarse a hacer otra cosa inmediatamente, mientras el proceso, en un segundo plano, da salida a los datos a través del puerto con el registro de tiempo adecuado.

Comunicación entre tareas

Las colas son también ideales a la hora de transmitir datos entre diferentes tareas, dado que la operación de cada tarea es infranqueable. En otras palabras, una vez que empieza la operación de la cola, no puede ejecutarse ninguna otra tarea hasta que finalice la operación.

Una o más tareas pueden llenar una cola y otras pueden vaciarla. Cuando la cola esté completa una tarea que intente añadir más datos se bloqueará hasta que haya espacio libre. Del mismo modo, si una cola está vacía, una tarea que intente leer la cola se bloqueará hasta que se añadan datos a la misma.

La cola de BasicX ha sido diseñada para no necesitar semáforos en la intercomunicación de tareas, aunque hay excepciones. Por ejemplo, si coloca numerosos datos en una cola y desea que ninguna otra tarea tome los datos de lado, puede que sea necesario el uso de un semáforo.

Almacenamiento de datos

Las colas son útiles incluso si no se está utilizando una comunicación por puerto serie o la capacidad multitarea. Las colas constituyen un método excelente para almacenar datos para acciones futuros.

¿Cómo utilizar una cola?

El primer paso es abrir la cola. Internamente, una cola se implementa como un buffer circular, y los indicadores para la cola se mantienen dentro de la propia cola. Al abrir la cola se inicializan los punteros.

La sobrecarga de los punteros internos requiere 9 bytes. Por ejemplo, si define una matriz de una cola de 20 bytes se dejan 11 bytes libres para datos.

Después de abrir la cola, puede llamar al procedimiento PutQueue para añadir datos, y al procedimiento GetQueue para extraer datos. Existen otros subprogramas disponibles para gestionar las colas.

Ejemplo 1 – ordenación de bytes

En este ejemplo se ilustra la ordenación de una matriz de 3 bytes copiada en una cola. Tenga en cuenta que los bytes están ordenados de manera que el elemento inferior (byte 1) se transfiere en primer lugar:

```
Sub Main()

    ' Permitir espacio para 3 bytes, más una sobrecarga de 9 bytes.
    Dim Queue(1 To 12) As Byte
    Dim A(1 To 3) As Byte
    Dim B As Byte
    Dim N As Byte

    Call OpenQueue(Queue, 12)

    ' Cargar matriz.
    For N = 1 To 3
```

```
        A(N) = N
    Next

    ' Enviar la matriz de 3 bytes a la cola.
    Call PutQueue(Queue, A, 3)

    ' Extraer bytes 1 por 1.
    For N = 1 To 3
        Call GetQueue(Queue, B, 1)
        Debug.Print CStr(B); ", ";   ' Prints 1, 2, 3,
    Next

End Sub
```

Ejemplo 2 – transferir matrices completas

Si utiliza la tarea PutQueue para transferir toda una matriz a una cola en una única operación, se mantiene el orden interno de bytes si utiliza posteriormente el procedimiento GetQueue para leer una matriz en una sola operación. Ejemplo:

```
Sub Main()

    Dim Queue(1 To 12) As Byte
    Dim N As Byte
    Dim A(1 To 3) As Byte
    Dim B(1 To 3) As Byte

    Call OpenQueue(Queue, 12)

    ' Cargar la matriz.
    For N = 1 To 3
        A(N) = N * 2
    Next

    ' Transferir toda la matriz.
    Call PutQueue(Queue, A, 3)

    ' Leer toda la matriz.
    Call GetQueue(Queue, B, 3)

    For N = 1 To 3
        Debug.Print CStr(B(N)); ", ";   ' Prints 2, 4, 6,
    Next

End Sub
```

Ejemplo 3 – cómo esquivar las reglas estrictas de escritura

En este ejemplo se indica cómo utilizar una cola para hacer frente a las reglas estrictas de escritura del lenguaje:

```
Dim Oven(1 To 50) As Byte
Dim Pi As Single
```

```
Dim Fridge(1 To 4) As Byte

Sub Main()

    Call OpenQueue(Oven, 50)
    Pi = 3.14159

    ' Poner algunas Pi en el horno (Oven).
    Call PutQueue(Oven, Pi, 4)

    ' Poner cuatro piezas de tamaño byte de Pi en la nevera (Fridge).
    Call GetQueue(Oven, Fridge, 4)

End Sub
```

