Basic Express
Operating System Reference

Version 2.0

© 1998-2002 by NetMedia, Inc. All rights reserved.

Basic Express, BasicX, BX-01, BX-24 and BX-35 are trademarks of NetMedia, Inc.

Microsoft, Windows and Visual Basic are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Adobe and Acrobat are trademarks of Adobe Systems Incorporated.

2.00H

Contents

Persistent variables in BasicX

A new concept unique to BasicX is persistent variables. These are variables that, instead of residing in RAM, reside in EEPROM, which means they retain their values when power is removed.

Persistent variables have properties that are similar to normal variables. They can be used as arguments in expressions, subroutines or function calls. They can be assigned to like other variables.

Defining persistent variables

How do we define them? As an example, we first write a program using all RAM-based variables:

```
Dim MidnightTemp As Single ' RAM based variable

Sub LateNight()

    Dim Temp As Single

    Do
        ' Are we within 60 seconds after midnight?
        If (Timer < 60.0) Then

            ' Record temperature and quit.
            Call ReadTemperatureSensor(Temp)
            MidnightTemp = Temp
            Exit Sub
        End If

        ' Check every 45 seconds.
        Call Sleep(45.0)
    Loop

End Sub
```

Once the program is running normally, we take the following variable and change it from RAM-based:

```
Dim MidnightTemp As Single
```

to EEPROM-based:

```
Dim MidnightTemp As New PersistentSingle
```

Notice that the rest of the program does not change at all. Only the type of the variable changes. When the compiler recompiles the program, the correct code is generated to make everything happen in the background.

Persistent variables, by their nature, retain values after a power outage and are therefore ideal for things like setpoints, user-defined parameters, timeout values, security delays, motor start delays and staging of compressors. They're also convenient for data logging of alarm conditions, peak temperatures and timestamps. Imagination is your only limit when it comes to persistent variables.

Persistent variables can be initialized over the network, or you can run a special local program that is used solely for initialization. After that initialization process, your main program runs normally and makes use of the variables. The persistent variables can always be updated over the network or by a separate program later.

There are some limitations to keep in mind when using persistent variables:

Write cycle limits

There is a limit to how many times you can "write" or assign to a persistent variable. Typically the EEPROM inside a BasicX chip is guaranteed for 100 000 write cycles. This seems like forever if you're only updating a parameter every few days or so. On the other hand, if your program accidentally goes into a fast loop writing to a persistent variable, you might exceed the limit in just seconds.

Reading, however, is practically infinite. You can read a persistent variable just like any other variable without worrying about wearing it out.

Write time

A persistent variable takes much longer to write than a RAM-based variable. Each byte takes approximately 4 milliseconds to write. BasicX tries to use this time to best advantage through multitasking or other uses for the processor during this time. Depending on the program, however, this time delay may be significant.

Parameter passing

If you pass a persistent variable as a subprogram argument, it must be passed by value. You can't pass it by reference.

Module level declarations

All persistent variables must be declared in module-level code. They can't be used as local variables.

Known Bugs (BX-01 only)

If you write to a persistent variable, the operating system allows you to read the variable before the write operation is complete. In the following example, assume X is initially 1:

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

' Initial value of X is 1.
X = 3
Y = X
```

In the last statement, the value of Y may be 1 instead of 3. A workaround is to insert a delay of at least 4 ms after writing to a persistent variable.

This problem affects only BX-01 systems, not other BasicX systems.

Block data classes

Array initialization issues

One problem with conventional arrays is that there is no easy way to initialize them without incurring a performance penalty. Traditional Basic dialects use DATA statements for similar purposes, but DATA statements are awkward to use and are not VB compatible.

BasicX solves this problem by providing the following system-defined block data classes, which are the equivalent of initialized arrays stored in EEPROM memory:

1-Dimensional array classes (byte only):

```
ByteVectorData
ByteVectorDataRW
```

2-Dimensional array classes:

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

Block data classes are either read-only, or read-write. If the class name has an RW suffix, it is read-write. Otherwise it is read-only.

The class name prefix determines the data type of the object. VectorData classes are always Byte type. TableData classes can be Byte, Integer, Long or Single.

All block data objects must be declared at module level. Example object declarations:

```
Dim B As New ByteVectorData          ' 1D byte array, read-only.
Public BRW As New ByteVectorDataRW   ' 1D byte array, read-write.
Private S As New SingleTableData     ' 2D float array, read-only.
```

Source method

The Source method defines the data file from which an object gets its data. The file is read at compile time, then loaded into EEPROM at the same time the BasicX program is downloaded. Example code:

```
Call B.Source("ByteVector.txt")
Call BRW.Source("C:\Temperatures.dat") ' Full PC path name can be used.
Call S.Source("CalibrationCurve.dat")
```

The Source method must be called before reading or writing to the object's internal data. The Source argument must be a single string literal.

Each block data source file is a plain ASCII text file that contains a list of numbers. VectorData files should contain 1 number per row. TableData files can contain anywhere from 1 to 253 numbers per row, and each row must contain the same number of entries. Numbers are separated by either comma or space delimiters.

A block data object is treated as similar to an array, where the array dimensions are determined implicitly by the source file. For 1D objects, the number of rows determines the array dimensions. For 2D objects, the number of columns and rows determines the array dimensions.

Warning -- if you attempt to read or write to the Value property of a block data object before calling the Source method, results are undefined (see below for details of the Value property).

Value property

The Value property allows you to read the internal data stored in a block data object. In the case of read-write objects, you are also allowed to write to the property. The Value property is the default property, which means you refer to it implicitly.

A 1D block data object is treated as similar to a 1D array, where the index corresponds to the row number. Row numbering start at 1. In the following example, V(1) is the first element of the array:

```
' At module level.
Dim V As New ByteVectorDataRW  ' Object is read-write.
Dim B As Byte

    [...]

' Read the 1st element.
B = V(1)

' Write to the 3rd element.
V(3) = B + 5
```

A 2D block data object is treated as similar to a 2D array, where the first index corresponds to the column number and the second index is the row number. Column and row numbering starts at 1. In the following example, VRW(1, 1) is the first element of the array:

```
Dim VRW As New LongTableData  ' Object is read-only.
Dim L As Long

    [...]

' Read column 2, row 3.
L = VRW(2, 3)
```

Warning -- a block data object is similar to a persistent variable in regards to write cycle limitations and the amount of time it takes to write to the object (see the section on persistent variables for more details).

DataAddress property

The DataAddress property returns the starting EEPROM address of the object's internal data. The DataAddress property can also be used as the address parameter in system calls GetEEPROM and PutEEPROM.

DataAddress is type Long and is read-only. Example code:

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

    [...]

' This returns the starting EEPROM address of the object's
' internal data.
Addr = T.DataAddress

' These 2 statements are equivalent ways of reading the first
' element of T.
A1 = T(1, 1)
Call GetEEPROM(T.DataAddress, A2, 2)
' At this point A1 and A2 should be equal.
```

BasicX networking (BX-01 only)

Network hardware

The network contained within BasicX is one of its most powerful features. Multiple chips can be interconnected in networks limited only by physical limitations of cable length and electronics. The BasicX network addressing scheme allows for over 65 000 nodes per network. Each node on the network has the same access and priority as any other node on the network, allowing peer-to-peer communication from any node to any other node at any time.

Physically, the BasicX network is configured in a bus topology. A star configuration can be implemented with a BasicX hub. In the standard bus configuration BasicX chips are daisy-chained together. For short distances within a circuit board, within a common chassis, or a controlled space like a wiring harness for a robot or vehicle, BasicX chips can be interconnected using only one common wire and a single resistor per chip.

For long distances, BasicX chips are made to interconnect using a standard RS-485 transceiver chip. Depending on the transceiver used, a BasicX network can span thousands of feet and nodes. Without repeaters, typically 32 BasicX systems can be interconnected over network distances up to 1000 feet. A ground wire is also necessary and can be obtained from the building ground, chassis ground in an auto or equipment rack, or through another wire or shield in the cable bundle. Many customers choose a four wire system to send power and ground on two wires and the network on another two. Inexpensive Category 5 (CAT5) wire is preferred.

Network software

From the programmer's view, any BasicX variable can be set or received over the network. Data can even be received from systems that were never programmed to send data. How is this possible? This certainly sounds like a strange concept, but it works and is very powerful. The network in a BasicX chip is a completely separate system from the execution of code and runs as an independent task. Even if the Basic program has stopped, the network can still send and receive data from a halted BasicX chip.

Lets make a simple example to demonstrate the network:

```
' Program ChipA, node address 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
```

Program ChipA is not very complicated. Notice there's no code in this program to send data or receive commands. In fact, the counter will not count at all without some help since OkToCount starts out as false. When the BasicX chip is downloaded with code, the programmer defines whether the chip requires networking. If so, the programmer specifies a node address for the chip. Networking is then started before the main program is started. In this case let us assume 99 as the node address of the BasicX Chip executing ChipA.

Whenever a BasicX program is compiled, a map file is generated. The map file shows the location of all static (module-level) variables in RAM. All persistent variables are also included, with locations in EEPROM. The map file has the filename *ProgramName*.mpx.The map file is needed so that other chips can connect with the data they want. In our example, the map file looks something like this:

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

The naming convention is "ModuleName_VariableName". The module name is followed by the variable name, with an underscore separator. The module name may or may not be the same as the program name.

