

Manual Scan4a.Bs2

The purpose of this program is to setup the moon tracker for studying the sky. Besides being able to trim the head into alignment, it provides for manual over-rides in servo control, preprogrammed 'park' positions, ability to reset and record center values in the field, visual feed-back for public demonstrations that permit real time sampling, and data collection via Excel¹ spreadsheet – that can be email attached in any shared data experiment.

PROGRAM DESCRIPTION:

The first two lines of code are 'meta' commands that tell the compiler what microchip to compile to and what language version of basic it can handle. I constructed the device using the Stamp In Class 'Basic Stamp 2' microcontroller from www.parallax.com

```
' {$STAMP BS2}
' {$PBASIC 2.5}
```

The next line of code tells the compiler to 'reserve' 8 bytes in the EEPROM beginning at address 0. This is where the import 'trim' settings, data counter, and first 'good' moon readings are stored.

```
DATA @0, (8)
```

It's then followed by commands to set the direction of the I/O pins on the lower 8 bits of the controller to input and then clears them.

```
DIRL = $00
OUTL = $00
```

The next lines of code define I/O pin constants to make the code easier to read.

```
Cap          PIN 13
Az_Pin       PIN 14
Elev_Pin     PIN 15
Ground       PIN 7
Buzzer       PIN 12
```

The next series of lines are general constants (not pins), they are varied and have different uses. The comments immediately after them provide info for the use.

```
Center       CON 750      ' position code for servo center.
Charge       CON 10       ' msec of charge.

Left         CON 3        ' Mask for reading left sensors only
```

¹ Using the old PLX-DAQ macro from Slema ... slightly modified.

```

Top          CON 6          ' Mask for reading top sensors only
OneEye       CON 15         ' Mask for reading all sensors as one eye.

MinServo     CON 250        ' min servo
MaxServo     CON 1250       ' max servo
HalfB        CON 128        ' half a byte
BuzFrq       CON 4000       ' buzzer frequency in Hertz.
ParkHead     CON 400        '

```

These are followed by the variables that will be used by the software. Valid variable types are Word (2 bytes) or Byte. Words can hold values ranging from zero to 65535. Whereas, bytes hold values between zero and 255.

```

dXY          VAR Byte       ' trim for azimuth
dZ           VAR Byte       ' trim for zenith

Auto         VAR Byte
Cnt          VAR Byte
X            VAR Byte

A            VAR Word       ' We need 4 variables to hold
B            VAR Word       ' the values read from the
C            VAR Word       ' CDS cells.
D            VAR Word
All          VAR Word

Status       VAR Word
Temp         VAR Word
XY           VAR Word
Z            VAR Word

```

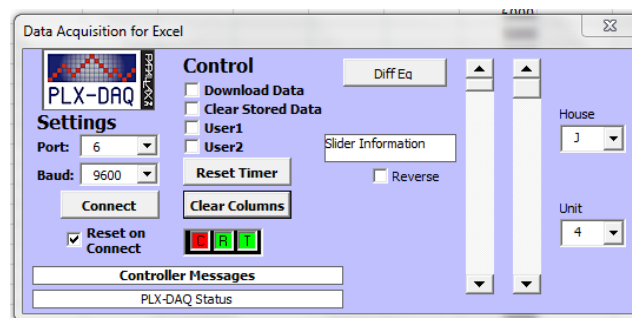
Then, the tri-color led pins are both set to output zero. More on this later.

```

LOW 8                ' Tri-color LED pins
LOW 9

```

Next, the dialog in Excel is initialized with its startup values. We use DEBUG in this controller to generate the equivalent of PRINT in other versions of basic. As well as, DEBUGIN for the equivalent of INPUT in other versions of basic. And the dialog can exchange information using this technique. Here is the dialog before the commands below are executed:



```

DEBUG "SLIDER1,MAX," , DEC MaxServo,CR           ' Maximum slider value
DEBUG "SLIDER1,MIN," , DEC MinServo,CR           ' Minimum slider value

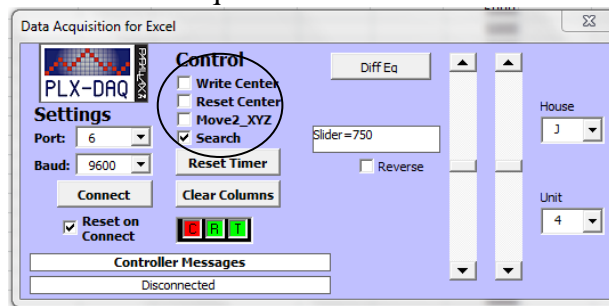
DEBUG "SLIDER2,MAX," , DEC MaxServo,CR           ' Maximum slider value
DEBUG "SLIDER2,MIN," , DEC MinServo,CR           ' Minimum slider value

DEBUG "DOWNLOAD,LABEL,Write Center",CR
DEBUG "STORED,LABEL,Reset Center",CR
DEBUG "USER1,LABEL,Move2_XYZ",CR
DEBUG "USER2,LABEL,Search",CR

DEBUG "STATUS",CR
DEBUGIN DEC Status

