AppKit:
Interfacing the CH1786 Modem to a BASIC Stamp II

Reprinted with permission from Nuts and Volts (http://www.nutsvolts.com) April 2000.

I remember my first modem.... What a beauty: a 300-baud brick of a device that plugged right into the back of my trusty Commodore 64. On its first day of operation, I used that modem to chat with my friend, Bruce, who lived about 20 miles away. We stayed up all almost night typing messages back and forth to each other. It was horribly inefficient – as far as communications goes – but it was just so darned cool; our computers could talk to each other. Not long thereafter we discovered uploading and downloading. The world had suddenly become a much brighter place.

Today, modems are ubiquitous and we take them for granted. Our access to the Internet is enabled with modems and yet, we hardly give it a thought. I write these articles in Texas and, using my modem, send them off to California for publication. The point is, our big blue world has been made quite a bit smaller by connecting devices with modems. So...how might we use one with a Stamp?

The BASIC Stamp is a great tool for data collection and control. It's ease of programming, I/O and built-in serial capabilities make it a great lab tool that can be connected to a host PC. But what if the Stamp data-collector and the PC are not located in the same room or, better yet, you want to talk to several remote devices?

As we've seen, it's really not a problem; all we need is a modem. There are applications all over the Internet the demonstrate connecting a modem to the Stamp. The problem, thus far, has been packaging. How does one neatly package a salvaged modem with the Stamp? It can be a bit tricky, especially if you want one nice, neat, *small* box in your remote location.

A California-based company called Cermetek (www.cermetek.com) has solved the packaging problem with their line of embedded modems. These modems are very small – not much bigger than a BS2. And, like the BS2, there configured as DIP (albeit 0.8") packages. The modem that we'll use in our application is the Cermetek CH1786, which is available in an App Kit directly from Parallax.

CH1786 Basics

For review, the purpose of a modem is to allow to computers to "talk" to each other over some form of remote connection; usually, but certainly not limited to, telephone lines. In its most basic form, this is accomplished by converting the serial stream of ones and zeros to tones that are compatible with telephone line standards. This is called modulation. The receiving modem converts these tones back to ones and zeros. This is called demodulation. Hence, the term, modem. It's a contraction for modulation/demodulation.

Phone modems also provide an important electrical interface to the telephone line. This is called a Data Access Arrangement (DAA). Keep in mind that the telephone companies are justifiably fussy about devices that are connected to their lines. The CH1786 includes an FCC approved DAA. Even with the approved DAA, it is recommended that protective devices be installed between the modem and the telephone jack. That will reduce the chance of a surge coming in from the telephone lines (pretty common occurrence in high-lightning areas) from damaging the CH1786. The nice thing about using the Cermetek modem is that you can put it into a commercial device without going through the rigors and considerable expense of an FCC evaluation.

Like the Stamp, modems are "smart" devices and will respond to commands. The CH1786 is no exception. The list of commands is referred to as the "AT" (for attention) command set and was originally developed by Hayes for its modems. All modem manufacturers have adopted the AT command set (with modifications). With the sophistication of current modems, the AT command set is very broad and, unfortunately, not always universal. Thankfully, we only need a few of the basic commands for our project.

Our Project: A Remote Temperature Monitor

Our demonstration project is a simple recording thermometer – a remote recording thermometer. It constantly scans a Dallas Semiconductor DS1620 digital thermometer and records the current, high and low values. We'll get the data from the Stamp by calling it with a terminal program.

When a call is detected by the modem, the line is answered and the temperature information is sent to the user's remote terminal. From the remote location, the user is able to refresh the temperature display and even reset the stored high and low temperature values.

To keep things simple, we'll use a general-purpose terminal program (such as Windows® Terminal) as our remote access. But keep in mind that your terminal program does not have to run on Windows®. That's one of the neat things about the remote contact part of this project: the user can be running any operating system. The only requirements of the terminal program are that it runs ANSI emulation and can dial into your Stamp project through the remote computer's phone modem.

