The FLIMbox as of January 27, 2009

This tutorial applies to both the AVNET and the Digilent FLIMbox.

Introduction

Changes/updates

1. Addition of the scan/enabled line data at the FIFO output for synchronization (line or frame).
2. Use of the 100 MHz internal clock to generate lower frequencies for harmonics. The new 8 windows design.
3. Stopping counts but not the clock on FIFO overflow.
4. External clock at 20 MHz for synchronization with the 3-axis card.

We have made significant changes in the operation of the FLIMbox. The motivation for the changes in the operation of the scan/enable line was to provide a flexible input to the FLIMbox to synchronize with external sources, mainly the scanner and its various modes of operations (frame, line and circular). In the original Ryan’s design, the scan/enable line was simply turning on (or off) the FLIMbox, including the clock, resulting in elaborated synchronization protocols and no information about the data when the scan line was off. In the new design, the logic level of the scan/enable line is registered along the data stream and the software decides when the data is valid, such as during frame acquisition. This new mode of data acquisition has allowed us to use the FLIMbox for single point FCS (when a scan enable signal is not available), for circular scan and to put the FLIMbox in idle mode waiting for external data. Since data are always acquired, it is possible to sense when the external synchronization comes on, without establishing a predetermined sequence of events for the activation of the card.

The circular scanning with the ISS instrument required an additional change in which the clock of the 3-axis card was synchronized with the clock of the FLIMbox. Since the 3 axis card can synchronize to a 20MHz clock, we derived a clock signal from the 100MHz clock of the Avnet card and we had this clock as the master clock for both the 3-axis card and the FLIMbox achieving full data and scanning synchronization.

During this development, we noticed that some of the cards were working and some others failed. After reviewing all the circuits, we realized that most of the problems were with some critical high frequency internal operation of the FLIMbox that became unstable, giving an erratic behavior. The critical value is reached when the internal operational frequency of the FPGA reaches the value of 190MHz or above. This value is obtained when the FPGA generates 4 windows at 48 MHZ (4*48MHz=192MHZ). The maximum reliable operating frequency for all our boards is just too close to the critical value. This prompted us to change the internal design so that this limit is never reached.
One critical change was to modulate the light at high frequency (For example 80 MHz or 40 MHz) but to run the FPGA at half that frequency. For example, with the 2-photon laser (at 80 MHz) the laser signal to the FLIMbox is first divided by 2 so that the internal operation of the FPGA is now 40 MHz. When using 4 windows, the maximum internal operating frequency is 4*40MHz=160 MHz, well within the limit of operation of the FLIMbox. This division by 2 of the external synchronization fixed the problem, however, added some limitations to the operation of the FLIMbox. These limitations are hardly seen in the normal phasor analysis. However, if we want to exploit the multi harmonic capabilities of the digital frequency-domain approach, for example using the Fianium white laser, using this simple solution (dividing the clock by 2) we will allow using a rich spectrum of high modulation frequency. A problem with the 4 windows design is the 4th harmonic is multiplied by zero so that we have no modulation at this frequency. For example, when dividing the clock by 2, to safely operate the FLIMbox, the following table gives the available harmonics.

<table>
<thead>
<tr>
<th>Laser modulation</th>
<th>Fundamental and harmonic available</th>
</tr>
</thead>
<tbody>
<tr>
<td>80 MHz (4w, base at 40 MHz)</td>
<td>80 MHz, nothing at 160 MHz and weak harmonic at 240 MHz</td>
</tr>
<tr>
<td>40 MHz (4w, base at 20 MHz)</td>
<td>40 MHz, nothing at 80 MHz, 120MHz, 200MHz,</td>
</tr>
</tbody>
</table>

Although the card works in a stable mode, the lack of true second harmonic of the laser (the 80 MHz repletion rate laser) can be problematic.

For this reason, we explored a different design that operates with 8 windows, but at smaller (20 MHZ) laser modulation frequency. The new version operating at 20 MHz which gives the following set of frequencies

| 20 MHZ (8w, 2 channels) | 20 MHz, 40 MHz, 60 MHz, 80 MHz, 100 MHz, 120 MHz, 140 MHz, nothing at 160 MHZ, 180 MHz, 200 MHz, 220 MHz |

The only missing harmonics is the 160 MHZ, but the 140MHz and the 180 MHZ are available.

**Operation of the FLIMbox**

After this introduction, we can now better understand the new firmware and the various checkboxes in the FLIMbox page of SIMFCS.

