Column #72, April 2001 by Jon Williams:

Searching The 1-Wire™ Bus

A few months ago, Parallax introduced their newest Stamp, the BS2p microcontroller. I'm happy to report that, after a bit of a rocky production start (getting raw parts was tough), it's in full production now and doing very well. I really love the BS2p – it is definitely my favorite Stamp of the bunch.

Of its many neat new features, the BS2p has native support for Dallas 1-Wire™ components. Remember that 1-Wire™ components are uniquely identified by an eight-byte serial number. The first byte indicates the device (called the Family Code), the next six bytes are the serial number and finally, the last byte is a CRC of the first seven. This unique serial number allows up to 150 devices to coexist on a single-wire bus that can be up to 100 meters long – and those numbers grow with proper cabling and drive circuitry.

Working with a single 1-Wire™ device is easy since we can simply ignore its serial number (by using the SKIP ROM command). When two or more devices are on the bus, however, we need to know the serial number of each so that we can communicate with them individually. Issuing the SKIP ROM command on a bus with more than one device will cause data collisions and a big, garbled mess of data coming back to the Stamp.

Practicalities

When we're simply experimenting, individually reading the serial numbers from our 1-Wire™ devices and embedding them into our project code is no big deal. It would be a problem, however, if our project turned into a real product and we were going to manufacture a hundred units a day. Manually recording individual serial numbers just isn't practical; there's got to be a better way. So what do we do when we have multiple devices on the 1-Wire™ bus and we don't want to record the serial numbers manually?

Dallas Semiconductor knew this would be an issue in the design phase of the 1-Wire™ bus and implemented a command that is common to all 1-Wire™ parts that is called SEARCH ROM ($F0). With SEARCH ROM and a bit of code, we can individually identify any number (up to the limits of our EE storage space) of 1-Wire™ devices connected to the BS2p. The search algorithm code seems easy now, but trust me, it was a bear....

Get Ready For A KISS

For a really good description of the 1-Wire™ search algorithm, please download the *iButton Standards* documentation from Dallas Semiconductor. It was in this document that I found a description and flowchart that actually helped me implement the 1-Wire™ search algorithm. Here's my "*Keep It Simple, Stamp-Guy*" description of the process:

The 1-Wire™ bus is reset and the SEARCH ROM command is issued. Two bits are read from the bus (this process actually takes place 64 times – once for each bit in the serial number). Each 1-Wire™ device will return a bit, then its compliment. Since the design of the 1-Wire™ bus causes the output of the devices to logically AND with each other, four possible combinations of bits will be returned to the Stamp:

00 – bus conflict (some zeros retuned; some ones returned)
01 – all devices have a zero in this bit position
10 – all devices have a one in this bit position
11 – there are no devices present

The second two combinations are the easiest to deal with. We simply note the bit in our temporary serial number variable, then write the bit back to the bus. This keeps the devices online (this will make more sense in a moment).

When 00 is returned, there has been a conflict, that is, some devices have a zero in this bit position and some have a one (ANDing them together will return zero). In the beginning, the algorithm will arbitrarily select zero as the "hot bit" and write it back to the bus. When this happens, any device that just returned a one bit will be disabled until a new search is initiated. The bit location of the collision is saved.

As the search continues, a new collision location is compared with the last. If the new location is in the same spot as the last (this would only happen on a new device search), one is selected as the "hot bit" and the process continues. If the new location is greater than the last, a zero is used (because we're probably in the same device as the last collision). If the new location is less than the previous, the last known bit value is used.