Ejemplo 4 – comunicaciones entre tareas

El siguiente ejemplo utiliza una cola para permitir las comunicaciones entre tareas:

```
Dim Queue(1 To 32) As Byte
Dim DrainStack(1 To 40) As Byte
'-----------------------------------------------------
Sub Main()

    ' Iniciar la cola.
    Call OpenQueue(Queue, 32)
    CallTask "DrainQueueTask", DrainStack
    Call FillQueue

End Sub
'-----------------------------------------------------
Sub FillQueue()

    Dim k As Integer

    k = 0
    Do
        Call Sleep(1.0)
        k = k + 1

        ' Poner dos bytes (entero de 16 bits) en la cola.
        Call PutQueue(Queue, k, 2)
    Loop
End Sub
'-----------------------------------------------------
Sub DrainQueueTask()

    Dim j As Integer

    Do
           ' Obtener los dos bytes de la cola.
        Call GetQueue(Queue, j, 2)
    Loop
```

```
End Sub
```

Limitaciones

(1) Cada cola requiere 9 bytes de sobrecarga interna, por lo que necesitará una cola de 10 bytes como mínimo para enviar un solo byte de datos.

(2) No puede crear una cola que sea más pequeña que el elemento más grande que vaya a enviar. El motivo es que BasicX comprobará la cola antes de enviar cualquier dato para verificar que efectivamente puede realizar el envío de datos en una sola operación. Esto es necesario a fin de evitar conflictos con otras tareas que utilicen la cola.

(3) Si la tarea intenta enviar más datos de los que caben en la cola, la tarea se bloqueará de manera indefinida.

(4) Si transfiere toda una matriz a una cola en una sola operación, de manera que el elemento inferior se transfiere en primer lugar.

Reloj de tiempo real

El sistema operativo tiene un reloj de tiempo real (RTC) integrado que conserva automáticamente un registro de la fecha y hora. Se proporciona un grupo de llamadas del sistema para permitirle leer y configurar el reloj:

Llamadas del sistema:
GetDate – devuelve la fecha.
GetTime – devuelve la hora del día.
GetDayOfWeek – devuelve el día de la semana.
GetTimestamp – devuelve la fecha y la hora del día.
PutDate -- configura la fecha.
PutTime – configura la hora del día.
PutTimestamp – configura la fecha y la hora del día.
Timer – devuelve los segundos (puntos flotantes) desde medianoche.

Internamente, el reloj de tiempo real (RTC) está formado por un conjunto de registros internos que son
actualizados continuamente por el sistema operativo para conservar los registros de la fecha y la hora del día. El reloj tiene una frecuencia de marcación de 512 Hz.

Ejemplo – para encender el sistema de riego a las 9:00 PM todos los días:

```
Sub ScheduleIrrigation()

    Dim Hour As Byte, Minute As Byte, Second As Single

    Do
        ' Espera hasta las 21:00.
        Call GetTime(Hour, Minute, Second)
        If (Hour = 21) and (Minute = 0) Then
            Call TurnOnIrrigation
            Exit Sub
```

```
        End If
    Loop

End Sub
```

Posibles conflictos de los recursos del sistema

El sistema operativo BasicX le ofrece acceso a un número de recursos del sistema. Algunos de estos recursos pueden entrar en conflicto entre sí en determinadas condiciones.

Por ejemplo, el dispositivo Com1 utiliza el mismo hardware que la red integrada (sólo X-01), lo que significa que no debería utilizarse Com1 y la red en el mismo programa. En otras palabras, si desea utilizar Com1 para comunicaciones RS-232, entonces la red no estará disponible. De manera inversa, si utiliza la red, entonces será Com1 el que no estará disponible para las comunicaciones RS-232.