About The Circuit

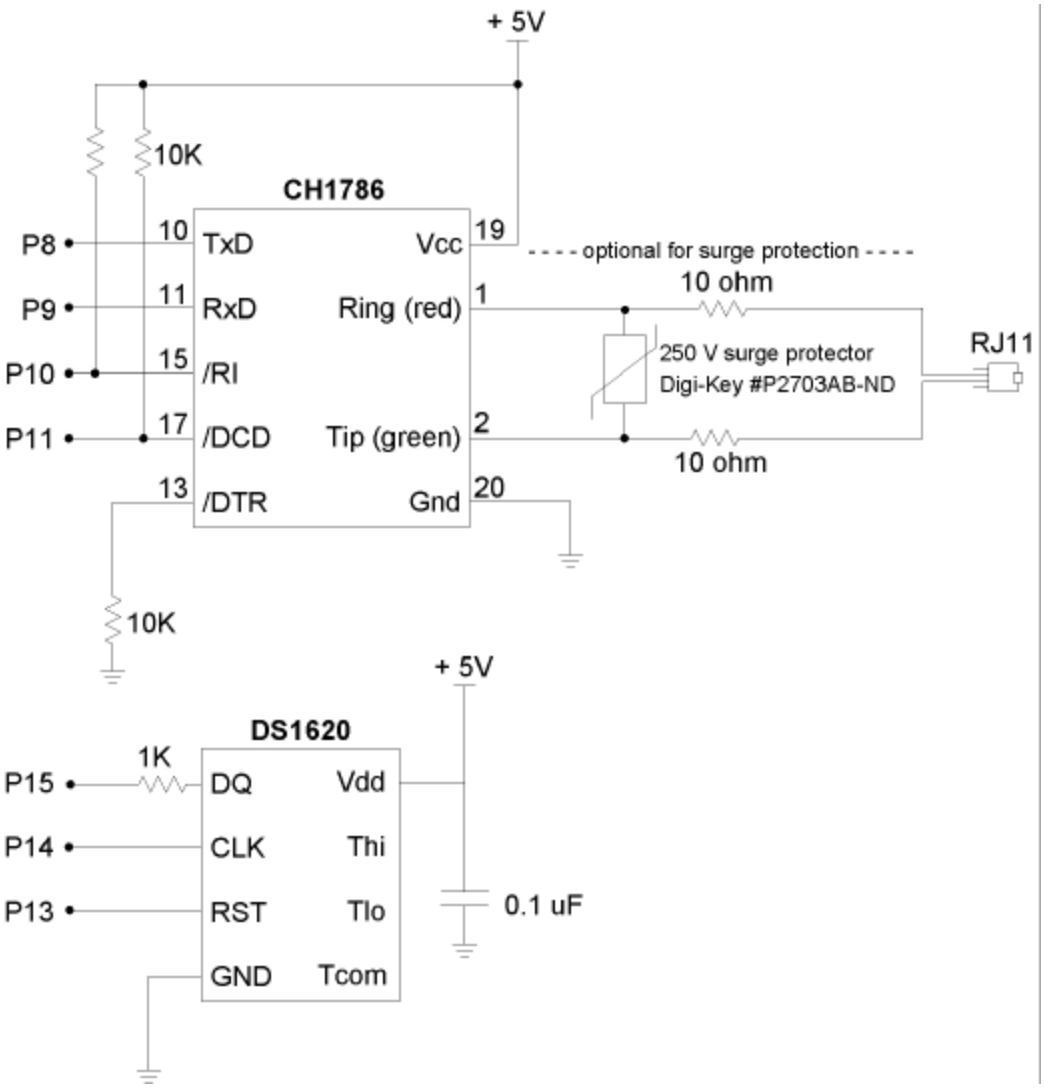
With my typical adherence to the "KISS" theory (Keep It Simple, Silly), the circuit is very straightforward. Four lines are used to connect the Stamp to the CH1786: TX (transmit), RX (receive), RI (ring indication) and DCD (data carrier detect). A quick scan through the code will show that we're not even using the Ring Indicator it's there for possible future use.

The reason is because the RI line does not go low and stay low during the ring, it actually pulses (low-high transitions) at a 20 to 30 Hertz rate (corresponds with the ringing sound and is probably there to allow your processor to detect distinctive ringing where available). The code includes a short sub-routine for dealing with the pulsing RI line should you want to use this input in your projects (more later).