The stable firmwares that divide the clock by 2 and have the synch signal for the 3-axis card are the following

FLIMhet_0_6_2_2ndHarmonic_SynchOut.fbf (to use for 48MHz modulation on the second harmonics)

FLIMhet_0_6_2_IntClk_SynchOut.fbf (to use for single point FCS when lifetime is not needed)

Flimhet_0_6_2_80MHz_2ndHarmonic.fbf (to use with the 2-photon laser)

FLIMhet_0_6_5_8w_2ch.fbf (to use for two channels, 20 MHZ laser modulation, multi harmonic mode)

(more to be added)
**TI:Sa laser**

With the 80 MHZ TI:Sa laser we use the firmware “80 MHz laser, 4 windows”, using the second harmonics. The firmware works with 4 windows, using a divide by 2 for the laser. In this way the card works at 40 MHz internally. To use the card in this mode you must check “Use second harmonic” and the “Digilent decoder”.

**Old 48 MHz operation**

There is no stable working firmware for 4 windows and 48 MHz. Some of the cards are working at this speed, for example the card we have on the Zeiss system. Other cards we tested have instabilities.

**New 20 MHz operation**

To operate with the Olympus and the ISS-Alba, we use the multi-harmonic harmonic firmware, 8 windows, with the Digilent decoder. This is our best firmware and we are further developing this new design only.

**Programming**

**What is the Digilent decoder?**

A major improvement of the new design with respect to the original Ryan’s design was to use the scan/enabled line as a separate line in the FIFO output rather than using to turn on/off the operation of the FLIMbox. For this operation we needed one more line. According to the Ryan’s design, we were using 8 lines for the phase (granularity of 8 bits), 2 lines for the ccTime (together we had 10 lines) and 6 lines for the 2 channels (1 bit to indicate if a photon is present) and 2 bits to indicate in which window (of 4 possible) the photon appeared. Apparently there were not free lines in a word of 16 bits to be used for the scan/enabled. However, an analysis of the possible combinations of windows and photons in the channels show that not all 64 (6 bit) combination are possible, but the possible combinations channels and windows are less than 32. Therefore, with a proper decoder, only 5 bits are sufficient to describe the possible combinations leaving one line free for the scan enabled. For historical reason, this approach goes under the name of “Digilent decoder”. This decoder approach was implemented also in the Avnet card.

For the operation with 8 windows, we calculated that we needed 2 more bit (the decoder uses 128 possible combinations). However, we also realize that we are never using 256 points for the phase but only 64. For this design the bits of the FIFO output are separated as follow: 6 for the phase, 2 for the ccTime, 1 for the scan/enable and 7 for the decoder (channels and windows).

**How to use the decoder?**

The decoder is a simple table from the point of view of the programmer. For example, for the 7-bit decoder, the table is 128 entries. Given one entry from 0 to 127, the output (8 bits) is divided in the channel and window for that channel. There is no time involved in the decoding so that there is no slow down for using it.

**How to use the scan/enabled line?**
In the new version of the firmware using the decoder and the scan/enable line, each line of the FIFO is masked and decoded so that the programmer has access to the value of the phase, time, scan enable, photon and window. This is done with a simple “one line” call.

In our program, we continuously monitor the counts intensity, which are displayed on the monitor and the scan line. The logic of using the scan line depends on the particular mode of operation.

For continuous operation (typically FCS), the scan line is not used. When the data acquisition starts, the global time variable is zeroed and acquisition continues until the specified amount of time (or number of photons) is reached.

For synchronization with the scanner signal, the scan line is continuously polled. Upon a transition of the scan line (either positive or negative going edge, depending on the instrument) the data are considered valid the global time starts and counts at the proper time are put in the “frame” or “line” formatter. Since we know a priori when the frame or line will complete, the system will wait for the next transition of the scan line to indicate that either a new frame or a new line is available.

This synchronization method works with all systems we tried since we only need the start of frame or the start of line. For example, the Olympus gives a start line that has a pre-start delay. However, this time is known so we know when the data is valid. For the Zeiss we have a brief pulse indicating the beginning of frame. Even if this pulse could be short, we buffer this pulse in the card so that it is always read.

As you can see, using a proper logic and knowing the length (in time) of the frame or line, the program can decide when data are valid and display the data in the image window without a special predetermined sequence of operations. The only requirement is that we know the time for a frame or line but not when the acquisition of the frame is starting.

We found that the synchronization with the frame acquisition is now perfectly reliable and there are no more pixels or line missing.