```

And, here is the dialog after the lines above are executed. The circled items are returned in a single value, when STATUS is requested:



Then the following lines determine from status if any *startup* conditions need processing. Let me repeat that in different words, the switches have one function during ‘run-time’ ... and another function during ‘startup’ of the program.

This piece of code reports to Excel the position of the last valid read and the trim values stored internally for the system. It does so directly into the spread sheet.

```

IF Status & 1 THEN
  GOSUB Last_Found
  DEBUG "CLEARDATA",CR
  DEBUG "LABEL,Date,Time,XY,Z,dXY,dZ",CR
  DEBUG "DATA,DATE,TIME," , DEC XY, " ,", DEC Z, " ,", DEC dxy, " ,", DEC
dz,CR
  END
ENDIF

```

Note – the ‘&’ operator is a binary ‘AND’ operation of one value with another. So the first line reads in plain English as, “if the 1st bit is set in *status* then” ... do these lines.

This piece of code checks to see if the operator wants to ‘park’ the heads for storage.

```

IF Status & 2 THEN
  XY = center
  Z = ParkHead
  GOSUB Move2_XYZ

```

```

GOSUB Move2_XYZ
GOSUB Move2_XYZ
END
ENDIF

```

This last test determines if the operator wishes to clear the last moon reading and return the heads to the upright ‘zenith’ position for bubble level check. (4th bit) If it is not checked, it will return the head to the last position it had a good moon reading.

```

IF Status & 8 THEN
    GOSUB Reset_Head
    DEBUG "USER2, SET, 0", CR
ELSE
    GOSUB Last_Found
ENDIF

GOSUB Move2_XYZ
GOSUB Move2_XYZ
GOSUB Move2_XYZ

```

MAIN

This is the main routine used to control all other routines. So, let’s jump right in and look at the code:

We start by creating an infinite loop structure:

```
DO
```

Then we tell the dialog in Excel to clear the spreadsheet of any old data and then label the top row of cells in the spread sheet.

```

DEBUG "CLEARDATA", CR
DEBUG "LABEL, Date, Time, Ref, L/R, T/B, Auto", CR

```

Then a counter is cleared and a ‘nested’ do-loop is created that only loops 128 times.

```

Cnt = 0
DO WHILE Cnt < 128

```

We fetch the 4 switches from the dialog in Excel.

```

DEBUG "STATUS", CR                                ' ask for info
DEBUGIN DEC Status                                ' then get it

```

Next, we check to see if the operator ‘aborted’ a search in progress?

```
IF Status = 0 AND Auto <> 3 THEN Auto = 0
```

Then we check to see if the ‘reset’ switch is set? If so, we clear the ‘trim’ settings from memory.

```

IF Status & 2 > 0 THEN                                ' reset center
  DEBUG "SLIDER1,SET,", DEC Center,CR
  DEBUG "SLIDER2,SET,", DEC Center,CR
  dXY = HalfB
  dZ  = HalfB
  GOSUB Record_Data
  Status = Status | 5
  DEBUG "STORED,SET,0",CR
ENDIF

```

Next, we see if any scroll bars have changed? And, if they have, read the new XY & Z into memory.

```

IF Status & 4 > 0 THEN                                ' have scroll bars changed?
  DEBUG "SLIDER1,GET",CR                               ' Request 1st scroll bar
  DEBUGIN DEC XY                                       ' put it in XY
  DEBUG "SLIDER2,GET",CR                               ' Request value for new Z
  DEBUGIN DEC Z                                         ' put reply in Z
  GOSUB Move2_XYZ                                       ' go there!
ENDIF