So how will we know when the user is calling? The DCD line will tell us. This input goes low when the remote user has called in and the two modems are "hooked up." It's very convenient. You might wonder then, why b
Ring Indicator? One possible reason is to use it as a sort of warning. By detecting the Ring, critical processes could be finished before the modem answers and has to deal with the remote connection. You might also decide to ignore a call based on current conditions. In our case, we're going to tell the modem to answer the phone and let us know when everything is connected.

The DS1620 is connected with a standard 3-wire interface. The clock and data signals can be shared with other devices that use the same 3-wire bus.

Also notice that an external power supply is required. This is important. In operation, the modem draws more current than the Stamp can provide. A simple three-terminal regulator gives us the necessary 5 volts from a junk-box DC power supply.



The Software

The general structure of the software is a task switcher. In operation, a task switcher works like this:

do maintenance
do a task
do maintenance
do another task
...

This design allows the program to be broken up into small, manageable tasks that don't take a long time to run. This is important in our design since we want to keep tabs on the DCD indication. We'll put that in the *maintenance* section.

When using the task switcher design, there are two choices for updating the task control variable: in the maintenance section (before the **BRANCH** command) or individually within each task. The method used here is the latter; the task variable is updated by the current task. This allows the system to be more dynamic. For example a task might use the state of an input pin to determine which task will run next.

The error handler doesn't actually do anything except wait for one second, then jump back to the initialization section to try again. You could enhance this code by illuminating an error indicator. Just be sure to turn off your error indicator when the modem does initialize.

Most of modem's behavior is controlled by values stored in what are called "S registers." There are dozens or S registers. Thankfully, we only need to worry about two. The first, S0, allows the modem to answer the phone line if set to a value greater than zero. We'll use two.

carrier from the remote device once it's answered the phone. Thirty seconds should be long enough for the two devices to connect. Don't ignore this setting. I did while experimenting and found that when this register wasn't set, my project would never let go of the phone line when a call had been answered.

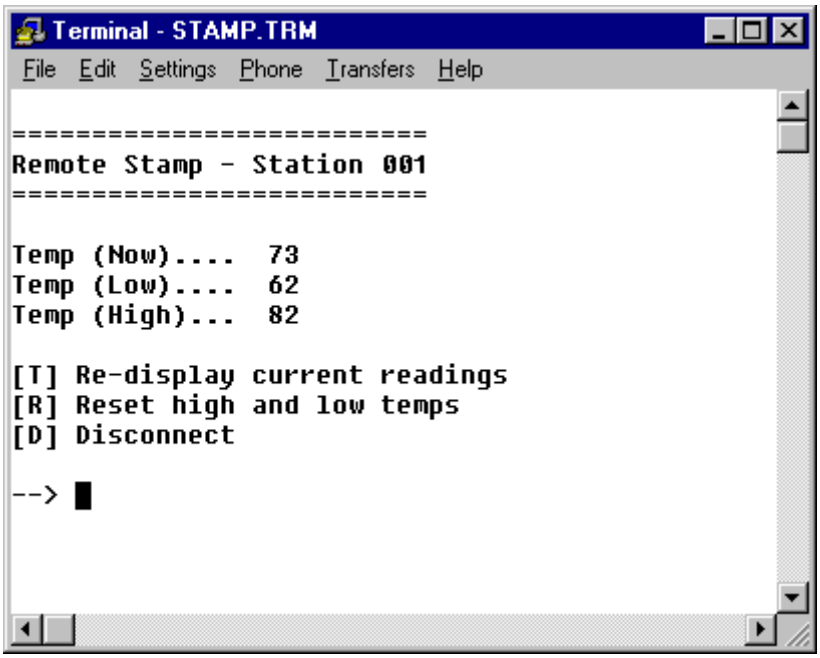
Once the modem is working, we'll fire-up the DS1620. Interestingly, it also communicates serially with the Stamp, but in this case it's synchronous serial communications. What this means is that the Stamp must provide a clock signal with the data. The BS2 commands **SHIFTOUT** and **SHIFTIN** take care of this very nicely.