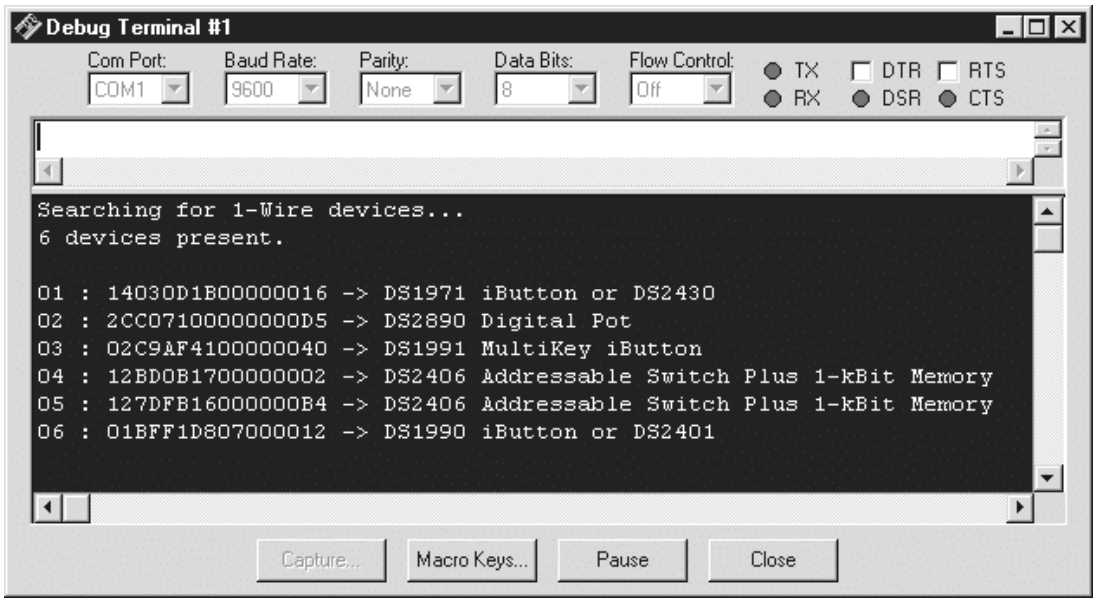
Yes, it can be a bit confusing and if anything like me, it may take a few days to sink in. The nice part is that it does work. As each device is identified and stored, it is taken offline. We know that we've found all the available devices when there are no reported bus collisions. If there are no devices connected to the bus, the pull-up will case 11 to be returned, indicating that there are no devices. You must be sure that your bus pin is pulled-up (through 4.7K). If it's not, 00 could be returned and the algorithm will attempt to search forever (I know this from having knocked a resistor loose in a late-night programming session).

Implementing The Search Algorithm

Have you ever been told not to believe anything unless you see it in writing? Me too. What I learned when it came to implementing the search algorithm for 1-Wire™ parts, is that not everything is writing is worth reading or helpful. Now, I'm not trying to knock Dallas Semiconductor. They're home-town guys, build cool parts and have been very nice to me. That said (or typed, as it were), I was a bit frazzled after staying up until 4 AM one morning trying to make the search algorithm work. I wasn't alone. A good friend and great BS2p programmer couldn't make it work either.

I felt just a bit better when I called one of Dallas' Applications Engineers later that morning (after a few hours of sorely needed sleep) and heard him refer to the 1-Wire™ search algorithm as a nightmare. Really. It's not easy. The good news for you is that after finding a third document on the algorithm, I was able to implement it successfully. So, the hard work is done and as you'll see in just a bit, you can easily integrate the 1-Wire™ search algorithm in your own BS2p programs. As I indicated earlier, the *iButton Standards* manual had the right combination of text and flowchart to make sense of the algorithm. After spending a long sleepless night, the right docs helped me get the code working in under an hour.

Figure 72.1: Searching the 1-wire bus debug screen



The Code In All Its Parts

Last month we talked about using extra program slots to hold additional code and data. This month we're going to put that theory to practical use with our project.

Remember me mentioning – okay, harping on -- program design; you know, knowing what your program is supposed to do before you actually write it? Well, here's what I wanted to do this month:

- Check to see if the 1-Wire™ bus has been searched for devices
- If not, search the bus (and use an external program for portability)
- If a device is found and the CRC is good, write the serial number into the caller's EE space
- If there are any CRC errors, ignore the bad device and report errors to the calling program
- When the bus has been searched, display the number and type of devices found

I broke the project into three separate files since I wanted to reuse the search algorithm and I didn't want to burden my main project file with all the string data that describes the

devices. What we'll do here is take advantage of information passing between programs (using PUT and GET) as well as the new BS2p command, STORE, that allows one program to READ or WRITE the EEPROM from another.