```

Then, we read the heads and provide real time data on screen. To begin with, we tell Excel we are sending ‘DATA’ along with the reserved words ‘DATE’ and ‘TIME’. These two can only appear immediately after the DATA directive. They are replaced in the spread sheet with the computers date and time stamp. Next, I transmit the value 32 as a reference line in Excel.

```

DEBUG "DATA,DATE,TIME,", DEC 32,","

```

Then, I load ‘Temp’ with the constant ‘Left’ and call CdS_AB. It reports back with the light levels in A, B, and the arc-tangent in ‘Temp’. I only care about the ‘arc-tangent’, so that is all that goes on screen.

```

Temp = Left
GOSUB CDS_AB
DEBUG DEC Temp,","

```

Now I read the top versus bottom light levels. And report this along with the value of the 'Auto' variable. Just for debugging purposes.

```
Temp = Top
GOSUB CDS_AB
DEBUG DEC Temp, ",", DEC Auto, CR
```

Then I add one to the variable 'Cnt' to keep track of how many data lines I have transmitted so far.

```
Cnt = Cnt + 1
```

Next, I check to see if the operator has selected 'search' from the dialog. And, if so, call the *Zero_In* routine to adjust the XY & Z values. I hand those values off to the Excel spreadsheet by setting the proper scroll bar. That causes the 3rd switch on the dialog to automatically set. So on the next pass through the loop, the new positions will be absorbed and move the head. If the variable 'Auto' reaches 3, then lock has been achieved and this routine then records that data, clears the search switch, and beeps.

```
IF Status & 8 > 0 THEN
  IF Auto > 2 THEN Auto = 0
  GOSUB Zero_In
  DEBUG "SLIDER1,SET,", DEC XY,CR
  DEBUG "SLIDER2,SET,", DEC Z,CR
  IF Auto = 3 THEN
    GOSUB Record_Data
    DEBUG "USER2,SET,0",CR
    GOSUB Beep_OK
  ENDIF
ENDIF
```

This last check determines if the operator is writing a new 'trim' position.

```
IF Status & 1 > 0 THEN
  WRITE 0,0,0
  dXY = XY - (Center - HalfB)
  dz = Z - (Center - HalfB)
  WRITE 2,dXY,dz
  GOSUB Beep_OK
  DEBUG "DOWNLOAD,SET,0",CR
ENDIF
```

```
' record center.
' write CNT=0
' compute XY adjustment
' compute Z adjustment.
' record center trim.
' announce "Done"
' clear DOWNLOAD flag.
```

Subroutines

These subroutines appear in all of the manual scan versions as well as the lunar scan versions written. As a matter of fact, I just *copy & paste* the subroutines into the programs I write and go from there.

ZERO IN:

This routine is called repeatedly until lock is reached. What it does when called depends on the value in AUTO ... initially AUTO is set to zero. You cannot lock onto the moon with just a single call to this routine. This allows the calling routine to determine if too many executions have occurred without conclusive results ... thereby generating an error. When the value in AUTO reaches 3, this routine has reached 'lock' and no further calls are required to find the moon.

```
SELECT Auto

CASE = 0
  Z = 1050
  Auto = 1

CASE = 1
  LOW 8          ' when both set low, LED flashes green.
  LOW 9
  Temp = Left    ' examine Left vs Right light levels.
  GOSUB CDS_AB   ' read the CdS cells
  IF Temp = 32 THEN
    Auto = 2     ' move on to next test
  ELSE
    XY = XY + Temp - 32 ' keep looking
  ENDIF

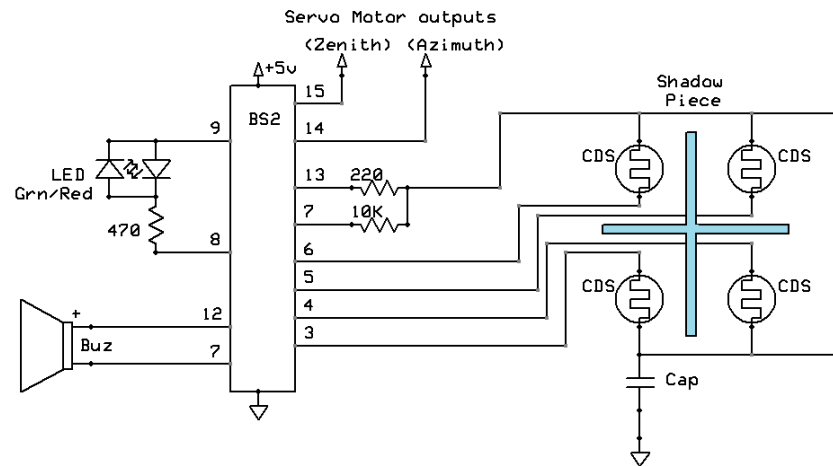
CASE = 2
  HIGH 8          ' when both high, LED flashes red.
  HIGH 9
  Temp = Top      ' tell CDS_AB to examine Top vs Bot
  GOSUB CDS_AB
  IF Temp = 32 THEN
    Auto = 3     ' lock achieved
  ELSE
    Z = Z - Temp + 32 ' keep looking
  ENDIF

ENDSELECT

RETURN
```