The first thing we do with the DS1620 is tell it that we're writing to it. Once we've done that, we send the configuration byte. Our setup is for use with a CPU (that is, we're going to retrieve data from it) and to use free-run mode (continuously scan temperature). With our configuration byte sent, we have to release the DS1620 momentarily so that it can be stored in the DS1620's EEPROM. This takes about 10 milliseconds. With that done, we send it the command to start running. Pretty easy stuff.

Okay, now we're into the heart of the code. The first part (maintenance section) gets the current temperature and updates the stored values. Calling ScanT does this. This routine connects with the DS1620 and retrieves the current temperature. The temperature comes back as a 9-bit value that is actually expressed in 0.5 degrees (Celsius) increments. In our program, a call to **GetF** will take care of the conversion to Fahrenheit. If you want to stick with Celsius, change the call to **GetC**. Both of these routines deal with converting the DS1620 data to whole number values and setting the sign appropriately. Once we've got our current temperature, we update stored high and low values if necessary.

Next we check the DCD line. If it's low, the modem has answered and we can talk to the remote device. On detecting DCD, the program jumps to **GetMdm**. The first thing this does is pause a bit so that the other end can be ready. I found that some combinations of modems/terminal programs did not respond to the carrier detection as fast as the Stamp/CH1786. You may need to experiment with the **PAUSE** value.

With everything hooked up and ready the Stamp sends a menu to the remote computer. The menu displays our current and stored temperature readings. With the menu displayed, the program waits for the remote user to press a key.



The key is decoded with a **LOOKDOWN** command. This code is simple and very effective. The first thing we do is set the variable, *cmd*, to a known bad value. **LOOKDOWN** will convert a valid key (upper or lower case) to a value from zero to five. If the key is not valid, *cmd* will not be changed. Since we only have three things to do but have six possible *cmd* values, we divide command by two. This fixes it for the **BRANCH** command. If the key was bad, *cmd* will not work in the **BRANCH** command (its value exceeds the range of options) and the program will loop back to get another key.

If the remote user presses "t" or "T" the program will go get the current temperature and redisplay the menu.

If the user presses "r" or "R" the program will prompt the user for a confirmation before resetting the stored high and low temperatures. The confirmation is handled by a routine called **YesNo**. This routine will display the prompt, "Are you sure? (Y/N) : " and then wait for up to five seconds for the user to respond. If the user does not respond or presses a key other than "y" or "Y," the answer variable is set to No and the program resumes.

The menu choice of "d" or "D" (for disconnect from session) is handled in the same manner.

Extending the Project

There are several logical extensions to this simple project. The first would be the addition of a real-time-clock (i.e., DS1302) to record the time and date of the high and low temperature values. The Stamp might also collect the state of other inputs at the time of the temperature recordings. Finally, you might take advantage of the modem's ability to dial out by sending an emergency message to the user's pager (see below).

For advanced applications, a specialized terminal program could be developed. This would simply sending and receiving large amounts of data. Such a program could, for example, be setup to automatically poll the remote Stamps and record the readings to a hard disk. The RTCs in the remote units could be checked and automatically synchronized with the PC. The possibilities with a custom application are endless.

Sending Data to A Pager

Access to most numeric pagers is incredibly easy: dial the pager service, wait for the service to be ready, then punch in a set of numbers on the telephone keypad. This can be accomplished with a modem. The trick is to keep the modem in "command" (dialing) state so that DTMF digits can be sent to the pager service. You can do this by adding a semicolon to the end of the telephone number.

This bit of code will send "911" to a digital pager:

```
Pager:      SEROUT TX1, T2400, 10, ["ATDT123-456-7890;", CR]
            PAUSE 10000 ' let service answer and get ready
            ' send the numbers
            SEROUT TX1, T2400, 10, ["ATDT911", CR]
            PAUSE 5000 ' let number be dialed
            SEROUT TX1, T2400, ["+++"]
            PAUSE 2000
            SEROUT TX1, T2400, 10, ["ATH0", CR] ' hang up
            RETURN
```

Some pager services expect a terminating character (usually '#'). If this is the case with your service, simply add it to the message string:

```
SEROUT TX1, T2400, 10, ["ATDT911,#", CR]
```

The comma inserts a small delay (which is programmable in the modem through the S8 register) between the message numbers and the terminating character.