Otro ejemplo – interno en el chip de BasicX se encuentra un temporizador de hardware llamado Timer1. Este temporizador lo utiliza el dispositivo Com2 así como el procedimiento InputCapture. Si abre el Com2 como un puerto serie, es posible que el procedimiento InputCapture entre en conflicto con las comunicaciones serie. Si el dato serie llega al dispositivo Com2 mientras se está ejecutando el procedimiento InputCapture, es posible que los datos se pierdan. Del mismo modo, si transmite los datos desde el Com2, y llama al procedimiento InputCapture antes de que el buffer de salida haya terminado de transmitir, los datos de salida podrían ser ilegibles.

A continuación, se incluye una lista de recursos del sistema (llamadas del sistema inclusive) que podrían entrar en conflicto entre sí:

Com1
Com2/Com3
DACPin / PutDAC
InputCapture
Network
OutputCapture
RTC (Reloj de tiempo real)
Pin I/O Group (leer debajo)

Hay un grupo de llamadas del sistema que utilizan pines de entrada/salida (I/O) y desactivan las interrupciones a fin de cumplir los exigentes requisitos del control de tiempo. Estos se engloban dentro del grupo "Pin I/O Group," que se detalla a continuación:

Pin I/O Group

CountTransitions
FreqOut
PlaySound
PulseIn
PulseOut
RCTime

La siguiente tabla muestra los posibles conflictos entre los recursos del sistema utilizados en la misma tarea. La marca "X" significa que dos recursos podrían entrar en conflicto entre sí:

	Pin I/O Group	DACpin / PutDAC	Input Capture	Output Capture	Com2/3	RTC	Com1	Network
Pin I/O Group					X	X	X	X
DACPin/PutDAC					X		X	X
InputCapture					X			
OutputCapture					X			

Com2/Com3	X	X	X	X				
RTC	X							
Com1	X	X						X
Network	X	X					X	

Esta tabla trata únicamente de los recursos del sistema que se encuentran dentro de la misma tarea. Los conflictos de recursos entre múltiples tareas son mucho más difíciles de tratar y requieren por tanto un profundo conocimiento del sistema BasicX

Nota – si un par de recursos aparece en conflicto en la tabla anterior, esto no significa necesariamente que ambos recursos no pueden utilizarse nunca dentro de la misma tarea. Por ejemplo, considere Com2 y InputCapture. Si usa Com2 *sólo* para salida, y si garantiza que todos los bytes salen del buffer de salida del dispositivo Com2 antes de llamar al procedimiento InputCapture, entonces los dos recursos pueden utilizarse en la misma tarea.

Otro ejemplo: el reloj de tiempo real (RTC) también aparece en la lista como posible conflicto con el procedimiento PulseOut. Sin embargo, esto sólo es cierto si el procedimiento PulseOut genera un amplitud de pulsos comparable en tamaño al pulso del reloj del sistema (1,95 ms aprox.). Si garantiza que el procedimiento PulseOut siempre genera pulsos significativamente de menor tamaño, entonces no se produce ningún conflicto con el reloj de tiempo real (RTC).

Registros para funciones especiales

El sistema operativo le permite un acceso directo y al nivel inferior a los siguientes registros para funciones especiales. Estas funciones son comunes en todos los sistemas BasicX:

Registros comunes de BasicX

```
Tipo        Nombre      Descripción

Byte        ACSR        Registro de estado y control de comparador analógico
Byte        UBRR        Registro de tasa de baudios UART
Byte        UCR         Registro de control UART
Byte        USR         Registro de estado UART
Byte        UDR         Registro de datos I/O UART
Byte        SPCR        Registro de control SPI
Byte        SPSR        Registro de estado SPI
Byte        SPDR        Registro de datos I/O SPI
Byte        PIND        Pines de entrada, Puerto D
Byte        DDRD        Registro de dirección de datos, Puerto D
Byte        PORTD       Registro de datos, Puerto D
Byte        PINC        Pines de entrada, Puerto C
Byte        DDRC        Registro de dirección de datos, Puerto C
Byte        PORTC       Registro de datos, Puerto C
Byte        PINB        Pines de entrada, Puerto B
Byte        DDRB        Registro de dirección de datos, Puerto B
Byte        PORTB       Registro de datos, Puerto B
Byte        PINA        Pines de entrada, Puerto A
Byte        DDRA        Registro de dirección de datos, Puerto A
Byte        PORTA       Registro de datos, Puerto A
Byte        EECR        Registro de control EEPROM
Byte        EEDR        Registro de datos EEPROM
Byte        EEARL       Byte inferior del registro de dirección de EEPROM
Byte        EEARH       Byte superior del registro de dirección de EEPROM
Byte        WDTCR       Registro de control del temporizador guardián
Byte        ICR1L       Byte inferior de registro de captura de entrada T/C 1
Byte        ICR1H       Byte superior de registro de captura de entrada T/C 1
Byte        OCR1BL      Byte inferior de registro B comparativo de salida de
```

