



Column #88 August 2002 by Jon Williams:

Digital Data Recording

My first job after military service was working for a large turf irrigation company. With a background in electronics, my focus was, of course, directed at irrigation controllers [sprinkler timers]. I got pretty good at fixing them and was quickly promoted to the testing group where I got to work with new designs. Working with new ones was much more fun than fixing the broken ones.

Keep in mind that a sprinkler controller is a real-time device designed to sequentially activate selected stations [outputs] at some predetermined time. In our lab, the standard piece of equipment used to verify this behavior was a paper strip chart recorder (this was 18 years ago). We'd program the controller to run a test sequence and start the strip-chart recorder, noting the time that the test was started. In the morning we would verify the controller activity by reading the strip-chart markings.

Why am I dredging up what is – electronically – ancient history? Well, a recent posting on the BASIC Stamp mailing list caused me to remember my time in the test lab. A Stamp user was looking to build an event recorder using the BASIC Stamp. The post made me think about the good old days of paper strip-charts and the PC system we ultimately designed to replace them. I wondered what I could [simply] do with a stock BS2. As it turned out, the project is pretty neat.

Just Save The Changes

A big problem with the old strip chart recorders is that they used paper to record activity and would happily spit out loads of paper – even if nothing was happening. If we equate paper to memory, this is really just a waste. A more efficient plan is simply to note the time when something changes.

And that's what we'll do here. Our project this month, is a simple event recorder that will monitor up to eight inputs and record changes in the state of these inputs to an external EEPROM. We'll use a real-time-clock for accurate timing and our design will scan the inputs every second.

Inside the "Box"

I envisioned this project as something that would be used remotely – a smart "black box." This being the case, the interface to the user is provided via the PC through the Stamp programming port. The heart of the circuit (Figure 88.1) is simply the Stamp and a couple of 8-pin dip components (the EEPROM and RTC). The circuit and battery can be easily packaged in a very small plastic or metal box.

I will leave input conditioning up to you. For my test, I used a simple N.O. switch circuit as shown in Figure 88.2. If you want to monitor high-voltage devices or AC, the switch could be replaced by the contacts of a relay. Or, even better, you could use an optical isolator.

The hardware's pretty simple, isn't it? Well, the code really is too, albeit fairly long. As presented, the program uses up almost all of the Stamp's code space. Most of the program is fairly modular and we'll be taking advantage of some of the I2C code developed a couple of months ago – this time we'll put it to real use.

Before we get into the detailed explanation, let's review what the program should do:

- Read the number of records stored from the Stamp's EEPROM
- Read the start date (month, day, year) from the Stamp's EEPROM
- Draw the menu
- Wait for user input – update time while waiting
- Respond to the user input

Figure 88.1: EEPROM and RTC with BS2

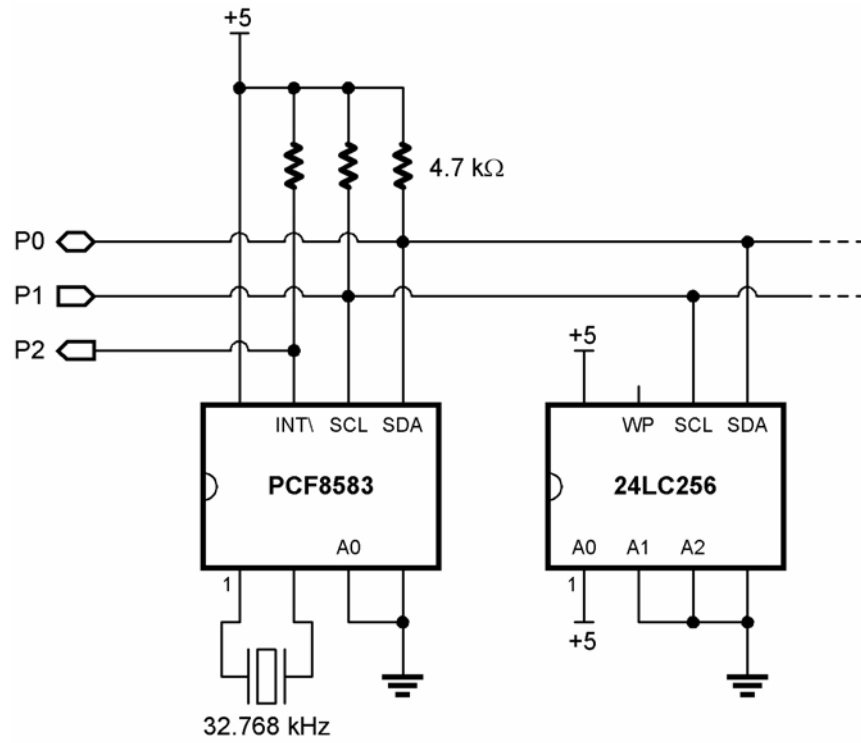
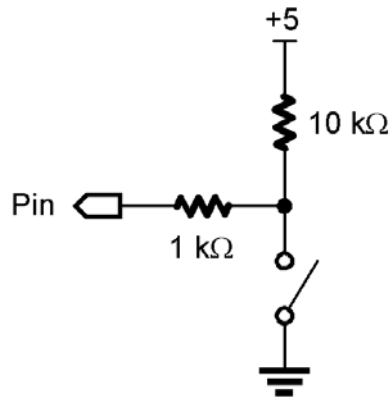