If the map file is included in another program's project, the project now knows where CounterA and OkToCount are located in program ChipA. If you edit and change the ChipA source code, the map file changes too, keeping each program up to date with the latest locations of variables. Of course the "other" program needs to be recompiled if variables are moved around, added or deleted. All you have to remember is the module name on the remote system, the name of the variable in the module, and the board address of the remote chip.

Now it is time to create program ChipB. This program will read and write data from the running program ChipA.

```
' Program ChipB, node address 86.
Dim MyCounter As Integer
Dim CountStart As Boolean
Dim Result As Byte
'-------------------------------------------------
Sub Main()

    ' It's OK to start the remote counter.
    CountStart = True

    ' Set the remote data to start the counter.
    Call PutNetwork(99, ChipA_OkToCount, CountStart, Result)

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

        ' Stop the counter above 1000.
        If Mycounter > 1000 Then
            CountStart = False
            Call PutNetwork( _
                99, ChipA_OkToCount, CountStart, Result)
            Exit Do
        End If
    Loop
End sub
```

This program starts the counting on ChipA by sending a boolean to the location OkToCount. At this point ChipB starts polling the counter, and when it's greater than 1000, it will cause the counting to stop by setting OkToCount to false. Due to the timing of the two systems, when ChipB has completed, the counter in ChipA will be much greater than 1000.

What have we learned with this simple experiment? In a few lines of code, we have two chips talking together and exchanging information. As we pointed out before, ChipA has no code to send or receive information from any other chip. In fact hundreds of chips could extract the value of the counter and turn on and off the counting process. ChipA does not need to know who they are or have any code to process their requests.

Let's try another example of a simple irrigation system. Program IRRG:

```
' Program IRRG, node address 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

            ' Turn on the valve.
            Call PutPin(5, bxOutputHigh)

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

            WaterON = False

            ' Turn off valve.
            Call PutPin(5, bxOutputLow)
        End If
    Loop

End Sub
```

In this simple example, we show the use of the real time clock to track time of day. At 10:30 AM every day, this program will open a valve, then close it 15 minutes later. It sets the value of WaterOn to be true when the valve is on. This way any other chip that wants to check the status of the water valve just needs the following statement:

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

And then check the value of RemoteWaterOn. It will reflect the current state of the valve.

Notice again that IRRG has no code to send the WaterOn variable to anyone. It just happens behind IRRG's back. What is great about this concept is that you can concentrate on getting the job done in the controller device and not worrying about the networking details. Of course the "master" needs to know more details about the network, but the "master" always has to anyway.

We will stay on this irrigation theme to demonstrate some of the other network concepts. We'll now make the system more complex so that we can change the times, days and lengths of watering. If we use persistent variables, then we can survive a power outage and come up again with the schedule intact. We will just demonstrate one valve and leave it for the reader to expand the system.

```
' Program IRRG2, node address 10.
Dim WaterDay(1 To 7) As New PersistentBoolean ' True = water today.
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) ' Sleep for 60 seconds.
            Next
            WaterOn = False
            Call PutPin(5, bxOutputLow)
        End If
    Loop
End Sub
```

Now we've made the controller more complex, but in this case the master can still tell if the water is on by using the same simple statements:

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

No matter how complicated we make the irrigation controller, as long as we keep the same use of the WaterOn variable, any member of the network can see what is going on.

The above examples are simple master/slave applications. But since a BasicX system can send data back and forth to any other BasicX system, you can also build peer-to-peer applications. These types of programs are more complicated to write because they require more handshaking and thought between cooperating systems. Here, flexibility becomes more important as systems become more complex, where you may have motors, servos and other actuators being controlled in response to inputs from temperature, pressure or similar sensors.

Broadcasting and groupcasting

You can use BasicX systems to broadcast data to all nodes on a network simultaneously. You can also divide nodes into groups, and "groupcast" to individual groups. You can have up to 254 groups on a network.

Network addresses are 16-bit unsigned integers. Special addresses are reserved for broadcasting and groupcasting.

Broadcasting -- if the network address is 65 535 (&HFFFF), then the message is a broadcast, and goes to all nodes simultaneously.

Groupcasting -- if the upper byte of the network address is 255 (&HFF) and the lower byte is other than 255, then the message is a groupcast, where the lower byte is the group address. In other words, a packet with a network address of &HFFxx is a groupcast to group "xx".

Broadcasting and groupcasting put restrictions on the network address of each BasicX system. For example, a node obviously can't have the address 65 535, because that address is reserved for broadcasting.