**How to use the multi-harmonic capabilities?**

The harmonic content is contained in the 64 phase histogram for each channel. We perform an FFT to extract up to 32 harmonics. Generally, we only display 2 of them. Since the ISS software has the capability to store a multi-frequency image file, I suggest that you maintain the software and decide how many harmonics you want to store. Remember that if you save the raw data file, this file can be reprocessed to extract any harmonic you want. For the Ti:Sa laser operation we found that up to the 8-10th harmonic still gives reliable phase histograms, bringing the operation of the system to about 640 MHz. The limit seems to be the detector. For the 20 MHz operation we collected and analyzed data up to about 320 MHz which corresponds to the 16th harmonics. However, it is very likely that we will lower even more (to 10 MHz) the basic repletion rate of the laser so that we will use up to the maximum of 32 harmonics. The 320 MHz limit seems to be due to the laser modulator and in part to the detector.

**How the FIFO overflow problem was solved**

In the Ryan’s implementation, when the FIFO overflowed, data were missing and the entire time sequence could not be reconstructed causing a non-valid frame and/or pixel shifts. This was annoying and gave the impression that the system can only work at low counts. We changed the internal logic to
stop the input counts when the FIFO overflow is reached. This method gives a continuous time stream, although some of the counts are missing. This is what is happening with all photon counting cards that experience saturation when the card cannot process the photons. However, using this simple method, the user only sees a deformation of the intensity but no pixels are missing or frame must be disregarded.

We carefully tested our system using an electronic signal generator, since it is much more difficult to understand when saturation occurs using random arriving photons. For one channel operation, the maximum sustained frequency was 1.2MHz and for 2 channels operation was 600Khz. However, in the way the system works now, there is no sharp transition between normal mode and saturation. We found that from the user point of view, this simple solution (inhibiting the input when the FIFO overflows) fixes the problem. Note that saturation only occurs for high sustained rates. This is not different that saturating the ADC of the microscope acquisition card (Olympus or Zeiss) or the pile-up that occurs in other photon counting cards.

**Description of the software. Overview**

The following figure shows the current FLIMbox screen for data acquisition. This screen is still changing as simplifications and stabilization of the firmware bring new capabilities.

![FLIMbox screen](image)

**Description of functions and flags**
**FlimBox_init**: loads the firmware selected in field at right and initializes the DLL. This is done automatically upon program starting and it should be used only when the firmware is changed.

**FlimBox_close**: stops and closed the DLL. It is automatically called on program termination. This function should not be used for normal operation.

**Use Digilent decode**: this flag is associated with the specific firmware. In principle is not needed since the firmware determines whether or not the decoder must be used. It is there for historical reasons.

**Use second harmonic**: This flag is also associated with a specific firmware. It simply tells that the frequency of operation of the FLIMbox is half the value of the frequency at the I/O clock line.

**Subtract background**: It subtracts a previously store phase histograms so that a true background subtraction could be performed.

**Save data on stop**: automatically saves the raw data file when the stop button is pressed. We tend not to use this feature.

**Process enabled only**: This flag is used to construct the format the data into a frame of a given size and time length.

**Use 8 windows**: This flag is also superfluous since the firmware used is associated with 4 or 8 windows. It will be deleted when the firmware will stabilize.

**Integrate**: Add all the frames for display only. The raw data is never averaged. Data can be reprocessed to produce FLIM movies or to integrate all data in a single FLIM image.

**Scanner at the center**: When data acquisition is stopped, you can place the scanner at the center of the frame for alignment purposes.

**Harmonics 1 and 2** fields select which harmonics will be displayed. Remember that the raw data contains all the harmonics and that the data can be reprocessed if you decide to analyze a different harmonic.

The calibration factors are automatically calculated when the calibration operation is executed as described later.

For image acquisition there are a few things that must be set.

In the field at the bottom right, select the scanner card. 3 options are available: Olympus, Zeiss and ISS 3-axis card. The program has in memory the setting of the scanners for the different instruments so that by selecting the pixel time all the rest of the parameters is calculated automatically. The reason to have many fields is for development only. In a stable version of the software one single field is sufficient to set automatically all other fields.

For normal operation (with the ISS instrument) a measurement starts by pressing the data in button and terminates by pressing the stop button. Data can then be saved as raw data, calibrated according to the lifetime standard and stored as calibrated (or referenced) data. All the rest of the operation is pretty automatic. With other instruments, we put the card in the idle mode. The only thing you have to do is
to match the pixel time between the two instruments. When the data acquisition starts in the Olympus or Zeiss, the software senses the scan enable line and start to memorize the data. The user do not need to do anything particular other that saving the data when finished.