CDS AB:

This routine is the heart of the head reader ... and requires that I use a circuit to show how the information is collected from the head. Otherwise, the actions of the routine are cryptic ... and make no sense.



Moon Tracker Schematic

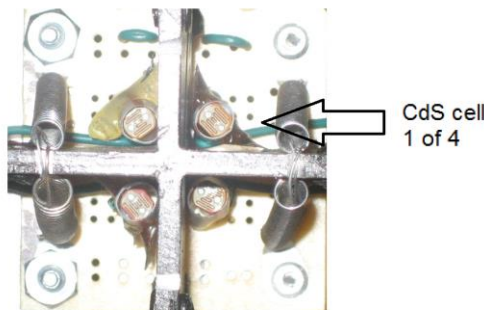


Photo of Head

To perform a measurement of light on the head, pin 13 is made high to charge the cap up to 5 volts. Next, one of the four pins (3, 4, 5, or 6) is set to an output of zero using the 'LOW' command (or its equivalent). The light striking that sensor(s) will 'bleed' the charge off slowly ... this is immediately followed by the command RCTIME to employ a fast internal *stop-watch* that times how long pin 13 takes to change from a logic 1, to a logic 0. And it records that time in a variable of your choosing. The value stored there is a count of elapsed time in 2 microsecond increments. Unless it took too long (more than 131,072 microseconds) and timed out, in which case the RCTIME command will set your variable to zero.

So, armed with this information, let's take a look at the code itself:

The first command is a fast way of saying “INPUT 1, INPUT 2, INPUT 3 ... INPUT 7” with a single command ... it makes all 8 lower pins *inputs* at the same time.

$$\text{DIRL} = 0$$

These next two commands start filling the cap and wait for a preset time to charge it:

HIGH Cap
PAUSE Charge

Next, which sensor we read is determined by a value the calling program first loaded in the variable named *Temp*. It is a 4 bit value (0..15) representing the discharge pins that should be used to drain the cap through the CdS cells. The operator '<<' shifts the bits left 3 bit positions to match pins 3 thru 6.

$$\text{DIRL} = \text{Temp} \ll 3$$

Since the lower 8 bit positions are always ready to output a zero, but cannot due the pin being declared as input ... when a bit position is made an output (instead of input) it will open a path for charge to leave the cap and go to ground. This makes it possible to open all four at once, and read the light by measuring the combined bleed ... which is what the constant *OneEye* is for. After setting the bleed pins, the next command is used to collect the stop-watch values in variable A.

RCTIME Cap,1,A

Then immediately after measuring the bleed time, stop the bleed!

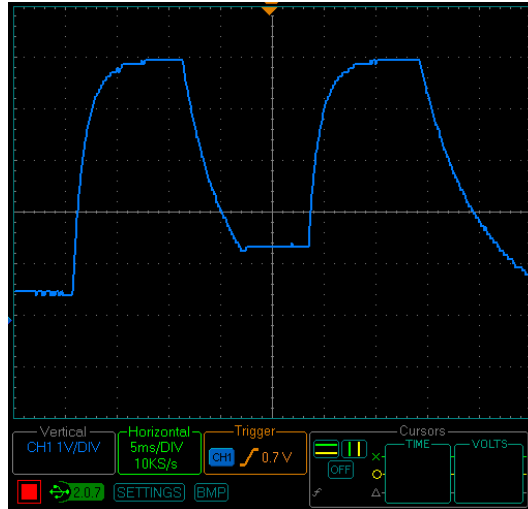
```
DIRL = 0          ' release pins
```