Figure 88.2: Pushbutton Circuit



Response from the user can be:

- "T" : Enter new time (hours, minutes, seconds)
- "D" : Enter new start date (month, day, year)
- "R" : Start recording
- "V" : View stored records
- "C" : Clear records

The last part of our planning is the storage of an event. As discussed earlier, we will only store changes that are detected. Here's how the bytes in an event record are structured:

- 0 : Day (offset from start date)
- 1 : Hour
- 2 : Minutes
- 3 : Seconds
- 4 : Inputs

Let me just explain byte 0 a bit further. Instead of storing the event date in a record (which would take four bytes), we simply store an offset (number of days) from the start date of recording. This trims three bytes from the record, allowing more records to be stored. It also means that we don't have to send the date to or retrieve it from the PCF8583; the start date is simply recorded in the Stamp's memory for future reference. Let's get on with the code.

In the variables section you'll notice a comment about not changing the order of a group of variables. In this program we will take advantage of the Stamp's implicit array structure of variable memory. A group of variables of the same size can be accessed as an array by using the first of the group as the array name and applying an index. Here's the group I'm referring to:

```
dayOfs  
hours  
minutes  
seconds  
scan
```

In the code, we can access to the same group using these names:

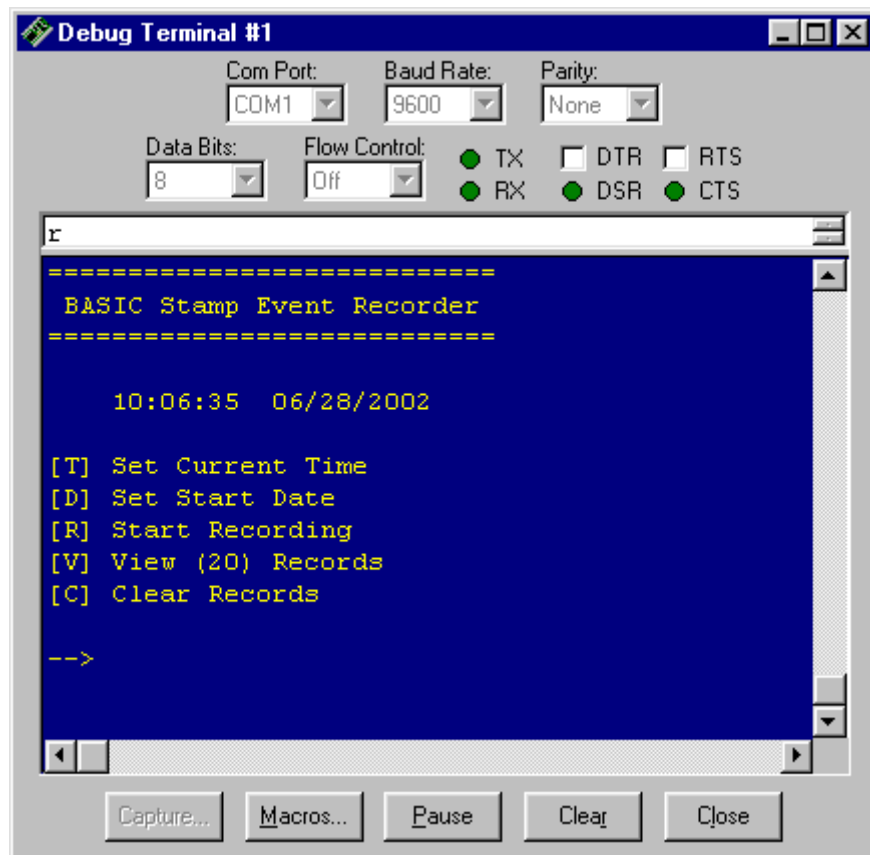
```
dayOfs(0)  
dayOfs(1) – same as hours  
dayOfs(2) – same as minutes  
dayOfs(3) – same as seconds  
dayOfs(4) – same as scan
```

As you've certainly deduced by now, we'll use a loop counter to iterate through these variables.

