Changing times

For very little outlay, Ed Buckley's atomic clock interface takes the rather unfriendly serial data from an MSF receiver module and turns it into an RS232 time and data stream using a Basic Stamp. Output is a bcd data stream containing year, month, date, day, hour and minutes.

The *Basic Stamp* is a versatile, easy to programme microcontroller. As well as a stand alone unit the *Stamp* is also a very handy interface device- reading in non-standard data, performing some manipulation and then passing the data on in a more usable format.

The following design illustrates this function by combining a *Stamp 1* module with a readily available atomic clock module. To finish up, I will then describe how you can have your cake and eat it too – achieve low software development time by using the *Stamp1* and low production costs by using a standard *PIC*.

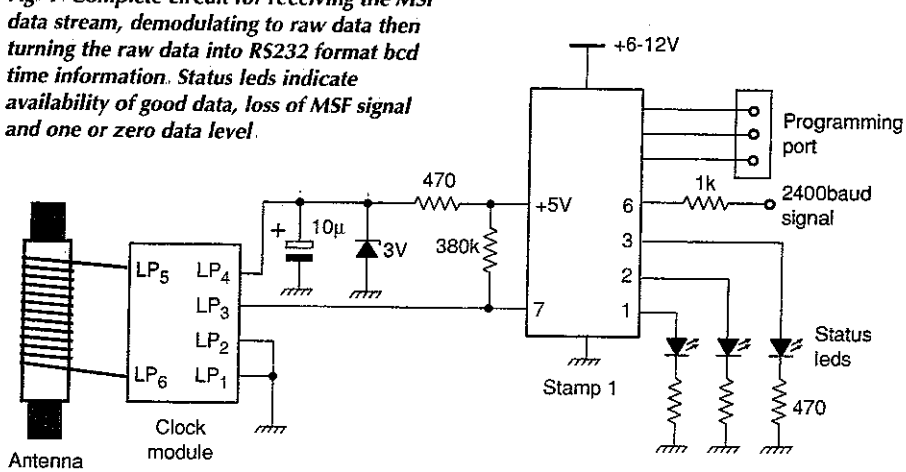
Design overview

We recently needed to fit a data logger to a series of truck-trailers. The data needed time stamping – no pun intended – but there was a serious risk of interruption of power to the real-time clock module.

The obvious solution was to incorporate an operator keypad to allow for clock resetting. As well as being expensive, this solution allowed the opportunity for operator error.

So we decided to look for a different solution. After looking at the atomic clock module now available, we decided to go down this route. Costs would be about the same as a keypad interface, but this alternative has the advantage that it is fully automatic.

Fig. 1. Complete circuit for receiving the MSF data stream, demodulating to raw data then turning the raw data into RS232 format bcd time information. Status leds indicate availability of good data, loss of MSF signal and one or zero data level.



Atomic clock summary

Atomic clock modules were featured in *Electronics World* March 1996. Here I will summarise the important features.

The off-air data stream is repeated every minute. High speed data is sent in the first 17 seconds, followed by null data that can be discarded. Next, low speed data is transmitted at 1bit/s. There is always a unique data format at the end of the minute period, which is 01111110_2. Data timing is shown in **Fig. 1** and the minute's data format in **Table 1**.

The circuit diagram with Stamp/Clock hook-up is shown in **Fig. 2**.

Writing the software

For our application, we require a burst of correct time data at 2400baud once a minute. The main system processor – a *Stamp 2* in this case – polls the signal line from the *Stamp 1* clock module. If valid data is present, it updates the system real-time clock.

The *Stamp 1* interfacing with the atomic clock is dedicated to reading and checking the validity of the incoming radio data. It only transmits the data if it proves valid.

We envisaged that the main problem would be loss of radio signal as the trailer moved around. For this reason the software always counts the number of data bits received from the last required data set until the unique format byte is received – in our case seven. If the counter is anything but seven, then a bit has been lost or found, the data is suspect and there is no transmission to the main system.

List 1 is the program listing. The input and output pins are allocated names to ease programme tracing and ram space allocated to the programme variables. The *Stamp 1* has up to 14 bytes of ram which may be addressed as bytes, for example b_3 or b_7, or words, as w_0 or

w_3, where $w_3=b_6+b_7$. Further, two bytes may be addressed by their individual bits, that is bytes b_0 and b_1 provide bits bit_0 to bit_{15}.

The dirs statement sets the pins to either input if zero or output if logic one; the msb comes first.

We initially look for the 01111110_2 signature by reading the individual data bits using the Get_bit routine into the variable *signal*. The variable *counter* is incremented every time a new bit is read.