The rules are as follows:

(1) The upper byte of each address is not allowed to be 255 (&HFF), which means all node addresses are restricted to range 0 to 65 279 (&HFEFF).

(2) The group address of each node has a range of 0 to 254 (&HFE).

Multitasking in BasicX

One of the most powerful features in BasicX is its ability to have multiple tasks running at the same time. Multitasking allows complex programs to be simplified by dividing them into smaller, more manageable pieces. Each task can cooperate with other tasks or work on its own. It's like having multiple processors all working at the same time.

Timing issues

Obviously there are some limitations, the first being that there is only one real processor in the system. Timesharing the processor adds a certain amount of overhead that can slow down a program.

Even so, it's not unusual for an embedded system to spend most of its time waiting for user input, or waiting for a specific time of day. Or a system may do nothing but log data at infrequent intervals. In these cases, the benefits of faster code development and simpler maintenance may far outweigh the limitations of multitasking.

Task stacks

Multitasking programs typically need more RAM than programs with a single task. If you use two or more tasks, each task (other than the main program) needs its own explicit stack. The main program uses an implicit stack that is automatically allocated by the compiler. Each task stack is a byte array that must be located in module-level (static) code.

How big should a stack be? That's not an easy question to answer. Stacks are used to process executable code, such as subprogram calls, math expressions and data transfers. The memory required for these processes is difficult to predict in general, especially if you use recursion.

A simple task with no local variables, simple equations and comparisons, and no subprogram calls, could get away with a stack as small as 32 bytes. Other complex tasks might need more than 1000 bytes of stack space.

Measuring stack usage

You may need to empirically determine the stack use. One technique is to start with an oversize stack that is preloaded with known data. After you run the program, look at the data in the stack to see how much was used. (this assumes the task doesn't write back the same data you started with).

A rule of thumb is to take this result, add 20 % to that number and then run the program again. If the program works, reduce the margin to 10 %, which gives you a cushion if the program executes differently.

These are only guidelines, of course, and stack requirements can depend greatly on the design of the program. For example, if the program's behavior is highly sensitive to network traffic or interrupts, then stack space may depend heavily on external events.

Fortunately task stacks are active only when the task is active. These are *not* core processor stacks and will not be affected *directly* by interrupts, network or other traffic from the core processor.

With good planning, programs with literally hundreds of tasks can be handled by the BasicX chip. The only limitations are total processing power and memory for variables and stacks. As a test, NetMedia created a program with 1000 tasks to test the multitasking engine. The program worked -- not fast, but it worked.

Task switching

Tasks are timeshared on a first-come first-served basis, except for tasks triggered by hardware interrupts. All tasks have the same priority, although if a task is critically important, it can be locked (see procedure LockTask), which means the task itself can decide when to relinquish control of the processor.

Under normal conditions, tasks are switched every clock tick. The tick frequency is 512 Hz, which means that if you had 16 tasks all running in an infinite loop, each task would run 32 times a second. Well-designed and cooperating tasks pass on their extra processing time to other tasks if they're in a loop waiting for data or events.

In the following example, a task is waiting for a user to press a button:

```
Do
    ' Read the button and exit if it's pushed.
    If GetPin(5) = 0 then
        Exit Do
    End If

    ' Allow the next task to run.
    Call Sleep(0.0)
Loop
```

In this particular application, the button spends most of its time doing nothing, which means the CPU may needlessly waste time if we're not careful. Ideally, the task should check the state of the button, then rapidly switch to the next task, which is the purpose of the call to *Sleep*.

Although Sleep's purpose is primarily for time delays, it also doubles as a mechanism for explicitly allowing the next task to run, while keeping the current task on the list of running tasks. In particular, you can call Sleep with a zero time delay to allow another task to run, but return immediately if no other task is ready. In this case a sleep of zero could return in microseconds.

Tasks can be taken off the running list for reasons other than Sleep. One example is sending data to a serial port. If the output queue is full and the program tries to place another character in the queue, then the task will go to sleep until the queue becomes free.

The network can also cause a task to sleep when a network message is sent. In this case the task sleeps until a response is either received or timed out.

Tasks for user interfaces

As we alluded to earlier, multitasking can be ideal for implementing user interfaces. Multitasking makes it easier to combine predictable, easily-scheduled functions with unpredictable events like button pushes, joystick deflections or other human interactions.

Let’s make a thermostat as an example. The thermostat compares the temperature to a setpoint. When the temperature is below the setpoint, the heat is turned on. Otherwise the heat is turned off.

The thermostat function can be placed in its own task with absolutely no worries about user interface things like buttons or displays.