The main project file (Program Listing 72.1) is called SRCHDEMO.BSP. At the top you'll see the $STAMP directive that causes the support files, OWSEARCH.BSP and DSNAMES.BSP, to be opened, syntax-checked and downloaded to the BS2p.

Let's get back to the main program. After defining useful constants and the variables we need, EEPROM space is allocated for possible 1-Wire™ devices. There is a location labeled Num_OW that is initialized to $FF to indicate that the bus has not been searched. I'm hedging here that I will never need to have 255 devices on my 1-Wire™ bus. When the search routine is complete, this location will be updated with the number of devices found. Keep in mind that the value of $FF is written to this location only on download. If it is modified later, the program will run with the new information.

At the next location, 80 bytes are set aside for serial number storage. At eight bytes per device, this is enough room for ten devices. Of course, more could be allocated. As your program grows, be sure to use the Memory Map facility of the compiler to make sure you'll have EEPROM space for data that you want to store.

Okay, let's move on to initialization. The first thing we do is look at that location called Num_OW. The first time this program runs we will find a value of $FF (put there during download). This indicates the necessity of a search. Before we run the search, we'll grab our own calling program (just in case we do an update and are not in slot 0). This information, the pin number for the 1-Wire™ bus and the starting location for serial number storage are passed to the search program using PUT. The search program is launched with the RUN command.

The search program (OWSEARCH.BSP – Program Listing 72.2) initializes by grabbing its own slot number, the calling program's slot number, the 1-Wire™ bus pin and the start of data storage in the caller. Then the search begins. When a device is located, the CRC is checked to make sure that what we got back was, in fact, valid data. The CRC algorithm is also described in the Dallas documentation and, as you can plainly see, is very easy to implement.

The CRC can be calculated manually, bit-by-bit, but it's much easier to use a table-drive algorithm when there is storage space available. We have plenty of space, so the CRC table is stored in this slot. If the CRC check is good (the process returns a zero when

good), the device count is incremented and the serial number is written in the caller's EEPROM space.

This is possible only in the BS2p with the new STORE command. Now you can see why we have to keep track of our own program slot as well as the caller's. When reading the CRC table, we have to use our own slot number; when writing serial numbers to the caller, we have to point to the caller first.

When the searching is complete, the number of devices found is written in the caller's EEPROM (just ahead of the serial number table) and any CRC errors are passed back through the scratchpad with PUT. Then we return to the caller which, in our case, will display a list of any found devices.

When the RUN command is issued, the slot is run. The difference is that any EEPROM changes made by code are maintained. So, the search demo code will have different values in its EEPROM than when we downloaded it. At the very least, the number of devices reported will be something other than $FF.

The rest of the demo program is very simple: Errors are reported and then the found devices, if any, are displayed (serial numbers and device types). Since I didn't want to burden my main program with the device name strings, I used another module (DSNAMES.BSP – Program Listing 72.3) to hold them. When you look at this listing you'll see that it consists of one constant definition and a whole lot of EEPROM storage. The trick was to devise a way of finding the description string for a given device.

There's probably a dozen good ways to do it and here's what I decided to do: The string descriptions are stored first, with each string being identified with a useful label, in this case, the device part number. After the strings, pointers to their starting locations are stored. This is done mathematically – and automatically – thanks to the nature of the Stamp compiler.

The Stamp compiler converts DATA labels into numbers that can be used in math expressions. So, the formula for the storage address of the string pointers is:

(device id * 2) + base address

You'll notice that I use the Word directive to cause the Stamp to store two bytes at the calculated pointer location. This is necessary since the string definitions require far more than 255 bytes of EEPROM, hence, will require two bytes for addressing. So the pointer location stores the EEPROM location of the first character its respective string. Each

string is terminated with a zero so we can find its end. The base address was set to $600 as this is the highest starting location we can use with a possible device number of $FF. This allows string storage all the way up to $599 (1536 bytes).