**Programming tips**

In this part we report crucial parts of the routines used to read the FIFO data through the DLL calls and to do the first step of data processing.

The main program starts a timer interrupt that calls the DLL for data ready. The DLL has a very large buffer so that the timing of the interrupt is not critical. The data from the DLL buffer must be decoded, separated in two channels and the valid data according to the scan line logic must be assigned to the two huge vectors.

Please note how the data from the DLL is handled. When the DLL buffer is full, you must check for the even or odd status of the counter since we are counting bytes. On odd bytes, the next part of the word is coming with the next buffer.

We are attaching the entire FLIMbox program, but in the following we report some of the crucial parts.

```
//this the decoder table for the 8 windows operation
var decode8W:array[0..80] of byte =
($00,
 $01,$03,$05,$07,$09,$0b,$0d,$0f, //w0-w7, ch1
 $10,$30,$50,$70,$90,$b0,$d0,$f0, //w0-w7, ch2
 $11,$13,$15,$17,$19,$1b,$1d,$1f, //w0-w7, ch1, w0 ch2
 $31,$33,$35,$37,$39,$3b,$3d,$3f, //w0-w7, ch1, w1 ch2
 $51,$53,$55,$57,$59,$5b,$5d,$5f, //w0-w7, ch1, w2 ch2
 $71,$73,$75,$77,$79,$7b,$7d,$7f, //w0-w7, ch1, w3 ch2
 $91,$93,$95,$97,$99,$9b,$9d,$9f, //w0-w7, ch1, w4 ch2
 $b1,$b3,$b5,$b7,$b9,$bb,$bd,$bf, //w0-w7, ch1, w5 ch2
 $d1,$d3,$d5,$d7,$d9,$db,$dd,$df, //w0-w7, ch1, w6 ch2
 $f1,$f3,$f5,$f7,$f9,$fb,$fd,$ff); //w0-w7, ch1, w7 ch2
```

```
procedure use4windows(dataword:word; var cctime, ccphase, ka, kb, ph1A, ph2A, data_enabled: integer; var time, tlast: int64);
var ph1, ph2, decoderbits: integer;
begin //of 4 windows processing
  CCTime := dataword and $03FF;
  CCPhase := dataword and $00FF;
  if tlast <> -1 then
    begin
      if CCTime = 0 then
        time:= time + 1024 - tlast
      else
        time := time + CCTime - tlast;
    end;
  tlast := CCTime;
  if FlimB.checkbox15.checked=false then
    begin
      Ph1 := (dataword shr 14) and 3;
      Ph2 := (dataword shr 12) and 3;
      Ph1A := (dataword shr 11) and 1;
      Ph2A := (dataword shr 10) and 1;
    end
  //end of flimbox decode
  else
    begin
      if (dataword and $400)=1024 then data_enabled:=1 else data_enabled:=0;
      DecoderBits := (dataword shr 11) and $1f; //get the 5 significant bits
      Decoder(DecoderBits, ph1, ph2, ph1A, ph2A);
    end;
  ka:=0;
  kb:=0;
  if Ph1a = 1 then
    begin
      ka := (CCPhase + (Ph1 shl 6)) mod modn;
      ka :=(modn-1-ka) div divn;
      a[ka div 16] :=a[ka div 16]+ph1a;
    end;
end;
if Ph2a = 1 then
begin
  kb := (CCPhase + (Ph2 shl 6)) mod modn;
  kb:=(modn-1-kb) div 2;
  b[kb div 16] := b[kb div 16] + ph2a;
end;
end;

//This is the most important routine for data acquisition. It reads the dll, decodes the fifo and send the data to the huge vectors

function FlimBoxgetnextcnt:integer;
//in save huge mode or image_mode must return myindex because used by the image and fcs routines
var i,j,k,count2,ii,jj,ka,kb,bufindex,tau,cctime,ccphase,left,transfer,decoderbits:integer;
dataword:word;
lt:int64;
aux:double;
m1,m2:double;
begin
  result:=0;
  if newcard=15 then
  begin
    result:=Digilentgetnextcnt;
    exit;
  end;
  fillchar(a,4*8,#0);
  fillchar(b,4*8,#0);
  if flimb.checkbox4.checked then //simulation mode
  begin
    if mydbread=nil then
    begin
      motor_flag:=1;
      read_test_file;
    end;
  //old code
  { if enableddata=false then retval:=0
    else//this is done in the simulation mode to enable data when the simulated enabled mode is on
      if flimb.checkbox6.checked=false then FlimBox_simulate
        else FlimBox_simulate_4phases;
  }
  //end of old code
  //new code
  k:=(random(100000)+10000)*2; //buf is in word, mydbread is in byte
k:=256\times256;