Thermostat example:

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

    Do
        ' Only check every 30 seconds.
        Call Sleep(30.0)

        ' Do we need heat?
        If Temperature < Setpoint Then

            ' Yes -- start the heat.
            Burner = True

            Do
                ' Check again in 30 seconds.
                Call Sleep(30.0)

                ' Is it two degrees warmer than the setpoint?
                If Temperature >= Setpoint + 2.0 Then

                    ' Turn off the heat, exit and wait for cooling.
                    Burner = False
                    Exit Do
                End If
            Loop

        End If

    Loop

End Sub
```

Task syntax

Notice that HeaterControlTask looks like an ordinary procedure. It can in fact be called like a procedure, but it can also be treated as a task by virtue of the way it is called. This is a syntax issue we'll explain when we get to the main program (see below).

In this document we use the convention that task names have the form *Name*Task, where "Task" is used as a suffix. The compiler does not require the suffix -- it's only there to make code easier to read.

Now lets make the thermostat more complicated and add time of day setbacks.

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

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

    MorningSetpoint = 75.0  ' Default morning setpoint.
    EveningSetpoint = 65.0  ' Default evening setpoint.
    Do
        ' Check the clock every 30 seconds.
        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
```

Notice that the thermostat procedure did not change at all. The two tasks are sharing the module-level variable called Setpoint. This works because SetbacksTask is a producer of Setpoint and HeaterControlTask is a consumer of Setpoint. (Difficulties can arise when there are multiple producers of a variable, but we'll address this condition when we discuss semaphores.)

Now for the user interface -- we'll add four buttons called Up, Down, Morning and Evening:

```
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  ' Default display.

    Do
        ' Allow other tasks to run.
        Call Sleep(0.0)

        ' Read buttons and take action as required.
        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
```

We still haven't changed the original HeaterControlTask, nor have we changed SetbacksTask. We now have a thermostat with a simple user interface, time of day setbacks, and heater control in not much more than a page of code. We haven't shown the main program that starts it all. Here it is:

```
' Allocate space to each task.
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

    ' Do nothing and give all time to other tasks.
    Do
        Call Sleep(120.0)
    Loop

End Sub
```

As we mentioned earlier, we need some way of distinguishing between procedures and tasks. We accomplish this with the CallTask instruction, which is how HeaterControlTask, SetbacksTask and ButtonHandlerTask are actually treated as tasks rather than normal procedures.

Note that a procedure treated as a task is not allowed to have parameters.

Sharing data with semaphores

Shared variables can have producers and consumers. A *producer* is a task that modifies the contents of a variable. This can be the left side of an assignment statement. Or it can be a subprogram call that modifies an argument. A *consumer* is a task that uses a shared variable on the right hand side of an assignment statement or as an argument to a subprogram.

In other words, a producer writes to the variable, and a consumer reads the variable.

Why semaphores?

Consider this line of code:

```
Setpoint = Setpoint + 1.0
```

This code is both a producer (*Setpoint =*) and a consumer (*Setpoint + 1.0*). This seemingly innocuous line of code creates multitasking headaches and can cause erratic operation of a program. Why? Because the compiler breaks the code into four (or more) pieces:

1. Get the variable called Setpoint and put it on the stack
2. Get the constant 1.0 and put it on the stack
3. Add the two items on the stack and leave the answer on the stack
4. Transfer the value on the stack to the variable called Setpoint

What happens if another task changes Setpoint during steps 2, 3 or 4 above? When the original task returns to run, it takes the now-obsolete stack copy of Setpoint and transfers it to Setpoint's memory location, overwriting it's new value. The efforts of the other task are useless.

How do we prevent this? By using the *semaphore*. The semaphore is an old railroad concept -- semaphores were used to keep two trains from occupying the same section of track at the same time, for obvious reasons. In computing, semaphores can be used to keep two tasks from using the same variable at the same time.

Rules for using semaphores

A semaphore is a flag -- a physical flag, in the case of railroads. It is raised (true) when someone has the use of the semaphore, and dropped (false) if it is free. These are the rules:

1. To take a semaphore it must first be unused (false).
2. You must take it and mark it used (true) in one indivisible operation.
3. When you are done with the semaphore, you must mark the semaphore unused (false), or else the semaphore will be marked in use indefinitely.

By following these rules, you signal to other tasks that you own the semaphore, and anybody else who wants it must wait until you're done.