A description is displayed by sending the device number to the subroutine called DisplayDeviceType. This code uses the math previously described to calculate the pointer to the device string. Two reads are required to get the starting address of the string. Once we have the starting address, we read a character, display it and then increment the address until a zero is located. You'll probably recognize this technique as we've used it many times before. The difference here is the ability to use different program slots to store the strings. This is a great opportunity to create a multilingual application – with different languages being stored in different program slots. Change the slot number to read from and you can change the language of your displays.

Okay, that's enough for this month. Seemingly simple code, but lots of good stuff going on with it and a pretty good demonstration of the usefulness of the new STORE command. I suggest you spend some time studying what we've done here – there are a lot of good code bits. And you can see why I like the BS2p so much. With the ability to handle 1-Wire™ and I2C devices easily combined with the reading and writing across EEPROM banks, it's a winner.

Does This Make Any Sense?

I was thinking the other day and it occurred to me that this column has been around for a few years now. First there was Scott, then me, then Lon – now back to me. Between the three of us, we've shared a lot of good tricks.

Don't worry, we're not going away, just considering what to do next to keep you interested and entertained. A friend recently suggested that spend a bit of time on inexpensive sensors and how to integrate them into Stamp projects. So that's what we'll do – for a while, anyway.

My buddies at the DPRG (Dallas Personal Robotics Group) gave me a nifty little light sensor (light to frequency) from TAOS: the TSL230. The great news is that it's easy to use and will work with any Stamp 2. But that's not the only part we'll be working with next month, so come back and join me.

