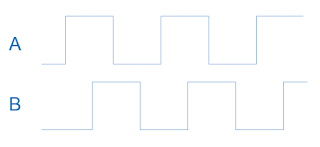
**[The LS7366 Quadrature Counter](http://davidjabon.blogspot.com/2016/04/the-ls7366-quadrature-counter.html)**

Quadrature encoded signals have many applications ranging from simple input/output control (e.g. a volume control on a radio or a knob on a toaster oven) to motor control (e.g., speed and position).  A quadrature encoded shaft allows one to track the position and direction of the rotation.  
  
Quadrature encoded signals look like the diagram below.  One signal is 90 degrees out of phase with the other, and this phase difference allows one to know which way the shaft is turning (A will lead B in one direction and B will lead A in the other direction).  To determine the position one counts the edges positively or negatively depending on the direction.  I will assume that you are familiar with how quadrature encoded signals work in this post.

[](https://1.bp.blogspot.com/-Gg3nXpBo_c4/Vs9PxfgGTWI/AAAAAAAAC3Y/rQVjaG_v8Z4/s1600/Quadrature_Signal.PNG)

Keeping track of rising and falling edges of the A and B signals can take up significant resources on a microcontroller.  There is a very nice [encoder library for the Arduino,](http://www.pjrc.com/teensy/td_libs_Encoder.html) and the Arduino [resources](http://playground.arduino.cc/Main/RotaryEncoders) are very helpful and adequate for many purposes.  However, there are circumstances when a microcontroller cannot keep up with the pace of the signals, or it becomes too arduous to deal with multiple encoders.  Or you may want to do something that is timing dependent on the microcontroller (e.g. taking sensor measurements)  and cannot allow interrupts for handling the quadrature signals.  
  
It is in these latter situations that you might be interested in the [LS7366 Quadrature Counter](http://www.lsicsi.com/pdfs/LS7366.pdf) interfacing chip.  This IC takes care of all the counting, and the microcontroller communicates with the chip via a standard four wire SPI interface. The chip even has some programmable output pins.  Everything I will share with you (and considerably more) can be found in the [datasheet](http://www.lsicsi.com/pdfs/LS7366.pdf).  However, in sharing my exploration of this chip, I hope I can save you some time and energy.  I found that this chip is loaded with features, and all of its capabilities and features are overwhelming at first.  In some sense, this post is a reader's guide for the datasheet.  In addition, I wrote [a simple Arduino library for the LS7366](https://github.com/davidjabon/LS7366).  
  
In the summary below, I will occasionally quote from the data sheet. All quotations are directly from the datasheet.  
  
**Registers**  
  
There are six registers: two configuration (mode) registers called MDR0 and MDR1, a status register STR, a data register DTR, a count register CNTR, and a special register OTR which designed as a "convenient dump site for instantaneous count data which can then be read without interfering with the counting process." As far as I can tell, all readings of count data come from the OTR register in the sense that the if one requests to read the CNTR register, the contents of the CNTR register are transferred to the OTR register first, and then the OTR register is read out.   
  
The registers have codes:

| Register | Code |
| --- | --- |
| MDR0 | 001 |
| MDR1 | 010 |
| DTR | 011 |
| CNTR | 100 |
| OTR | 101 |
| STR | 110 |

**Operations**  
There are four register operations: Clear, Read, Write, and Load.  Not all the operations can applied to all the registers, and there is a table in the datasheet that specifies how each operation applies to each register. For example, the data register DTR cannot be read.  It can only written to.  Another example: you cannot write to the counter register CNTR.  However, you can write to the data register DTR and then load the data register DTR into the counter register CNTR. Most of other operations are intuitive.  
  
The operations have codes:

| Operation | Code |
| --- | --- |
| Clear | 00 |
| Read | 01 |
| Write | 10 |
| Load | 11 |

**Opcodes**  
Eight bit opcodes are straightforwardly constructed from the operation codes and the registers codes:  Bits 7 and 6 are the operation code. Bits 5, 4, and 3 are the register code.  Bits 2, 1, 0 are not used.   
  