Let's move on to the EEPROM section; another place with something a little different. Storing data in the Stamp's EEPROM is easy and we do it quite frequently. I want to point out a little-used modifier of the DATA statement: Word. This let's use store a word-sized variable as easily as putting it into a variable. The compiler will store the variable as two bytes, using the "Little Endian" approach (low byte first). Keep in mind that this modifier only works with DATA and not with WRITE or READ. For WRITE and READ we still must deal with bytes.

The initialization section is straightforward, simply reading the number of records stored and the start date from the Stamp's EEPROM. A short PAUSE is inserted to allow the DEBUG window to open before we get draw the menu.

Figure 88.3: DEBUG Menu



Drawing the menu is easy and since we're using the DEBUG window, we'll take advantage of the cursor positioning command and, later, the ability to clear a line from the cursor position to the right. These commands only work with the DEBUG window, so if you decide to change the program to work with a standard terminal, you'll need to write your own positioning and "clean up" code.

After the menu is displayed, we grab the current time from the PCF8583 – so let's go there. Jump down the routine called `Get_Clock`. This routine uses the low-level I2C code to access the seconds, minutes and hours data from the PCF8583. When retrieving consecutive bytes from the

PCF8583 (and other memory-type devices), we must first set the starting address with a write command. Once the address is set, we can perform sequential reads from that address. The PCF8583 will automatically increment its address pointer so we can do subsequent reads without having to send the next address.

Since the PCF8583 uses BCD for the time registers, we will convert to decimal with a bit of code that takes advantage of the HighNib and LowNib variable modifiers. This really isn't necessary for the program, but as most of us are more comfortable dealing with decimal numbers, it makes sense to do it.

With the time in hand, we'll put it in the display along with the current start date and wait for the user to press a key. If no key is pressed within 900 milliseconds, the SERIN line times out and goes back to the Show_Time_Date code. What this does is create a "live" display, showing us the current time in the PCF8583.

When a key is pressed, it is decoded with a LOOKDOWN table. This will convert the response from a character to a value between zero and nine – if the key is valid. If the key isn't valid, the LOOKDOWN table will have no effect. Next we divide by two, giving us a possible [legal] value between zero and four that will be used by BRANCH for our menu routines. If the key was legal, BRANCH will work. If not, the BRANCH statement will fall through and the code will restart at Main.

The next section deals with code to handle each of the menu commands. The first two routines are identical, so we'll just discuss the first: Set_Time.

When we want to enter a new time, the screen is cleared and we're asked to enter the hours. To keep things simple (for the Stamp), we'll use the 24 hour format. The DEC2 modifier is used with SERIN to limit the number of characters accepted. If the value entered is out of range, we clear the entry and try again. The same technique is used to get the minutes and seconds.

Once a valid time has been entered, it is sent to the PCF8583 with the Put_Clock subroutine. This routine works very much like Get_Clock; just going the other direction (data to PCF8583).

The heart of the program is, of course, recording data. When this option is selected, we'll clear the days offset counter then collect the current inputs and invert them. The reason for this is that we want to force the recording code to create an entry at the beginning. This way we have stored the starting time and initial state of the inputs.

Let me get away from code for a bit and tell you about the Interrupt\ output of the PCF8583. This output is used to indicate alarms from the device (by being pulled low). By default, it outputs a 1

Hz square wave. This is perfect for us to trigger our new scan cycle. Each time this output goes low, we know it's a new second. Monitoring the Interrupt\ line is more efficient than continuously reading the time and looking for a change.

The code will loop at `Wait_For_Int` until Stamp pin 2 is pulled low. When this happens, we grab the current time from the PCF8583. At first, this may not seem necessary if there was no new event. We have to do it though since we're keeping track of the days ourselves. So, if the current hours is zero (midnight) and the current seconds value is also zero, we've just hit a new day and we increment the `daysOfs` variable. If not, we simply skip ahead and look at the inputs.