If the light levels are too low ... the BS2 can time out (indicated by storing zero in your variable), so I change this value to the MAX (unsigned integer) for mathematical reasons (avoids division by zero).

```
IF A = 0 THEN A = $FFFF      ' if 0 then indicate unreadable
```

The next line is for nothing more than to show the trigger voltage on an O-Scope. And if your sweep rate is set at 5 ms/div (like mine), this will produce a line with one division width on the screen of the O-Scope. (see capture)

PAUSE 5 ' makes trigger visible on O-scope



Scope Trace of voltage on cap as head is read by CDS_AB routine,
1v/div vertical and 5 ms/div horz.

Here you can see the flat line between the two readings that is generated by the pause command. Nice debug tool, as you can actually read the trigger voltage on screen. Additionally, we have a mind expanding observation! The cap attached to pin 13 is not digital ... as the scope clearly shows ... yet the BS2 only cares if the voltage on pin 13 is above 1.4 volts or not. If it is, it reads as a logic *one* ... otherwise, it's a logic *zero*.

The IF-THEN-ELSE structure that follows the pause command is used to determine if an additional read is necessary? If it is, the cap charges the same as before. However, the results are placed in variable B.

Then, the greater of A or B is used, to store the number of bits (less 7) containing the max value, in variable C².

```

IF Temp < OneEye THEN
  HIGH Cap
  PAUSE Charge
  DURL = (OneEye-Temp) << 3
  RCTIME Cap,1,B
  IF B = 0 THEN B = $FFFF

  IF A > B THEN
    C = (NCD A) - 7
  ELSE
    C = (NCD B) - 7
  ENDIF
  Temp = (A >> C) ATN (B >> C)
ELSE
  B = 0
  Temp = 0
ENDIF

```

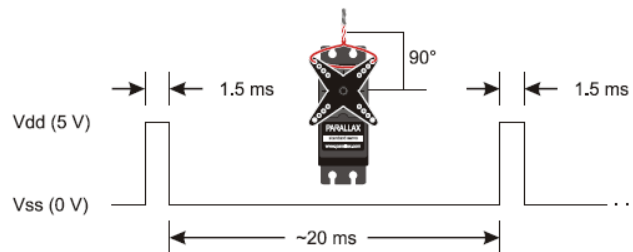
² The right shift that occurs (using variable C) ensures that the resulting angle returned by the *Arc-Tangent* function is always in quadrant one.

Then the *Arc-Tangent* of the B/A is evaluated and stored in *Temp*. Due to the shift using C, the returned angle will always be between 0 and 90 degrees. With 45 degrees indicating that the values A and B are really close to each other. And that will be the basis of the moon trackers logic³ deciding if it's looking at the light source. By trying head movements that bring the Temp value as close as possible to 45 degrees. Of course, the actual return value in *Temp* is in 'brads' ... not degrees. I will not go into the difference here, but only to refer interested readers to the built-in manual contained in the [Basic Stamp Editor](#). From inside the editor, click on the help 'icon' and look up the ATN binary operator.

MOVE2 XYZ:

This routine moves the servos to the position stored in XY and Z. But, here again I need to take a moment and point out how the information is passed to the servo from the microcontroller, and how that information effects the articulation of the heads. Otherwise, this code also would appear cryptic ... and make little sense.

I use 2 [Parallax Standard Servos](#) to move the base in the XY plane (pin 14), and the head in the Z axis perpendicular to the XY plane (pin 15). The timing diagram below, shows the high and low transitions that are needed on the single wire protocol to command the servo to go to its 'center' position. Changing the 'high' pulse duration controls where the servo will turn the horn connected to its drive shaft.