intb:=0;
for j:=0 to k-1 do
   begin
       buf[intb]:=mydbread^[intread]+(mydbread^[intread+1] shr 8);
       intb:=(intb+1) mod (sizeofb);
       intread:=(intread+2);
       if intread>=maxread then
       begin
           FLIMb.memo1.clear;
           fillchar(huge_data^,mysize*2,#0);
           fillchar(huge_data1^,mysize*2,#0);
           nframes:=0;
           intb:=0;
           intread:=0;
       end;
   end;
   retval:=k;
//end of new code
end
else
begin
   retval:=FLIMBoxRead(@buf,sizeofb*2); //call the DLL reading routine
   if odd(retval)   //do not process this data, it is corrupted due to glitch in the dll routine
   then
       begin
           if (badvalue>0) and (badvalue<256)
           then
               begin
                   move(bufbyte[0],bufbyte[badvalue],retval);
                   move(saveatbuf,bufbyte,badvalue);
                   retval:=(retval+badvalue) div 2;
                   badvalue:=0;
               end
           else
               begin
                   badvalue:=retval;
                   exit;
               end
       end
       else retval:=retval div 2;
   if retval>sizeofb
   then retval:=sizeofb;
end;
result:=retval;
delta_count1:=0;
delta_count2:=0;

// now process the saved buffer for counts
lt:=timemonitor; //store the initial time before any processing
for j:=0 to retval-1 do
  begin
    dataword:=buf[j];
    if flimb.checkbox2.checked
      then
        use8windows(dataword,cctime,ccphase,ka,kb,ph1A,ph2A,data_enabled_monitor,timemonitor,tlastmonitor)
      else
        use4windows(dataword,cctime,ccphase,ka,kb,ph1A,ph2A,data_enabled_monitor,timemonitor,tlastmonitor);
        delta_count1:=delta_count1+(ph1a);  //count all, even the doubles.  This is for display only
        delta_count2:=delta_count2+(ph2a);
    if save_huge then // modify the huge vector only during data acquisition, used for FCS
      begin
        aux:=timemonitor*r;
        myindex:=round(aux) mod mysize;
        myindex1:=myindex;
        if image_mode=false then //do not write the huge vector in image mode since it is written
          in the processoneframe
            begin
              huge_data^[myindex ]:=huge_data^[myindex ]+ph1a;
              huge_data1^[myindex1]:=huge_data1^[myindex1]+ph2a;
            end;
          end; //of save_huge
      end; //of retval loop
lt:=timemonitor-lt;  //this is the total time advancement of this buffer
if lt<>0 then
  begin
    if data_enabled_monitor=1 then Flimb.label27.caption:='Data enabled' else
    Flimb.label27.caption:='Data disabled';
    m1:=a[0]-a[2];
    m2:=a[1]-a[3];
    ac1_monitor:=sqrt(sqr(m1)+sqr(m2));
    ph1_monitor:=(180/pi)*arctan2(m1,m2);
    if dc1_monitor<>0 then md1_monitor:=ac1_monitor/dc1_monitor else md1_monitor:=0;
    m1:=b[0]-b[2];
    m2:=b[1]-b[3];
    ac2_monitor:=sqrt(sqr(m1)+sqr(m2));
    ph2_monitor:=(180/pi)*arctan2(m1,m2);
    if dc2_monitor<>0 then md2_monitor:=ac2_monitor/dc2_monitor else md2_monitor:=0;
dph_monitor:=ph1_monitor-ph2_monitor;
dcl_counterg:=dcl_monitor*flimBox_freq*rr/lt;  //this is the value used for the display
monitor
dc2_counterg:=dc2_monitor*flimBox_freq*rr/lt;  //and accounts for missed counts
end;
result:=retval;
if image_mode or save_huge then  //  save the buffer in the working buffer
begin
  if flimindex+retval<flimdblength then
    begin
      move(buf[0],mydb^[flimindex],retval*2);
      flimindex:=(flimindex+retval) mod flimdblength;
    end
  else
    begin  //check for buffer overrun
      transfer:=flimdblength-flimindex;
      left:=retval-transfer;
      move(buf[0],mydb^[flimindex],(transfer)*2);
      flimindex:=(flimindex+retval) mod flimdblength;
      move(buf[transfer],mydb^[0],left*2);
      myflimpr:=0;
      flimindex:=0;
    end;
  end;
end;