As always, I wish you Happy Stamping.

```
' Program Listing 72.1
' Nuts & Volts - Stamp Applications, April 2001

' {$STAMP BS2p,OWSEARCH.BSP,DSNAMES.BSP}

' ----[ Title ]---------------------------------------------------------
'
' File..... SRCHDEMO.BSP
' Purpose... Demonstrates Dallas 1-Wire Search Algorithm
' Author.... Jon Williams
' E-mail.... jonwms@aol.com
' Started... 27 FEB 2001
' Updated... 05 MAR 2001


' ----[ Program Description ]-------------------------------------------
'
' The purpose of this program is to demonstrate the use of external
' program slots to hold code code or subroutines.  When first downloaded
' to the BS2p, the number of Dallas 1-Wire devices is unknown (flag value
' of $FF).  This program reads that flag and if it is $FF, the search
' program slot is called to conduct the search.  Search results are
' written directly to the this program's EE (thanks to the BS2p STORE
' function).
'
' When the search is complete, a list of all found devices is displayed
' in a DEBUG window.


' ----[ Revision History ]----------------------------------------------
'
' 28 FEB 2001 - Version 1 complete and tested


' ----[ I/O Definitions ]-----------------------------------------------
'
OWpin          CON    15             ' 1-Wire bus


' ----[ Constants ]-----------------------------------------------------
'
SearchPgm      CON    1              ' search program slot
NamesPgm       CON    2              ' DalSemi OW device names

' 1-Wire Support
'
OW_FERst       CON    %0001          ' Front-End Reset
OW_BERst       CON    %0010          ' Back-End Reset
OW_BitMode     CON    %0100
OW_HighSpd     CON    %1000
```

```
ReadROM         CON     $33             ' read ID, serial num, CRC
MatchROM        CON     $55             ' look for specific device
SkipROM         CON     $CC             ' skip ROM (one device)
SearchROM       CON     $F0             ' search


' ----[ Variables ]------------------------------------------------------
'
thisSlot        VAR     Nib             ' this program's slot (0 - 7)
devices         VAR     Byte            ' 1-Wire devices found
addr            VAR     Word            ' EE address of device SN
offset          VAR     Nib             ' offset byte of device SN
idx             VAR     Byte            ' loop counter
dByte           VAR     Byte            ' data byte
devType         VAR     Byte            ' device type (first byte of SN)
devName         VAR     Byte            ' device identifier
strPtr          VAR     Word            ' string pointer
char            VAR     Byte            ' string character to print
crcErrors       VAR     Byte            ' indicates problems with search

tempROM VAR     Byte(8)                 ' data from 1-Wire device
crcVal  VAR     Byte                    ' CRC of returned data


' ----[ EEPROM Data ]----------------------------------------------------
'
Num_OW  DATA    $FF                     ' number of 1-Wire devices
OW_Data DATA    0(80)                   ' space for 10 1-Wire SN's


' ----[ Initialization ]-------------------------------------------------
'
Initialize:
  READ Num_OW,dByte                     ' ROM codes present?
  IF (dByte < $FF) THEN Main            ' yes, run this program

  DEBUG CLS
  PAUSE 5
  DEBUG "Searching for 1-Wire devices...",CR

  GET 127,thisSlot                      ' get this pgm slot #
  PUT 126,thisSlot                      ' save it
  PUT 125,OWpin                         ' save OW I/O
  PUT 124,OW_Data                       ' save data storage start
  PUT 123,0                             ' clear CRC errors

  RUN SearchPgm                         ' run the search program


' ----[ Main Code ]------------------------------------------------------
```

```
'
Main:
  GET 123,crcErrors
  IF (crcErrors = 0) THEN ShowDevices
  DEBUG "Warning: CRC errors during search.",CR

ShowDevices:
  READ Num_OW,devices               ' get number of devices
  DEBUG CR,DEC devices," device"
  IF (devices = 1) THEN SkipEss
  DEBUG "s"

SkipEss:
  DEBUG " present.",CR
  IF devices = 0 THEN Done

  FOR idx = 1 TO devices            ' display each one
    DEBUG CR, DEC2 idx," : "
    addr = 8 * (idx - 1) + OW_Data  ' calculate device address