Since inputs can be "noisy" the code at `Check_Inputs` will debounce them. We've used this code before; it simply loops a few times and makes sure that an input doesn't change (bounce) during the loop. Any non-changing input is passed through the loop as a good input.

If there has been a change in the inputs, we'll save the change and record the event to our 24LC256 with the `Put_Record` subroutine. Let's go there.

The `Put_Record` subroutine updates the record count and checks to make sure we still have room in the 24LC256 for data. If not, the program stops, otherwise we'll save the current scan time and inputs. The current record number is stored in the Stamp's EEPROM so we can retrieve it after a power loss or reset. The next step is to setup for the `Put_Byte` subroutine by selecting our device type and the address size.

The `Put_Byte` routine is a general-purpose update from older code that lets us use it for either the 24LC256 or the PCF8583 (if we want to set something other than the time). The update includes passing the device as a variable and a flag for the number of bytes in the internal address. The device code is actually supposed to be the write code for the device, but the program masks out the lower bit (read bit), just in case of an error. After getting an ACK from the device (it's ready), we'll check the address size. For the 24LC256 we'll send two bytes (high byte first); for the PCF8583 we would only send one byte. After that, it's a simple matter of sending the data byte and generating a stop. This code, as well as `Get_Byte`, are good copy-and-paste code chunks for programs that use I2C devices.

Back to `Put_Record`. A loop is used to iterate through the bytes in our event record. Now we can fully understand the earlier discussion of keeping the event variables in specific order in our definition section. It's not enough that they're defined, they have to be defined consecutively so that this loop code will send the correct information to the 24LC256. The 24LC256 address for a given byte in the record is calculated and finally stored with `Put_Byte`.

With the record stored, we'll display the change on screen and then check to the state of "interrupt" pin. As I told you earlier, what we'll see on the input of pin 2 is a 1 Hz square wave generated by the PCF8583. If the storage and display of our record takes less than 500 milliseconds (I measured it at about 290 milliseconds with a BS2), then pin 2 will still be low when we're done. What we don't want to do, then, is go back to Wait_For_Interrupt – because we'll just do things again unnecessarily. So, we'll just wait for this pin to go back high, then we'll jump to Wait_For_Interrupt.

To stop recording we'll reset the Stamp or cycle the power. Back to the menu, we should see the correct number of stored records indicated. Pressing "V" will display them on screen. The code at View_Recs is responsible for the display. It uses a loop and Get_Record to retrieve the data from the 24LC256.

Finally, we will want to clear our records for a new cycle. This is a simple matter of clearing the variable and writing zeros to the EEPROM locations that hold our record count. There's no need to actually erase the 24LC256 – this would simply be a waste of its available write cycles.

Improvements

What you'll notice is that there isn't a whole lot of code space left for improvements – mostly because of all the DEBUG statements used to create our interactive display. We're going to solve that next time by using an external control program with Visual Basic. Until then, Happy Stamping!

Column #88: Digital Data Recording

```
' =====
'
' File..... EventRecord.BS2
' Purpose... Simple Event Recorder
' Author.... Jon Williams
' E-mail.... jwilliams@parallaxinc.com
' Started... 26 JUN 2002
' Updated... 28 JUN 2002
'
'   {$STAMP BS2}
'
' =====
'
' -----
' Program Description
' -----
'
' This program scans the upper eight inputs for changes and, when detected,
' records the new inputs with the day [offset] and time to an EEPROM
'
' Event Record Structure:
' 0 : days offset from start date
' 1 : hours
' 2 : minutes
' 3 : seconds
' 4 : input scan
'
' -----
' Revision History
' -----
'
' -----
' I/O Definitions
' -----
'
SDA          CON      0          ' I2C serial data line
SCL          CON      1          ' I2C serial clock line
IntPin       VAR      In2        ' interrupt input pin from RTC
NewInputs    VAR      InH        ' inputs on pins 8 - 15
TermIO       CON      16         ' Terminal IO
'
' -----
' Constants
' -----
PCF8583      CON      %10100000  ' device code for RTC
```

```

EE24LC256      CON      %10100010      ' device code for EEPROM
ACK            CON      0              ' acknowledge bit
NAK           CON      1              ' no ack bit

RecSize       CON      5              ' five bytes per event record
MemSize       CON      32768          ' assuming 1 24LC256
MaxRecs       CON      MemSize / RecSize

ByteSize      CON      0              ' byte-sized address (RTC)
WordSize      CON      1              ' word-sized address (EEPROM)

Yes           CON      0
No            CON      1

TermBaud      CON      84             ' 9600-8-N-1 (matches DEBUG)
CrsrXY        CON      2             ' DEBUG Position Control
ClrRt         CON      11            ' clear line to right

' -----
' Variables
' -----

device        VAR      Byte           ' device to write/read
devAddr       VAR      Word           ' address in device
addrSize      VAR      Bit            ' (bytes in address) - 1
i2cReg        VAR      Byte           ' register address
i2cData       VAR      Byte           ' data to/from device
i2cWork       VAR      Byte           ' work byte for TX routine
i2cAck        VAR      Bit            ' ACK bit from device

records       VAR      Word           ' events stored
recNum        VAR      Word           ' counter for view display
oldInputs     VAR      Byte           ' last event input data

' do not change order of next five variables
' -- program uses implicit array structure of user memory

daysOfs      VAR      Byte           ' offset from start date
hours         VAR      Byte           ' time of event
mins          VAR      Byte
secs          VAR      Byte
scan          VAR      Byte           ' event data

month         VAR      recNum.LowByte  ' start date
day           VAR      recNum.HighByte
year          VAR      Word

response      VAR      Word           ' user response
idx           VAR      Nib
    
```