The code listing above is for example only; I don't suggest that you embed operational data this way. A better way to go would be to store your pager service number(s) and possible message in DATA statements. Then call a subroutine that points to the stored pager service number and message.

A final note on sending messages to pagers: Some paging services a finicky about DTMF dialing speed, that is, the duration of the each digit in the dial string. The CH1786 has a default dialing speed of 95 milliseconds for each digit. This should work for most systems. If you find, however, that your system is not being dialed correctly or digits are missing from the transmission string, you can adjust the dialing speed with the S11 register. To change the dialing speed to 125 milliseconds, you can add this line to the modem initialization section:

```
SEROUT TX1, T2400, 10, ["ATS11=125", CR]
```

Sending text messages to an alphanumeric pager is a substantially more involved procedure, so much so that it may not be possible to do with the current line of Stamps (I haven't tried myself). If you're interested, download the Telelocator Alphanumeric Protocol (TAP) from www.motorola.com.

Wrap Up

Just when you though **SERIN** and **SEROUT** across wires on your bench was cool, we've demonstrated that connecting to a remote Stamp across a telephone network is pretty easy too. And, with Cermetek's line of embedded modems, professionally packaging our project is not an issue. As an example of what can be done with a Stamp 2 and a Cermetek modem, take a look at www.hotwireftx.com.

So what's next? Well, I've got some ideas but I'd really rather hear from you. Feel free to put your computer's modem to work and send your suggestions via e-mail.

Resources:

Jon Williams
3718 Valley View Lane, #3040
Irving, TX 75062
(972) 659-9090
jonwms@aol.com

Parallax
599 Menlo Drive, Suite 100
Rocklin, CA 95756
(888) 512-1024
www.parallaxinc.com