    STORE thisSlot                  ' point to local EEPROM
    FOR offset = 0 TO 7             ' eight bytes per device
      READ (addr + offset),dByte    ' read it from EEPROM
      DEBUG HEX2 dByte              ' show it
    NEXT
    READ addr,devType               ' get device type
    GOSUB DisplayDeviceType         ' display it
  NEXT

Done:
  DEBUG CR
  END


' ----[ Subroutines ]------------------------------------------------------
'
' This subroutine is used to display the part number and description of
' a connected device.  The text data and pointers to it are stored in the
' EE of a different program slot.

DisplayDeviceType:
  DEBUG " -> "
  addr = devType * 2 + $600         ' calulate string pointer addr
  STORE NamesPgm                    ' point to names EEPROM
  READ addr,strPtr.LowByte          ' get the string location
  READ addr+1,strPtr.HighByte

ReadAChar:
  READ strPtr,char                  ' read character from string
  IF (char = 0) THEN DevTypeDone    ' at end? (0 = Yes)
  DEBUG char                        ' no, print the char
  strPtr = strPtr + 1               ' point to next char
```

```
  GOTO ReadAChar

DevTypeDone:
  RETURN
```

```
' Program Listing 72.2
' Nuts & Volts - Stamp Applications, April 2001


' {$STAMP BS2p}

' ----[ Title ]-----------------------------------------------------------
'
' File...... OWSEARCH.BSP
' Purpose... Searches 1-Wire bus for device ROM codes
' Author.... Jon Williams
' E-mail.... jonwms@aol.com
' Started... 26 FEB 2001
' Updated... 05 MAR 2001


' ----[ Program Description ]---------------------------------------------
'
' This program will search for all connected Dallas 1-Wire devices. The
' calling program passes its slot number, the 1-Wire buss pin and the
' start of EEPROM space for the storage of Dallas 1-Wire serail numbers.
'
' If a device is found, the 8-byte serial number is recorded in the
' EEPROM of the calling program slot.  The search will continue until all
' devices have been found.  The program will then jump back to the caller.
'
' REFERENCE: iButton Standards documentation, ROM Search Figure 5-3
'            Dallas Semiconductor
'            www.dalsemi.com


' ----[ Revision History ]------------------------------------------------
'
' 28 FEB 2001 - Tested and working per Dallas specification


' ----[ Constants ]-------------------------------------------------------
'
' 1-Wire Support
'
OW_FERst       CON     %0001          ' Front-End Reset
OW_BERst       CON     %0010          ' Back-End Reset
```

```
OW_BitMode      CON     %0100
OW_HighSpd      CON     %1000

ReadROM         CON     $33             ' read ID, serial num, CRC
MatchROM        CON     $55             ' look for specific device
SkipROM         CON     $CC             ' skip ROM (one device)
SearchROM       CON     $F0             ' search

Yes             CON     1
No              CON     0


' ----[ Variables ]-----------------------------------------------------
'
thisSlot        VAR     Nib             ' this program slot
caller          VAR     Nib             ' calling program
ow_pin          VAR     Nib             ' BS2p pin for OW bus
dStart          VAR     Byte            ' start of EE data

lastDisc        VAR     Byte            ' previous collision location
doneFlag        VAR     Bit             ' done with this search?
moreDevs        VAR     Bit             ' more devices on bus?
result          VAR     Nib             ' result of bit read
bitIndex        VAR     Byte            ' current bit position
tempROM         VAR     Byte(8)         ' holds ROM search
discMarker      VAR     Byte            ' collision in this search

devCount        VAR     Byte            ' devices found
addr            VAR     Byte            ' EE address to store
offset          VAR     Byte            ' loop counter

crcVal          VAR     Byte            ' CRC test value
crcErrors       VAR     Byte            ' Errors on search


' ----[ EEPROM Data ]----------------------------------------------------
'
CRC_table
DATA    0,94,188,226,97,63,221,131,194,156,126,32,163,253,31,65
DATA    157,195,33,127,252,162,64,30,95,1,227,189,62,96,130,220
DATA    35,125,159,193,66,28,254,160,225,191,93,3,128,222,60,98
DATA    190,224,2,92,223,129,99,61,124,34,192,158,29,67,161,255