Column #88: Digital Data Recording

```
' -----  
' EEPROM Data  
' -----  
  
NumRecs      DATA   Word 0      ' stored records  
StartMonth   DATA   6          ' start date of recording  
StartDay     DATA   28  
StartYear    DATA   Word 2002  
  
' -----  
' Initialization  
' -----  
  
Init:  
  READ NumRecs, records.LowByte      ' retrieve record count  
  READ (NumREcs + 1), records.HighByte  
  
  READ StartMonth, month             ' retrieve start date  
  READ StartDay, day  
  READ StartYear, year.LowByte  
  READ (StartYear + 1), year.HighByte  
  
  PAUSE 250                          ' let DEBUG window open  
  
' -----  
' Program Code  
' -----  
  
Main:  
  DEBUG CLS                          ' display menu  
  DEBUG "=====", CR  
  DEBUG " BASIC Stamp Event Recorder ", CR  
  DEBUG "=====", CR  
  
  DEBUG CrsrXY, 0, 6  
  
  DEBUG "[T] Set Current Time", CR  
  DEBUG "[D] Set Start Date", CR  
  DEBUG "[R] Start Recording", CR  
  DEBUG "[V] View (", DEC records, ") Records", CR  
  DEBUG "[C] Clear Records", CR  
  
  DEBUG CrsrXY, 0, 12, "--> ", ClrRt  
  
Show_Time_Date:                      ' show current time & date  
  GOSUB Get_Clock  
  DEBUG CrsrXY, 4, 4
```

```

GOSUB Display_Time
DEBUG " "
GOSUB Display Date

Get User Input:                                ' wait for response
DEBUG CrsrXY, 4, 12
SERIN TermIO, TermBaud, 900, Show_Time_Date, [response]
LOOKDOWN response, ["tTDrRvVcC"], response
response = response / 2
BRANCH response, [Set Time, Set Date, Go Record, View Recs, Clear Recs]
GOTO Main

' -----
' Menu Routines
' -----

' --- Time ---

Set Time:
DEBUG CLS, "Set Current Time"

Enter Hours:
DEBUG CrsrXY, 0, 2, "Enter Hours (0..23)..... ", ClrRt
SERIN TermIO, TermBaud, [DEC2 hours]
IF (hours > 23) THEN Enter Hours

Enter Minutes:
DEBUG CrsrXY, 0, 3, "Enter Minutes (0..59)... ", ClrRt
SERIN TermIO, TermBaud, [DEC2 mins]
IF (mins > 59) THEN Enter_Minutes

Enter Seconds:
DEBUG CrsrXY, 0, 4, "Enter Seconds (0..59)... ", ClrRt
SERIN TermIO, TermBaud, [DEC2 secs]
IF (secs > 59) THEN Enter_Seconds

GOSUB Put Clock                                ' send new time to PCF8583
GOTO Main

' --- Date ---

Set Date:
DEBUG CLS, "Set Start Date"

Enter Month:
DEBUG CrsrXY, 0, 2, "Enter Month (1..12)... ", ClrRt
SERIN TermIO, TermBaud, [DEC2 month]
IF (month < 1) OR (month > 12) THEN Enter Month