Here is a convenient summary of the opcodes:

#define CLR\_MDR0 0x08

#define CLR\_MDR1 0x10

#define CLR\_CNTR 0x20

#define CLR\_STR 0x30

#define READ\_MDR0 0x48

#define READ\_MDR1 0x50

#define READ\_CNTR 0x60

#define READ\_OTR 0x68

#define READ\_STR 0x70

#define WRITE\_MDR1 0x90

#define WRITE\_MDR0 0x88

#define WRITE\_DTR 0x98

#define LOAD\_CNTR 0xE0

#define LOAD\_OTR 0xE4

**SPI communication and counter data width**  
  
The chip uses a completely standard mode 0 SPI interface (polarity 0, phase 0), most significant bit first. The command byte is followed, if needed, by data either to be written or to be read. A somewhat unusual feature is that the CNTR, OTR, and DTR can be set to be 1, 2, 3, or 4 bytes wide.  Evidently this was done to keep the communications as compact as possible.  The default width is 4 bytes.  The data width is set in the mode register 1.  
  
**Mode registers**  
The wealth of options is laid out in the mode registers. There are two mode registers, named MDR0 and MDR1 in the datasheet.  Each is divided into sections.  
  
**Mode Register 0, MDR0**  
Bits 0 and 1 control the count mode (4 counts per encoder cycle, 2 counts per cycle, 1 count by cycle, non-quadrature mode).  Bits 3 and 4 control the "running mode."  There are four running modes: free running count, single cycle (counter becomes disabled if a carry or borrow occurs in the counter), range limit (counter freezes when count reaches the value in data register DTR), and module-n mode (input clock is divided by (n+1) where n is the value in the data register DTR).  Bits 4, 5 and 6 of MDR0 relate to the index, and I will omit the description.  Bit 7 adjusts the filter clock division.  
  
**Mode Register 1, MDR1**  
Bits 0 and 1 control the data width of the counter.

| data width | Bit 1 | Bit 0 |
| --- | --- | --- |
| 4 | 0 | 0 |
| 3 | 0 | 1 |
| 2 | 1 | 0 |
| 1 | 1 | 1 |

So the data width is 4 minus the binary number in bits 0 and 1.  Bit 2 can enable or disable counting.  Bit 3 is not used. Finally, bits 4 through 7 set the flag mode (see next section). The flags can be set on the index, on comparison with the data register, on borrow and on carry.  
  
**Output pins**  
There are two output pins, called flag pins.  Pin 8 is called LFLAG (latched flag) and is open drain. Pin 9 is called DFLAG ("dynamic flag") and has a push-pull output. The DFLAG idles high.   The dynamic flag, if enabled, will only go low when the condition, set in MRD1, holds.  For example, in compare mode, the DFLAG will go low only when counter register CNTR exactly equals the data register DTR and will stay high otherwise. In contrast, the LFLAG latches to low when the appropriate condition is met and will stay low until the status register is cleared.  
  
I have posted a simple Arduino library at [my github](https://github.com/davidjabon/LS7366).  
  
Here are photos of my testbench.

|  |
| --- |
| [https://2.bp.blogspot.com/-Eyr6dGVp_J0/VyUVGEfNZhI/AAAAAAAADEM/Pa27OnCndo4f_90f1mziAzuTrEz1hvJZQCKgB/s200/IMG_20160430_151411116.jpg](https://2.bp.blogspot.com/-Eyr6dGVp_J0/VyUVGEfNZhI/AAAAAAAADEM/Pa27OnCndo4f_90f1mziAzuTrEz1hvJZQCKgB/s1600/IMG_20160430_151411116.jpg) |
| Right side LS7366 test bench |

|  |
| --- |
| [https://3.bp.blogspot.com/-Dby1BlKMCX0/VyUVBxnUwzI/AAAAAAAADEI/IRbn8_9TrlAYTT6jH9LjWouzcJlXAHJBQCKgB/s200/IMG_20160430_151911383.jpg](https://3.bp.blogspot.com/-Dby1BlKMCX0/VyUVBxnUwzI/AAAAAAAADEI/IRbn8_9TrlAYTT6jH9LjWouzcJlXAHJBQCKgB/s1600/IMG_20160430_151911383.jpg) |
| Left side of LS7366 test bench |