Cermetek
406 Tasman Drive
Sunnyvale, CA 94089
(408) 752-5000 www.cermetek.com

```
' Nuts & Volts - Stamp Applications
' April 2000

' -----[ Title ]-----------------------------------------------------------
'
' File..... CERMETEK.BS2
' Purpose... Cermetek CH1786 demo program
' Author.... Jon Williams
' E-mail.... jonwms@aol.com


' -----[ Program Description ]---------------------------------------------
'
' This program monitors a Dallas Seminconductor DS1620 digital thermometer
' while waiting for an incoming call.  When a call is received, the Stamp
' causes the modem to answer the call then displays temperture data on
' the remote terminal.


' -----[ I/O Definitions ]--------------------------------------------------
'

' modem pins
'
TX1    CON    8                  ' transmit to modem
RX1    CON    9                  ' receive from modem
RI_    VAR    In10               ' ring indicator
DCD_   VAR    In11               ' carrier detect

' DS1620 pins
'
Rst    CON    13                 ' DS1620.3
Clk    CON    14                 ' DS1620.2
DQ     CON    15                 ' DS1620.1


' -----[ Constants ]-------------------------------------------------------
'
True   CON    1
False  CON    0

No     CON    1
Yes    CON    0

T2400  CON    396                ' 2400 baud for modem

LF     CON    10                 ' line feed character
FF     CON    12                 ' form feed (clear remote screen)


' DS1620 commands
'
RTmp   CON    $AA                ' read temperature
WTHi   CON    $01                ' write TH (high temp register)
WTLo   CON    $02                ' write TL (low temp register)
RTHi   CON    $A1                ' read TH
RTLo   CON    $A2                ' read TL
Strt   CON    $EE                ' start conversion
StpC   CON    $22                ' stop conversion
WCfg   CON    $0C                ' write configuration register
RCfg   CON    $AC                ' read configuration register


NTasks CON    3                  ' total number of tasks
```

```
' -----[ Variables ]------------------------------------------------------
'
tmpIn  VAR    Word               ' 9-bit temp input from DS1620
nFlag  VAR    tmpIn.Bit8         ' negative flag
hlfBit VAR    tmpIn.Bit0         ' half degree C bit
tempF  VAR    Word               ' converted fahrenheit value
tempC  VAR    Byte               ' converted celcius value
tmpNow VAR    Word               ' current temperature
tmpLo  VAR    Word               ' low temp
tmpHi  VAR    Word               ' high temp

sign   VAR    Byte               ' - for negative temps
sLo    VAR    Byte
sHi    VAR    Byte

inByte VAR    Byte               ' input from user terminal
cmd    VAR    Byte               ' command pointer
answer VAR    Byte               ' user response to prompt

task   VAR    Byte               ' task control variable

riFltr VAR    Byte               ' for ring indicator filter


' -----[ EEPROM Data ]------------------------------------------------------
'
' -----[ Initialization ]---------------------------------------------------
'
Init:  Dirs = %0110000100000000
       tmpLo = $FFFF             ' start with opposite extremes
       tmpHi = 0

I_Modm:      PAUSE 250                ' allow modem to power up

       ' train modem for speed
       '
       SEROUT TX1, T2400, 10, ["AT", CR]
       SERIN  RX1, T2400, 2500, Error, [WAIT ("OK")]
       PAUSE 250

       ' auto answer on second ring (S0=2)
       ' set max time for carrier detect to 30 secs (S7=30)
       '
       SEROUT TX1, T2400, 10, ["ATS0=2 S7=30", CR]
       SERIN  RX1, T2400, 2500, Error, [WAIT ("OK")]

I_1620:      HIGH Rst                 ' alert the DS1620
       SHIFTOUT DQ,Clk,LSBFIRST,[WCfg]  ' write configuration
       ' use with CPU; free run mode
       SHIFTOUT DQ,Clk,LSBFIRST,[%00000010]
       LOW Rst
       PAUSE 10                 ' pause for DS1620 EE write cycle
       HIGH Rst
       SHIFTOUT DQ,Clk,LSBFIRST,[Strt]  ' start temp conversions
       LOW Rst

NoDCD: IF DCD_ = Yes THEN NoDCD   ' make sure DCD is clear


' -----[ Main Code ]--------------------------------------------------------
'
Main:  GOSUB ScanT               ' get current temperature
       IF DCD_ = Yes THEN GetMdm  ' call received
```

```
       BRANCH task, [Task0, Task1, Task2]
       GOTO Main

Task0: ' task code here
       '
       task = 1                  ' select a specific task
       GOTO NextT                ' go do it

Task1: ' task code here
       '
       task = 2
       GOTO NextT

Task2: ' task code here
       '
       task = 0
       GOTO NextT

NextT: ' task = task + 1 // NTasks      ' round-robin to next task
       GOTO Main

       END

' -----[ Subroutines ]------------------------------------------------------
'
' ==============
' Modem Routines
' ==============