```

Column #88: Digital Data Recording

```
Enter_Day:
  DEBUG CrsrXY, 0, 3, "Enter Day (1..31)..... ", ClrRt
  SERIN TermIO, TermBaud, [DEC2 day]
  IF (day < 1) OR (day > 31) THEN Enter Day

Enter_Year:
  DEBUG CrsrXY, 0, 4, "Enter Year (2002+).... ", ClrRt
  SERIN TermIO, TermBaud, [DEC4 year]
  IF (year < 2002) THEN Enter Year

  WRITE StartMonth, month           ' save start date in EEPROM
  WRITE StartDay, day
  WRITE StartYear, year.LowByte
  WRITE (StartYear + 1), year.HighByte
  GOTO Main

' --- Record ---

Go Record:
  DEBUG CLS, "Recording... ", CR, CR

  daysOfs = 0                       ' start today
  oldInputs = ~NewInputs            ' force record on start

Wait For Int:
  IF (IntPin = No) THEN Wait For Int ' wait for new second
  GOSUB Get Clock                   ' get current time
  IF (hours <> 0) OR (secs <> 0) THEN Check Inputs
  daysOfs = daysOfs + 1             ' increment day counter
  IF (daysOfs = 0) THEN Stop_Recording ' if > 255 stop

Check Inputs:
  scan = %11111111
  FOR idx = 1 TO 5                  ' debounce inputs
    scan = scan & NewInputs
  PAUSE 5
  NEXT

  IF (scan = oldInputs) THEN Wait For No Int ' if same, skip
  oldInputs = scan                  ' save current scan
  GOSUB Put_Record                   ' save record in 24LC256

  DEBUG DEC3 daysOfs, " "           ' display record
  GOSUB Display Time
  DEBUG " -> ", BIN8 scan, CR

Wait_For_No_Int:
  IF (IntPin = Yes) THEN Wait_For_No_Int ' wait for high on int pin
  GOTO Wait For Int
```

```

' --- View ---
View Recs:
  IF (records = 0) THEN Main           ' oops...
  DEBUG CLS                            ' print header
  DEBUG "Records", CR
  DEBUG "Start Date: "
  GOSUB Display Date
  DEBUG CR, CR

  DEBUG "Day Time      Inputs ", CR
  DEBUG "--- -----", CR

  FOR recNum = 0 TO (records - 1)      ' print records
    GOSUB Get Record
    DEBUG DEC3 daysOfs, " "
    GOSUB Display_Time
    DEBUG " ", BIN8 scan, CR
  NEXT

  DEBUG CR, "Press a key..."         ' user escape
  SERIN TermIO, TermBaud, [response]
  GOTO Main

' --- Clear ---
Clear Recs:
  records = 0                          ' clear variable
  WRITE NumRecs, 0                     ' clear EEPROM
  WRITE (NumRecs + 1), 0
  GOTO Main

' -----
' Subroutines
' -----

Display Time:
  DEBUG DEC2 hours, ":", DEC2 mins, ":", DEC2 secs
  RETURN

Display Date:
  DEBUG DEC2 month, "/", DEC2 day, "/", DEC4 year
  RETURN

Put Record:
  records = records + 1                 ' update record count