Once the signature has been received we now know where we are and proceed to 'continue' where the *good_data* flag is set to false. Here we check to see if we have a good data set by looking at *counter*. If *counter* is does not hold seven, then the data is bad. If the counter does hold seven then the data is good and the *good_data* flag is set to true. This condition allows transmission of the time data ten times during the fast data period at the beginning of the minute period.

Now begins the slow speed data section. The first 16 seconds are unused and discarded. Next we log the year, month, day of month, day of week – which we discard – hour of day and minute.

Now the program returns to the beginning. If all is well the next eight data bits should be the unique signature and counter should return seven to allow transmission.

Most of the grunt work is handled by two subroutines – Get_bit and Read_convert.

Get-Bit. Get_bit decodes the data from the atomic clock module.

The data falling edge is first detected. Once detected, the programme pauses for 150ms before reading the data – if logic one is returned the data is zero and if zero, a one is returned. Note that the clock gives inverted data, which we correct in the line bit0=clk+1. The program then waits a further 200ms to ensure the clock data is in the stable high portion of its cycle before continuing.

Read_convert. At the end of each chunk of received data we take the opportunity to convert it into a decimal and then a binary-coded decimal format for easy reading by the system processor. Our main system clock uses bcd format so using bcd minimises the calculations later on.

Rounding up

To aid fault finding and add value, we included several leds – the i/o pins were there unused so why not spend 15p? The three leds have the following functions.

- Sync_flag led is on when the *good_data* flag is set, ie the previous minute's data was good.
- Bit_read flashes every two seconds – a stable on or off here indicates loss of incoming signal.
- Data_bit indicates whether the data bit received was a one or zero data.

Check_sum is a byte variable. It is cleared every minute and all data is added to give a

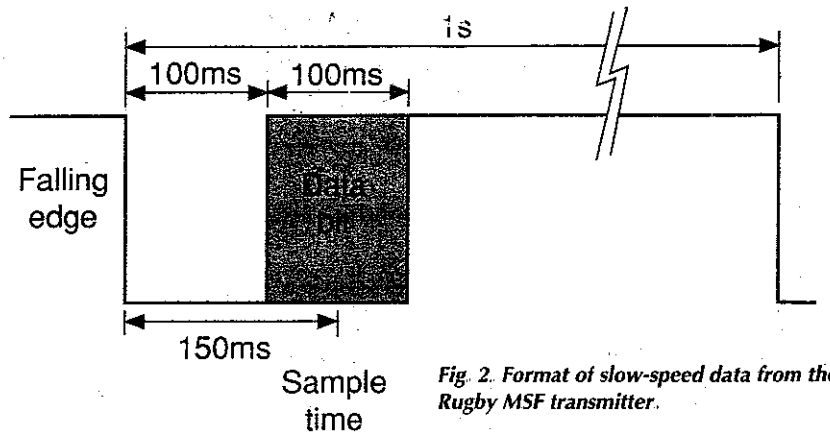


Fig. 2. Format of slow-speed data from the Rugby MSF transmitter.

Table 1. Time telegram of MSF transmitter – binary code for seconds 17 to 30 and data for March 1996. Note that each second value is one or zero. A one in second 17 for example represents 80 and a one in second 18 represents 40.

Second	Function	Value	Detail
0	Fast code	Not used in modules	
1	DUT1 code	Not used in modules	
2-16	–		
17	year (tens)	80	year 00-99, bcd
18	year (tens)	40	
19	year (tens)	20	
20	year (tens)	10	
21	year (units)	8	
22	year (units)	4	
23	year (units)	2	
24	year (units)	1	
25	month (tens)	10	month 01-12,bcd
26	month	8	
27	month	4	
28	month	2	
29	month	1	
30	day of month (tens)	20	day of month, bcd
31	day of month (tens)	10	
32	day of month	8	
33	day of month	4	
34	day of month	2	
35	day of month	1	
36	day of Week	4	day of week 1-7, bcd
37	day of Week	2	
38	day of Week	1	
39	hour (tens)	20	hour 00-23, bcd
40	hour (tens)	10	
41	hour (units)	8	
42	hour (units)	4	
43	hour (units)	2	
44	hour (units)	1	
45	minute (tens)	40	minute 00-59, bcd
46	minute (tens)	20	
47	minute (tens)	10	
48	minute (units)	8	
49	minute (units)	4	
50	minute (units)	2	
51	minute (units)	1	
52	always set to '0'	0	
53-58	always set to '1'	1	
59	always set to '0'	0	

resulting checksum which is used to detect errors between the *Stamp 1* and the main system processor – unsophisticated but works

What about the cake?

We have demonstrated the versatility of the *Stamp 1* as an interface. The Stamp scores over raw *PICs* in terms of speed of producing and debugging a simple programme.