How semaphores are implemented

The Boolean function Semaphore is supplied by the operating system. Interrupts are disabled while the function is executing. This is what it would look like if the function were written in Basic:

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

    ' Is the flag available?
    If Not Flag Then

        ' Take possession of the flag.
        Flag = True

        ' Tell the world we have it.
        Semaphore = True
    Else
        ' Someone else has the semaphore.
        Semaphore = False
    End If

End Function
```

Semaphore applications

The following example uses a semaphore to protect Setpoint (see the previous thermostat example):

```
Dim SetpointSemaphore As Boolean
'-------------------------------------------------
Sub Task1()
    Do
        ' Is it ours?
        If Semaphore(SetpointSemaphore) Then
            ' Make zero the lowest.
            If Setpoint > 0.0 Then
                Setpoint = Setpoint - 1.0
            End If
            ' Allow others to use the semaphone.
            SetpointSemaphore = False
        End If
    Loop
End Sub
'-------------------------------------------------
Sub Task2()
    Do
        ' Is it ours?
        If Semaphore(SetpointSemaphore) Then
            ' Make 90 the maximum.
            If Setpoint < 90.0 Then
                Setpoint = Setpoint + 1.0
            End If
            ' Allow others to use the semaphore.
            SetpointSemaphore = False
        End If
    Loop
End Sub
```

Although certainly more complicated than just using a variable, the semaphore is a simple way to protect common data. Once you start using them, you will find they're invaluable.

You can use semaphores to protect blocks of variables, not just one variable. If you have an array that your program writes to, or reads from, then you could use a semaphore to protect the entire array.

Semaphores can protect I/O ports or serial port data streams. Suppose you want to send a text message through a serial port to a display device? You don't want your message interleaved with characters from another task. You want the message to complete before other tasks send their messages. To solve this problem, you can create a serial port semaphore to block the port from other tasks until the first task is finished with it.

There are cases where you don't have any data to protect and you just need a way to trigger an event. An example is an alarm condition. You may have five tasks that could trigger an alarm and you don't want to miss any of them. The semaphore itself could be the alarm. When the alarm task is finished processing the alarm, it could free the semaphore so another task could set the alarm condition later.

Caution -- if you use multiple semaphores, you should always use them in the same order. A condition known as the *deadly embrace* can occur when one task sets a semaphore, a second task sets a different semaphore, and each task is waiting for the other to release its respective semaphore. Neither task can release anything because they're both waiting for the other. The program grinds to a halt if there are no other tasks.

An analogous situation is on a two-lane road where one car is waiting to turn across traffic, and a second car in the opposing lane wants to do the same thing but is behind the first car. If there are too many cars, traffic grinds to a halt because neither car can turn.

Using queues to share data

A powerful data sharing concept built into BasicX is the *queue*. A queue is a storage area where data goes in one side and out the other. An analogy would be a queue in a bank -- you enter a bank, get in line and eventually move to the front of the line. Another term for queue is FIFO (First In First Out).

Serial communications

Queues are particularly useful as data buffers in serial communications. When you open a serial port, you first open two queues -- one for input and one for output.

Input data arriving at the port are placed in the input queue, and the program processes the data whenever it can. If the program is busy doing something else, the queue just continues to fill until the program gets around to reading the data. As long as the queue doesn't overflow, you don't lose data.

Similarly, the program can put data in the output queue and go do something else immediately while background processing sends data out the port with the proper timing.

Communicating between tasks

Queues are also ideal for transmitting data between tasks, because each queue operation is unbreakable. In other words, once a queue operation starts, no other task is allowed to run until the operation is finished.

One or more tasks can fill a queue and other tasks can empty it. Whenever a queue becomes full, a task that tries to add more data will block until there is room. Similarly, if a queue is empty, a task trying to read the queue will block until data is added elsewhere.

The BasicX queue is designed so that semaphores are not usually needed for intertask communications, but there are exceptions. For example, if you place multiple pieces of data in a queue and want no other task to get data in edgewise, a semaphore may be required.

Data storage

Queues are useful even if you're not using serial communications or multitasking. Queues are a great way to store data for future action.

How to use a queue

The first step is to open the queue. Internally, a queue is implemented as a circular buffer, and pointers for the queue are maintained within the queue itself. Opening the queue initializes the pointers.

Internal pointer overhead requires 9 bytes. As an example, if you define a 20 byte queue array, that leaves 11 bytes available for data.

After opening the queue, you can call PutQueue to add data, and GetQueue to extract data. Several other bult-in subprograms are available for queue handling.

Example 1 – byte ordering

This example illustrates the ordering of a 3 byte array copied to a queue. Note that ordering is such that the low element (byte 1) is transferred first:

```
Sub Main()

    ' Allow room for 3 bytes, plus 9 byte overhead.
    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)

    ' Load array.
    For N = 1 To 3
        A(N) = N
    Next

    ' Send the 3 byte array to the queue.
    Call PutQueue(Queue, A, 3)

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

End Sub
```

Example 2 – transferring entire arrays

If you use PutQueue to transfer an entire array to a queue in a single operation, the internal byte ordering is preserved if you subsequently use GetQueue to read the array as a single operation. Example:

```
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)

    ' Load the array.
    For N = 1 To 3
        A(N) = N * 2
    Next

    ' Transfer the entire array.
    Call PutQueue(Queue, A, 3)

    ' Read the entire array.
    Call GetQueue(Queue, B, 3)

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

End Sub
```

Example 3 – sidestepping strong typing rules

This example illustrates how you can use a queue to get around the strong typing rules of the language:

```
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

    ' Put some Pi in the oven.
    Call PutQueue(Oven, Pi, 4)

    ' Put four byte-size pieces of Pi in the Fridge.
    Call GetQueue(Oven, Fridge, 4)

End Sub
```

Example 4 – intertask communications

The following example uses a queue for intertask communications:

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

    ' Start up the queue.
    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

        ' Put two bytes (16-bit integer) into the queue.
        Call PutQueue(Queue, k, 2)
    Loop
End Sub
'-----------------------------------------------
Sub DrainQueueTask()

    Dim j As Integer

    Do
        ' Get the two bytes from the queue.
        Call GetQueue(Queue, j, 2)
    Loop

End Sub
```

Limitations

(1) Each queue requires 9 bytes of internal overhead, so you need a minimum 10 byte queue to send a single data byte.

(2) You can't make a queue smaller than the largest data element to be sent. The reason is that BasicX will check the queue before it sends any data in order to verify that it can complete the sending of data in one operation. This is necessary to prevent conflicts with other tasks using the queue.

(3) If a task tries to send more data than will fit in a queue, the task will block indefinitely.

(4) If you transfer an entire array to a queue in 1 operation, the ordering is such that the low element is transferred first.

Real Time Clock

The operating system has a built-in Real Time Clock (RTC) that automatically keeps track of date and time. A group of system calls is provided to let you read or set the clock:

GetDate -- returns date.
GetTime -- returns time of day.
GetDayOfWeek -- returns day of week.
GetTimestamp -- returns date and time of day.
PutDate -- sets the date.
PutTime -- sets the time of day.
PutTimestamp -- sets the date and time of day.
Timer -- returns floating point seconds since midnight.

Internally, the RTC consists of a set of internal registers that are continually updated by the operating system to keep track of date and time. The clock has a tick frequency of 512 Hz.

Example -- to turn on irrigation at 9:00 PM every day:

```
Sub ScheduleIrrigation()

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

    Do
        ' Wait for 21:00.
        Call GetTime(Hour, Minute, Second)
        If (Hour = 21) and (Minute = 0) Then
            Call TurnOnIrrigation
            Exit Sub
        End If
    Loop

End Sub
```

Potential system resource conflicts

The BasicX operating system gives you access to a number of system resources. Some of these resources can conflict with each other under certain conditions.

For example, the Com1 device uses the same hardware as the built-in network (BX-01 only), which means Com1 and networking should not be used in the same program. In other words, if you want to use Com1 for RS-232 communications, then the network is not available. Conversely, if you use the network, then Com1 is not available for RS-232 communications.

Another example -- internal to the BasicX chip is a hardware timer called Timer1. The timer is used by the Com2 device as well as procedure InputCapture. If you open Com2 as a serial port, it is possible that InputCapture may conflict with serial communications. If serial data arrives at Com2 while InputCapture is executing, data may be lost. Similarly, if you transmit data from Com2, and call InputCapture before the output buffer has finished transmitting, the output data may be garbled.

The following is a list of system resources (including operating system calls) that may conflict with each other:

Com1
Com2/Com3
DACPin / PutDAC
InputCapture
Network
OutputCapture
RTC (Real Time Clock)
Pin I/O Group (see below)

There are several system calls involving pin I/O that turn off interrupts in order to meet stringent timing requirements. These are put in a general grouping called "Pin I/O Group," and are listed below:

Pin I/O Group

CountTransitions
FreqOut
PlaySound
PulseIn
PulseOut
RCTime

The following table illustrates potential conflicts between system resources that are used in the same task. An "X" entry means the two resources possibly conflict with each other:

	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	

This table addresses only system resources that are in the same task. Resource conflicts between multiple tasks are more difficult to address and require a thorough understanding of the BasicX system.

Note -- if a pair of resources appears to conflict in the above table, that doesn't necessarily mean the two resources can never be used in the same task. As an example, consider Com2 and InputCapture. If you use Com2 for output *only*, and if you insure that all bytes are flushed from Com2's output buffer before calling InputCapture, then the two resources can be used in the same task.

As another example, the real time clock is listed as conflicting with PulseOut. But this occurs only if PulseOut generates a pulsewidth comparable in size to the system clock tick (about 1.95 ms). If you insure that PulseOut always generates significantly shorter pulses, then there is no conflict with the real time clock.