```

Column #88: Digital Data Recording

```
IF (records > MaxRecs) THEN Stop_Recording ' check limit
WRITE NumRecs, records.LowByte ' store count in local EEPROM
WRITE (NumRecs + 1), records.HighByte

device = EE24LC256 ' setup for Write Byte
addrSize = WordSize

FOR idx = 0 TO (RecSize - 1)
  i2cData = daysOfs(idx) ' byte to store
  devAddr = ((records - 1) * RecSize) + idx ' caculate address in EE
  GOSUB Write_Byte ' write it
NEXT
RETURN

Get_Record:
device = EE24LC256 ' setup for Read Byte
addrSize = WordSize

FOR idx = 0 TO (RecSize - 1)
  devAddr = (recNum * RecSize) + idx ' calculate address in EE
  GOSUB Read_Byte ' go get it
  daysOfs(idx) = i2cData ' place in user variable
NEXT
RETURN

Stop_Recording:
END

' -----
' High Level I2C Subroutines
' -----

' Byte to be written is passed in i2cData
' -- address passed in devAddr

Write_Byte:
GOSUB I2C_Start ' send Start
i2cWork = (device & %11111110) ' send write command
GOSUB I2C_TX_Byte
IF (i2cAck = NAK) THEN Write_Byte ' wait until [ee] not busy
IF (addrSize = ByteSize) THEN Wr_Low_Addr
i2cWork = devAddr / 256 ' send address high byte
GOSUB I2C_TX_Byte

Wr_Low_Addr:
i2cWork = devAddr // 256 ' send address low byte
GOSUB I2C_TX_Byte
i2cWork = i2cData ' send data
```

```

GOSUB I2C_TX_Byte
GOSUB I2C_Stop
RETURN

' Byte read is returned in i2cData
' -- address passed in devAddr

Read_Byte:
GOSUB I2C_Start                                     ' send Start
i2cWork = (device & %11111110)                     ' send write command
GOSUB I2C_TX_Byte
IF (i2cAck = NAK) THEN Read_Byte                   ' wait until [ee] not busy
IF (addrSize = ByteSize) THEN Rd_Low_Addr
i2cWork = devAddr / 256                             ' send address high byte
GOSUB I2C_TX_Byte

Rd_Low_Addr:
i2cWork = devAddr // 256                           ' send address low byte
GOSUB I2C_TX_Byte
GOSUB I2C_Start
i2cWork = (device | 1)                             ' send read command
GOSUB I2C_TX_Byte
GOSUB I2C_RX_Byte_Nak
GOSUB I2C_Stop
i2cData = i2cWork
RETURN

' Write seconds, minutes and hours .. sequential mode
' -- variables are converted to BCD before sending to PCF8583

Put_Clock:
GOSUB I2C_Start
i2cWork = PCF8583                                   ' send device address
GOSUB I2C_TX_Byte
i2cWork = 2                                         ' start with seconds reg
GOSUB I2C_TX_Byte
i2cWork = ((secs / 10) << 4) | (secs // 10)
GOSUB I2C_TX_Byte
i2cWork = ((mins / 10) << 4) | (mins // 10)
GOSUB I2C_TX_Byte
i2cWork = ((hours / 10) << 4) | (hours // 10)
GOSUB I2C_TX_Byte
GOSUB I2C_Stop
RETURN

' Read seconds, minutes and hours .. sequential mode
' -- variables are converted from BCD storage format

```

Column #88: Digital Data Recording

```
Get_Clock:
  GOSUB I2C_Start
  i2cWork = PCF8583
  GOSUB I2C_TX_Byte
  i2cWork = 2
  GOSUB I2C_TX_Byte
  GOSUB I2C_Start
  i2cWork = (PCF8583 | 1)
  GOSUB I2C_TX_Byte
  GOSUB I2C_RX_Byte
  secs = i2cWork.HighNib * 10 + i2cWork.LowNib
  GOSUB I2C_RX_Byte
  mins = i2cWork.HighNib * 10 + i2cWork.LowNib
  GOSUB I2C_RX_Byte Nak
  hours = i2cWork.HighNib * 10 + i2cWork.LowNib
  GOSUB I2C_Stop
  RETURN

' -----
' Low Level I2C Subroutines
' -----

' --- Start ---

I2C_Start:
  INPUT SDA
  INPUT SCL
  LOW SDA
  I2C_Start:
  ' I2C start bit sequence
  ' SDA -> low while SCL high

Clock_Hold:
  IF (Ins.LowBit(SCL) = 0) THEN Clock Hold
  RETURN
  ' device ready?

' --- Transmit ---

I2C_TX_Byte:
  SHIFTOUT SDA,SCL,MSBFIRST,[i2cWork\8]
  SHIF TIN SDA,SCL,MSBPRE,[i2cAck\1]
  RETURN
  ' send byte to device
  ' get acknowledge bit

' --- Receive ---

I2C_RX_Byte Nak:
  i2cAck = NAK
  GOTO I2C_RX
  ' no ACK = high

I2C_RX_Byte:
  i2cAck = ACK
  ' ACK = low
```

```
I2C RX:
  SHIFTIN SDA,SCL,MSBPRE,[i2cWork\8]      ' get byte from device
  SHIFTOUT SDA,SCL,LSBFIRST,[i2cAck\1]    ' send ack or nak
  RETURN

' --- Stop ---

I2C Stop:
  LOW SDA                                  ' I2C stop bit sequence
  INPUT SCL
  INPUT SDA                                ' SDA --> high while SCL high
  RETURN
```