		Timer/Counter1
Byte	OCR1BH	Byte superior de registro B comparativo de salida de Timer/Counter1
Byte	OCR1AL	Byte inferior de registro A comparativo de salida de Timer/Counter1
Byte	OCR1AH	Byte superior de registro A comparativo de salida de Timer/Counter1
Byte	TCNT1L	Byte inferior de Timer/Counter1
Byte	TCNT1H	Byte superior de Timer/Counter1
Byte	TCCR1B	Registro de control B de Timer/Counter1
Byte	TCCR1A	Registro de control B de Timer/Counter1
Byte	TCNT0	Timer/Counter0 (8-bit)
Byte	TCCR0	Registro de control de Timer/Counter0
Byte	MCUCR	Registro general de control MCU
Byte	TIFR	Registro de indicador de interruptor de Timer/Counter
Byte	TIMSK	Registro de máscara de interruptor de Timer/Counter
Byte	GIFR	Registro de indicador de interruptor general
Byte	GIMSK	Registro de máscara de interruptor general
Byte	SPL	Indicador inferior de pila
Byte	SPH	Indicador superior de pila
Byte	SREG	Registro de estado

Registros BX-01

El BX-01 contiene este registro adicional:

Tipo	Nombre	Descripción
UnsignedInteger	RTCStopwatch	Registro de cronómetro del reloj de tiempo real

Registros BX-24 y BX-35

El BX-24 y BX-35 contiene estos registros adicionales:

Tipo	Nombre	Descripción
Byte	ADCL	Registro de datos inferior de ADC
Byte	ADCH	Registro de datos superior de ADC
Byte	ADCSR	Registro de estado y de control de ADC
Byte	ADMUX	Registro de selector de multiplexor de ADC
Byte	ASSR	Registro de estado modo asíncrono
Byte	OCR2	Registro comparativo de salida de Timer/Counter2
Byte	TCNT2	Timer/Counter2 (8-bit)
Byte	TCCR2	Registro de control de Timer/Counter2
Byte	MCUSR	Registro de estado general MCR

Todos los registros se tratan conceptualmente como propiedades de un objeto definido por el sistema llamado Register (Registro), y los identificadores toman la forma de Register.*Name.* Por ejemplo, el registro de cronómetro (stopwatch) en el sistema BX-01 es Register.RTCstopwatch.

Código de ejemplo:

```
Dim Time As New UnsignedInteger

' Leer el cronómetro, y después ponerlo a cero.
Time = Register.RTCStopwatch
Register.RTCStopwatch = 0
```

Con excepción del registro del cronómetro de BX-01, los demás registros están todos integrados en el procesador principal. El procesador de BX-01 se trata de un Atmel AT90S8515, mientras que los procesadores de BX-24 y BX-35 usan un Atmel AT90S8535.

Los registros están documentados en los ficheros AT90S4414_8515.pdf y AT90S_8535.pdf, incluidos en el CDROM de de instalación de BasicX. Si no dispone de un programa para visualizar ficheros PDF, podrá también encontrar una copia gratuita del software Adobe Acrobat Reader en el CDROM. Para instalarlo, ejecute el fichero de instalación setup.exe de la carpeta Adobe Acrobat, incluida en el CDROM de instalación de BasicX.

Puede encontrar información adicional acerca del procesador en la siguiente página web:

```
http://www.atmel.com/atmel/products/prod200.htm
```

Nota – Los registros Register.SPL y Register.SPH se utilizan como indicadores de pilas de 2-bytes que son internos del sistema operativo, por lo que no deberían confundirse con las pilas de tareas que utiliza un programa BasicX durante su ejecución. Cada pila de tarea utiliza su propio indicador de pila, que no tiene nada que ver con SPL o SPH.