DATA    70,24,250,164,39,121,155,197,132,218,56,102,229,187,89,7
DATA    219,133,103,57,186,228,6,88,25,71,165,251,120,38,196,154
DATA    101,59,217,135,4,90,184,230,167,249,27,69,198,152,122,36
DATA    248,166,68,26,153,199,37,123,58,100,134,216,91,5,231,185
DATA    140,210,48,110,237,179,81,15,78,16,242,172,47,113,147,205
DATA    17,79,173,243,112,46,204,146,211,141,111,49,178,236,14,80
DATA    175,241,19,77,206,144,114,44,109,51,209,143,12,82,176,238
DATA    50,108,142,208,83,13,239,177,240,174,76,18,145,207,45,115
DATA    202,148,118,40,171,245,23,73,8,86,180,234,105,55,213,139
```

```
DATA    87,9,235,181,54,104,138,212,149,203,41,119,244,170,72,22
DATA    233,183,85,11,136,214,52,106,43,117,151,201,74,20,246,168
DATA    116,42,200,150,21,75,169,247,182,232,10,84,215,137,107,53


' ----[ Initialization ]-----------------------------------------------
'
Initialize:
  GET 127,thisSlot                  ' location of this code
  GET 126,caller                    ' calling program
  GET 125,ow_pin                    ' OW bus pin
  GET 124,dStart                    ' start of data storage in caller

  crcErrors = 0


' ----[ Main Code ]-----------------------------------------------------
'
FirstSearch:
  lastDisc = 0                      ' no previous search, no problems
  doneFlag = No                     ' this search is not done

NextSearch:
  moreDevs = Yes                    ' assume there are move devices
  IF (doneFlag = No) THEN ResetBus  ' intialize for search
  doneFlag = No
  GOTO SearchDone

ResetBus:
  bitIndex = 1                      ' initialize starting bit
  discMarker = 0                    ' no bit collisions yet
  OWOUT ow_pin,OW_FERst,[SearchROM]  ' reset and start search

ReadBit:
  OWIN ow_pin,OW_BitMode,[result.Bit1,result.Bit0]

  IF (result <> %11) THEN CheckGoodBit ' if %11, no devices
  GOTO ExitSearch

CheckGoodBit:
  IF (result > %00) THEN RecordBit  ' if %00, we had a collision

Conflict:
  IF (bitIndex <> lastDisc) THEN Con_GT ' same as last, set bit to 1
  tempROM.LowBit(bitIndex-1) = 1
  GOTO SendBit

Con_GT:
  IF (bitIndex < lastDisc) THEN Con_LT ' new location, set bit to 0
  tempROM.LowBit(bitIndex-1) = 0
  discMarker = bitIndex             ' mark the location
```

```
  GOTO SendBit

Con_LT:
  IF (tempROM.LowBit(bitIndex-1) = 1) THEN SkipMark
  discMarker = bitIndex              ' mark if bit was 0
SkipMark:
  GOTO SendBit

RecordBit:
  '
  ' write bus bit to our array
  '
  tempROM.LowBit(bitIndex-1) = result.Bit1

SendBit:
  '
  ' send bit back to bus to keep desired devices online
  '
  OWOUT ow_pin,OW_BitMode,[tempROM.LowBit(bitIndex-1)]

  bitIndex = bitIndex + 1            ' point to next bit
  IF (bitIndex < 65) THEN ReadBit    ' done with this device?
  lastDisc = discMarker              ' save collsion marker
  moreDevs = No                      ' assume we're done
  IF (lastDisc > 0) THEN SkipDone
  GOTO SearchDone                    ' record current device

SkipDone:
  moreDevs = Yes                     ' there are more devices

SearchDone:
  GOSUB CRCcheck
  IF (crcVal <> 0) THEN DoNext       ' don't save if CRC bad
  devCount = devCount + 1            ' increment device count
  addr = 8 * (devCount - 1) + dStart ' calculate starting address
  STORE caller                       ' point to caller EE
  FOR offset = 0 TO 7
    WRITE (addr + offset),tempROM(offset)
' write serial number to caller
  NEXT

DoNext:
  IF (moreDevs = Yes) THEN NextSearch ' if more, go get them

ExitSearch:
  PUT 123,crcErrors                  ' flag any search errors
  STORE caller                       ' point to caller EE
              '
  WRITE (dStart - 1),devCount        ' record # devices found
  RUN caller                         ' return to calling program
  END
```

```
' ----[ Subroutines ]-------------------------------------------------
'
' CRC routine to check returned data
' - value of 0 indicates good CRC check

CRCcheck:
  STORE thisSlot                    ' CRC table is stored locally
  crcVal = 0                        ' initialize CRC
  FOR offset = 0 TO 7
    READ (CRC_table + (crcVal ^ tempROM(offset))),crcVal
  NEXT
  IF (crcVal = 0) THEN CRCdone
  crcErrors = crcErrors + 1

CRCdone:
  RETURN
```

```
' Program Listing 72.3