Special purpose registers

The operating system allows you direct, low-level access to the following special purpose registers, which are common to all BasicX systems:

Common registers

Type	Name	Description
Byte	ACSR	Analog Comparator Control and Status Register
Byte	UBRR	UART Baud Rate Register
Byte	UCR	UART Control Register
Byte	USR	UART Status Register
Byte	UDR	UART I/O Data Register
Byte	SPCR	SPI Control Register
Byte	SPSR	SPI Status Register
Byte	SPDR	SPI I/O Data Register
Byte	PIND	Input Pins, Port D
Byte	DDRD	Data Direction Register, Port D
Byte	PORTD	Data Register, Port D
Byte	PINC	Input Pins, Port C
Byte	DDRC	Data Direction Register, Port C
Byte	PORTC	Data Register, Port C
Byte	PINB	Input Pins, Port B
Byte	DDRB	Data Direction Register, Port B
Byte	PORTB	Data Register, Port B
Byte	PINA	Input Pins, Port A
Byte	DDRA	Data Direction Register, Port A
Byte	PORTA	Data Register, Port A
Byte	EECR	EEPROM Control Register
Byte	EEDR	EEPROM Data Register
Byte	EEARL	EEPROM Address Register Low Byte
Byte	EEARH	EEPROM Address Register High Byte
Byte	WDTCR	Watchdog Timer Control Register
Byte	ICR1L	T/C 1 Input Capture Register Low Byte
Byte	ICR1H	T/C 1 Input Capture Register High Byte
Byte	OCR1BL	Timer/Counter1 Output Compare Register B Low Byte
Byte	OCR1BH	Timer/Counter1 Output Compare Register B High Byte
Byte	OCR1AL	Timer/Counter1 Output Compare Register A Low Byte
Byte	OCR1AH	Timer/Counter1 Output Compare Register A High Byte
Byte	TCNT1L	Timer/Counter1 Low Byte
Byte	TCNT1H	Timer/Counter1 High Byte
Byte	TCCR1B	Timer/Counter1 Control Register B
Byte	TCCR1A	Timer/Counter1 Control Register A
Byte	TCNT0	Timer/Counter0 (8-bit)
Byte	TCCR0	Timer/Counter0 Control Register
Byte	MCUCR	MCU general Control Register
Byte	TIFR	Timer/Counter Interrupt Flag register
Byte	TIMSK	Timer/Counter Interrupt MaSK register
Byte	GIFR	General Interrupt Flag Register
Byte	GIMSK	General Interrupt MaSK register
Byte	SPL	Stack Pointer Low
Byte	SPH	Stack Pointer High
Byte	SREG	Status REGister

BX-01 registers

The BX-01 contains this additional register:

Type	Name	Description
UnsignedInteger	RTCStopwatch	Real Time Clock stopwatch register

BX-24 and BX-35 registers

The BX-24 and BX-35 contains these additional registers:

Type	Name	Description
Byte	ADCL	ADC Data Register Low
Byte	ADCH	ADC Data Register High
Byte	ADCSR	ADC Control and Status Register
Byte	ADMUX	ADC Multiplexer Select Register
Byte	ASSR	Asynchronous Mode Status Register
Byte	OCR2	Timer/Counter2 Output Compare Register
Byte	TCNT2	Timer/Counter2 (8-bit)
Byte	TCCR2	Timer/Counter2 Control Register
Byte	MCUSR	MCR general Status Register

All registers are treated conceptually as properties of a system-defined object called Register, and identifiers take the form Register.*Name.* For instance, the stopwatch register in the BX-01 is Register.RTCstopwatch. Example code:

```
Dim Time As New UnsignedInteger

' Read the stopwatch, then clear it.
Time = Register.RTCStopwatch
Register.RTCStopwatch = 0
```

Except for the BX-01 stopwatch register, the other registers are built into the core processor. The BX-01 processor is an Atmel AT90S8515, and the BX-24 and BX-35 processors use an Atmel AT90S8535.

The registers are documented in files AT90S4414_8515.pdf and AT90S_8535.pdf, which are included in the BasicX installation. If you don't have a PDF reader, we provide a free copy of the Adobe Acrobat Reader program. To install Acrobat Reader, run the setup.exe file in the Adobe Acrobat folder, which can be found on the BasicX Setup CD.

Additional information about the processor can be found at this web site:

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

Note – Register.SPL and Register.SPH are used for a 2-byte stack pointer that is internal to the operating system, and should not be confused with task stacks that are used by a running BasicX program. Each task stack uses its own stack pointer, which has nothing to do with SPL or SPH.