' error with modem
' - structured as seperate routine to allow user indications/enhancements
'
Error: ' additional code here
       PAUSE 1000
       GOTO I_Modm               ' try to initialize again


GetMdm:      PAUSE 5000                ' let other end get ready
Modm1: GOSUB DoMenu              ' show readings and menu
Get1:  SERIN RX1, T2400, [inByte] ' wait for input

       ' process user input
       cmd = 99
       ' convert letter to digit (0..5)
       LOOKDOWN inByte, ["tTrRdD"], cmd
       cmd = cmd / 2             ' fix for BRANCH
       ' branch to handler
       BRANCH cmd, [Cmd0, Cmd1, Cmd2]
       GOTO Modm1

Cmd0:  GOSUB ScanT               ' get current temp
       GOTO Modm1
Cmd1:  GOSUB RstT                ' reset high and low
       GOTO Modm1
Cmd2:  GOSUB Discon              ' disconnect from user
       IF answer = No THEN Modm1  ' stay with user
       GOTO NoDCD                ' back to the beginning


' clear remote terminal and display menu
'
DoMenu:      SEROUT TX1, T2400, [FF]
       SEROUT TX1, T2400, ["=========================", CR, LF]
       SEROUT TX1, T2400, ["Remote Stamp - Station 001", CR, LF]
```

```
       SEROUT TX1, T2400, ["=========================", CR, LF]
       SEROUT TX1, T2400, [LF]
       SEROUT TX1, T2400, ["Temp (Now).... ", sign, DEC tmpNow, CR, LF]
       SEROUT TX1, T2400, ["Temp (Low).... ", sLo,  DEC tmpLo, CR, LF]
       SEROUT TX1, T2400, ["Temp (High)... ", sHi,  DEC tmpHi, CR, LF]
       SEROUT TX1, T2400, [LF]
       SEROUT TX1, T2400, ["[T] Re-display current readings", CR, LF]
       SEROUT TX1, T2400, ["[R] Reset high and low temps", CR, LF]
       SEROUT TX1, T2400, ["[D] Disconnect", CR, LF]
       SEROUT TX1, T2400, [LF, "--> "]
       RETURN


' reset high and low temperatures
'
RstT:  SEROUT TX1, T2400, [CR, LF, LF, "Reset? "]
       GOSUB YesNo
       IF answer = No THEN RstX
       GOSUB ScanT
       tmpLo = tmpNow
       sLo = sign
       tmpHi = tmpNow
       sHi = sign
RstX:  RETURN

' disconnect
'
Discon:      SEROUT TX1, T2400, [CR, LF, LF, "Disconnect? "]
       GOSUB YesNo
       IF answer = No THEN DiscX
       SEROUT TX1, T2400, [CR, LF, LF, "Disconnecting.", CR, LF]

       ' return modem to command state
       '  and hang up
       '
       PAUSE 2000
       SEROUT TX1, T2400, ["+++"]
       PAUSE 2000
       SEROUT TX1, T2400, 10, ["ATH0", CR]
DiscX: RETURN


' confirm for [Y]es or [N]o
'   and get user input (default = No)
'
YesNo: SEROUT TX1, T2400, ["Are you sure? (Y/N) : "]
       answer = No

       ' get answer
       ' - but only wait for 5 seconds
       '
       SERIN RX1, T2400, 5000, YesNoX, [inByte]
       IF inByte = "y" THEN IsYes
       IF inByte = "Y" THEN IsYes
       GOTO YesNoX
IsYes: answer = Yes
YesNoX:      RETURN


' process ring indicator
' - filters pulsing ring indicator
' - waits for about 0.25 second of no RI pulsing before returning
'
DoRing:      ' your code here
       '  (i.e., count number of rings)
```

```
       '
RIWait:      riFltr = 0                ' clear the "no pulses" counter
RIchk: IF RI_ = Yes THEN RIWait   ' still pulsing
       riFltr = riFltr + 1        ' not pulsing, increment count
       IF riFltr > 50 THEN RIx          ' RI clear now
       PAUSE 5                          ' 5 ms between RI scans
       GOTO RIchk                 ' check again
RIx:   RETURN                     ' done - outta here


' ===============
' DS1620 Routines
' ===============

' get current temperature
' -- update high and low readings
'
ScanT: HIGH Rst                   ' alert the DS1620
       SHIFTOUT DQ,Clk,LSBFIRST,[RTmp]   ' read temperature
       SHIFTIN DQ,Clk,LSBPRE,[tmpIn\9]   ' get the temperature
       LOW Rst
       GOSUB GetF                 ' convert to Farhenheit
       tmpNow = tempF
       IF (tmpLo < tmpNow) THEN THigh
       tmpLo = tmpNow                   ' set new low
       sLo = sign
THigh: IF (tmpHi > tmpNow) THEN TDone
       tmpHi = tmpNow                   ' set new high
       sHi = sign
TDone: RETURN


' convert reading from 1/2 degrees input (rounds up)
'
GetC:  IF nFlag = 0 THEN CPos            ' check negative bit (8)
       sign = "-"                 ' set sign
       tempC = -tmpIn / 2 + hlfBit      ' if neg, take 2's compliment
       GOTO CDone
CPos:  sign = " "
       tempC = tmpIn / 2 + hlfBit
CDone: RETURN


' convert (1/2 degrees C) to Fahenheit with rounding
' -- general equation (for whole degrees): F = C * 9 / 5 + 32
'
GetF:  sign = " "
       IF nFlag = 0 THEN FPos1
       tmpIn = -tmpIn & $FF             ' convert from negative
       IF tmpIn < 36 THEN FPos0
FNeg:  sign = "-"
       tempF = tmpIn * 9 / 10 + hlfBit - 32
       GOTO FDone
FPos0: tempF = 32 - (tmpIn * 9 / 10 + hlfBit)
       GOTO FDone
FPos1: tempF = tmpIn * 9 / 10 + 32 + hlfBit
FDone:
RETURN
```