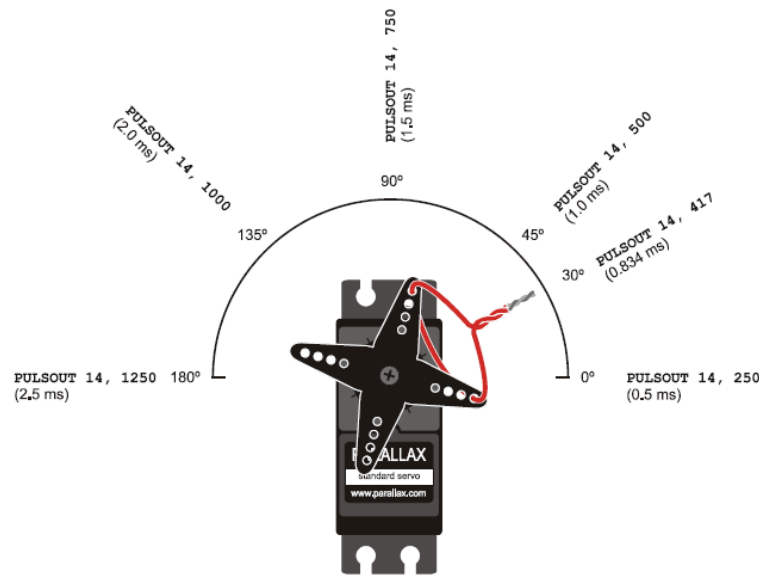
PBASIC allows me to use the (PULSOUT pin, duration) command followed by a pause command to create the signal. Just two lines! How cool is that?

```
PULSOUT Az_Pin, (XY + dXY - HalfB)
PAUSE 20
```

Now, you may have noticed the expression XY+dXY-HalfB used to control the duration of the pulse. In this version, I apply a 'trim' to the servo position. This means, whenever I see 750 in collected data ... that's the 'trim' center for that axis. Stored head readings do not contain 'trim' corrections.

So, here are some of the head positions together with the pulses you need to move the servo to that location:

³ not in this routine of course, but the routine that called it.



So, now that we know a little bit more about driving a servo, let's look at the code for *Move2_XYZ*:

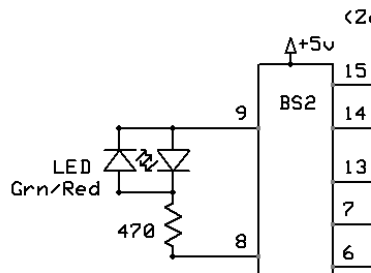
These lines ensure the values passed to the servos are 'safe' ... and will not damage the internal gears. (note – you can damage a servo by sending invalid commands!)

```
IF XY < MinServo THEN XY = MinServo
IF XY > MaxServo THEN XY = MaxServo
IF Z < MinServo THEN Z = MinServo
IF Z > MaxServo THEN Z = MaxServo
```

Then it's followed by a for-next loop structure that applies one signal to one servo, then another signal to the other servo. It repeats this output three times before returning. This has two effects; first, the current coming from the batteries is kept at a minimum. Only one servo at a time gets current. But, because it jumps back and forth between the two servo's ... it appears to the outside world that both servos are being moved together.

```
FOR X = 1 TO 3
  PULSOUT Az_Pin, (XY + dXY - HalfB)      ' send servo a pulse
  PAUSE 20
  TOGGLE 8
  PULSOUT Elev_Pin, (Z + dZ - HalfB)      ' send servo a pulse
  PAUSE 20
  TOGGLE 8
NEXT
```

By now, you may have noticed the `TOGGLE 8` command. This command flips the state of pin 8 from 0 to 1 ... or, 1 to 0. This may seem slightly cryptic, so let's refer back to our circuit schematic 'just part of it' and make use of the table that follows it to see what is happening:



Output on Pin		Light
9	8	Emmited
0	0	none
0	1	Green
1	0	Red
1	1	none

Truth Table

Now, when we (the programmers) make pins 8 & 9 LOW ... anytime afterwards, when we start to toggle pin 8, the light will blink green. When we make both 8 & 9 HIGH, the next time we toggle pin 8, the light will blink red. This works because these tiny devices are really 'diodes' ... a device that besides emitting light also passes current in one direction only. And since both diodes are reversed in the 'tri-color' led package, only one led will conduct at a time. So, why is it called a 'tri-color' led? If you toggle the color emitted back and forth between red and green thousands of times per second, the device appears 'yellow'. The 470 ohm resistor is there to protect the microcontroller from passing too much current in either direction.

The remainder of the subroutines are almost self-explanatory. I will depend on input from readers to 'fill' in the gaps, through questions that arise from time to time.