' Nuts & Volts - Stamp Applications, April 2001

' {$STAMP BS2p}

' ----[ Title ]---------------------------------------------------------
'
' File..... DSNAMES.BSP
' Purpose... Stores string names of DalSemi 1-Wire devices
' Author.... Jon Williams
' E-mail.... jonwms@aol.com
' Started... 01 MAR 2001
' Updated... 05 MAR 2001


' ----[ Program Description ]-------------------------------------------
'
' There is no actual code in this module. This program stores the names
' of various Dallas Semiconductor 1-Wire devices.  The device family code
' is used to map the strings in EEPROM.  The pointer to a device's string
' discription is stored at the location determined by this formula:
'
'    pointer = device_id * 2 + $600
'
' "pointer" is actually the low-byte of the address location.  The high
' byte is at pointer+1.


' ----[ Constants ]-----------------------------------------------------
'
PntrBase              CON     $600
```

```
' ----[ EEPROM Data ]-------------------------------------------------------
'
' Store strings first so labels can be used in address pointer table
'
Unknown        DATA   "Unknown device",0

' shared family codes

FCode01        DATA   "DS1990 iButton or DS2401",0
FCode04        DATA   "DS1994 iButton or DS2404",0
FCode10        DATA   "DS1920 iButton or DS18S20",0
FCode14        DATA   "DS1971 iButton or DS2430",0
FCode23        DATA   "DS1973 iButton or DS2433",0

' iButtons

DS1920         DATA   "DS1920 Temperature iButton",0
DS1921         DATA   "DS1921 Thermochron iButton",0
DS1963         DATA   "DS1963 4-kBit Monetary iButton",0
DS1971         DATA   "DS1971 256-bit EEPROM iButton",0
DS1973         DATA   "DS1973 4-kBit EEPROM iButton",0
DS1982         DATA   "DS1982 1-kBit Add-Only iButton",0
DS1985         DATA   "DS1985 16-kBit Add-Only iButton",0
DS1986         DATA   "DS1986 64-kBit Add-Only iButton",0
DS1990         DATA   "DS1990 Serial Number iButton",0
DS1991         DATA   "DS1991 MultiKey iButton",0
DS1992         DATA   "DS1992 1-kBit Memory iButton",0
DS1993         DATA   "DS1993 4-kBit Memory iButton",0
DS1994         DATA   "DS1994 4-kBit Plus Time Memory iButton",0
DS1995         DATA   "DS1995 16-kBit Memory iButton",0
DS1996         DATA   "DS1996 64-kBit Memory iButton",0

' 1-Wire chips

DS1822         DATA   "DS1822 Econo-MicroLAN Digital Thermometer",0
DS18B20        DATA   "DS18B20 Programmable Resolution Digital
Thermometer",0
DS18S20        DATA   "DS18S20 High Precision Digital Thermometer",0
DS2401         DATA   "DS2401 Silicon Serial Number",0
DS2404         DATA   "DS2404 EconoRAM Time Chip",0
DS2405         DATA   "DS2405 Addressable Switch",0
DS2406         DATA   "DS2406 Addressable Switch Plus 1-kBit Memory",0
DS2417         DATA   "DS2417 Time Chip With Interrupt",0
DS2430         DATA   "DS2430 256-bit EEPROM",0
DS2433         DATA   "DS2433 4-kBit EEPROM",0
DS2450         DATA   "DS2450 Quad A/D Converter",0
DS2505         DATA   "DS2505 16-kBit Add-Only Memory",0
DS2506         DATA   "DS2506 64-kBit Add-Only Memory",0
DS2890         DATA   "DS2890 Digital Pot",0
```

```
' string pointers

Pointers        DATA    @$01*2+PntrBase,Word FCode01
                DATA    @$04*2+PntrBase,Word FCode04
                DATA    @$10*2+PntrBase,Word FCode10
                DATA    @$14*2+PntrBase,Word FCode14
                DATA    @$23*2+PntrBase,Word FCode23

                DATA    @$02*2+PntrBase,Word DS1991
                DATA    @$06*2+PntrBase,Word DS1993
                DATA    @$08*2+PntrBase,Word DS1992
                DATA    @$09*2+PntrBase,Word DS1982
                DATA    @$0A*2+PntrBase,Word DS1995
                DATA    @$0C*2+PntrBase,Word DS1996
                DATA    @$1A*2+PntrBase,Word DS1963
                DATA    @$21*2+PntrBase,Word DS1921

                DATA    @$05*2+PntrBase,Word DS2405
                DATA    @$0B*2+PntrBase,Word DS2505
                DATA    @$0F*2+PntrBase,Word DS2506
                DATA    @$12*2+PntrBase,Word DS2406
                DATA    @$20*2+PntrBase,Word DS2450
                DATA    @$22*2+PntrBase,Word DS1822
                DATA    @$27*2+PntrBase,Word DS2417
                DATA    @$28*2+PntrBase,Word DS18B20
                DATA    @$2C*2+PntrBase,Word DS2890
```