If the application only requires a small number of units the *Stamp* is fine, either in its IC or discrete chip set format. For larger numbers however, the costs start to look a little heavy.

Realising this, Parallax, the manufacturers of the *Basic Stamp* has recently introduced a neat solution for *Stamp 1* users. By using the latest *Stamp 1* software and the Parallax *PIC* programmer, it is possible to programme a standard *PIC16C58* with both the *Stamp* interpreter and your programme. This gives low-cost development coupled with low cost production.

This system only works with the *Stamp 1*, *16C58* and Parallax's programmer, but it is a very useful tool for those quickie programmes that will fit into a *Stamp 1*. ■

Galleon, distributor of the clock modules, can be reached on 0121 359 0981.

List 1. Stamp 1 program to read and decode the Rugby MSF time clock.

```
' MSFCLCK.BAS
' Milford Instruments Original version dated March 1997

' Stamp decodes the one second pulses using Get_bit and
' keeps adding them to signal until the sync byte is received
' First 17 seconds are ignored and remainder are clocked into
' relevant bytes. Results are converted to BCD format before
' storing in bytes. Time is only transmitted at the start of
' the minute - if data is judged to be OK. Program counter
' holding number of seconds from last valid time data
' until sync byte is received.. should be 7.

Symbol clk          =pin7    'clock signal on pin 7
symbol rs232        =6       'rs232 signal on pin 6
symbol signal       =b1      'byte to receive data
symbol counter      =b11     'general counter variable
symbol good_data    =bit1    'good time message flag  0=false
symbol checksum     =b7      'data checksum variable
symbol sync_flag    =pin3    'in sync led
symbol bit_read     =2       'data bit received led
symbol data_bit     =pin1    'data bit rec. led, on for log. 1

dirs=%01001110     'set up the pin directions

'First look for the 01111110 signature
start:
 counter=0
 signal=0
 loop:
 gosub get_bit                          'go and get a bit
 signal=signal*2+bit0                   'add it to signal
 if signal=%01111110 then continue      'Sync byte received?
  'continue if not and increase counter by one
 counter=counter+1
 goto loop

continue:                               'sync byte now received
 good_data=0                            'reset valid data flag
 if counter<>7 then skip1               'check that we've kept in step
 if good data captured then set flag
 good_data=1
 skip1:
 counter=0
 sync_flag=good_data                    'light the in_sync led

 Now send the good data transmission whilst waiting to get
'past the fast code section at the start of the minute
'Total loop period approximately 1.4s
 for counter=1 to 10                    'send the good data at min. start
 if good_data<>1 then loop2             'check for good data flag
'Transmit if good data
 serout rs232,n2400,("T",b2,b3,b4,b5,b6,b7)
 loop2:                                 'if not jump to here
 bit3=good_data+1
 b9=bit3*35+100
 pause b9                               'get past the fast data section
 next

 now discard the first 16 seconds of info
 for counter=1 to 16
 gosub get_bit
 next

 Now start to read wanted data
'Reset the checksum variable
 checksum=0
 Now read the year data
 b8=7
 gosub read_convert
 b2=b10                 'result in b10 written to b2
 checksum=checksum+b2

 Now get the Month
 b8=4
 gosub read_convert
 b3=b10                 'result in b10 written to b3
 checksum=checksum+b3

 Now get day of month
 b8=5
 gosub read_convert
 b4=b10                 'result in b10 written to b4
 checksum=checksum+b4

 Now get the day of the week - and discard
 b8=2
 gosub read_convert

 Hour of Day
 b8=5
 gosub read_convert
 b5=b10                 'result in b10 written to b5
 checksum=checksum+b5

 And the Minute reading
 b8=6
 gosub read_convert
 b6=b10                 'result in b10 written to b6
 checksum=checksum+b6

goto start              'return for the next minute

 ========================================================
                  Sub-Routine to retrieve a bit
 ========================================================

Get_bit:
Marker:
 if clk=0 then read_bit                         'detect falling
edge
 goto marker

read_bit:
 pause 150                                      'wait until the
middle of the A bit
 toggle bit_read
 bit0=clk+1                                     'read the value on
clk into bit0
data_bit=bit0
 pause 200
return

 ========================================================
                  Read and Convert Routine
'========================================================
Read_convert:
 b9=0                                            Clear b9
 for counter= b8 to 0 step-1                    'one bit at a time
 gosub get_bit
 lookup counter,(1,2,4,8,10,20,40,80),b10   Get the
appropriate scalar
 b9=b10*bit0+b9                          Convert to true decimal
 next
 b10=b9/10*16                                    now convert to
bcd- first the high byte
 b10=b9//10+b10                         'now add the low byte
 return
```