Shift Registers

Issue 3– 13th Sept. 2013

This technical article explores the humble shift register and how it works.

The Shift Register

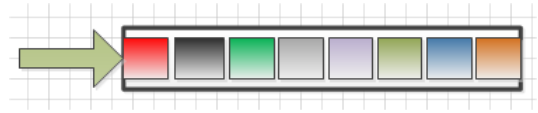
Wiki describes a shift register as "a cascade of flip flops, sharing the same clock, in which the output of each flip-flop is connected to the "data" input of the next flip-flop in the chain, resulting in a circuit that shifts by one position the "bit array" stored in it, *shifting in* the data present at its input and *shifting out* the last bit in the array, at each transition of the clock input", but what does that actually mean?

Imagine a tube laying on a table. Into the left of the tube an Orange cube is inserted.

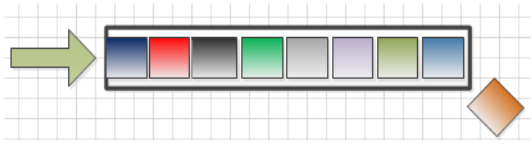
Next, a blue cube is forced in from the left.

The Orange cube has been pushed one place (cube width) to the right; it has been "shifted" right.

If we insert another six cubes into our tube, we end up with this:



The tube is now full, so if we force in one more cube, the orange cube will fall out to make room for the new one.

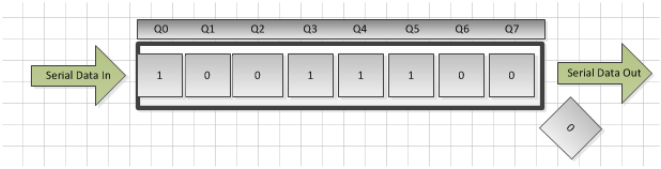


The Orange cube has been "shifted" along the tube to the right one position at a time until it fell out of the end.
This is the "data out" or "carry out".

A shift register basically works this way, expect instead of using cubes we shift "bits"; logic "1's" and "0's".

This type of shift register is called a FIFO or First In First Out. The Orange cube was the first one inserted and the first to fall out of the tube.

If we were to take the tube analogy and represent this using a real shift register component, it would look something like this:



The above diagram has a label for each position, Q0 to Q7. These positions are like holes cut in the tube so that you can see what logic value is in each position, and this is where shift registers come into their own in electronics; the ability to see what is in each position in the shift register.

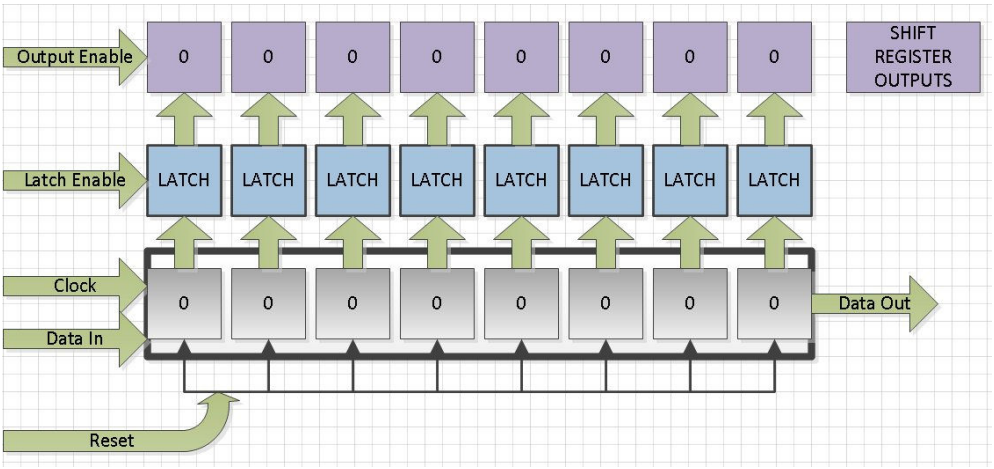
The shift register above is, as already stated, a FIFO type. But the above can also be called a SIPO; Serial In, Parallel Out. One of the uses for shift registers are for Serial to Parallel (output shift registers), and Parallel to Serial conversion (input shift registers).

There are many different makes and types of shift register IC available for the hobbyist, but two of the most useful are the 74HC595 and 74HC597 and there are several reasons why these two ICs are really rather useful.
These two ICs are fairly common and easy to obtain and can be very inexpensive, but they also have output or input latches that are missing on many other devices and this makes these ICs very useful indeed. This requires some explanation.

We stated that one of the things that makes a shift register useful is the ability to see what logic values is present in each position within the shift register chain, and it is. However, this can also be a serious disadvantage. Using the previous tube and cube example, as the Orange cube is shifted left to right, one position at a time, it's "visible" each time it's in it's new position.

See in the above graphic how the Orange cube is visible in each position as it's shifted from left to right ?

If we now imagine that the Orange cube represents a logic "1", and this controls an electronic device like a lamp or a relay, you will get a "chase" affect as the shift register chain moves to the right or is "clocked".

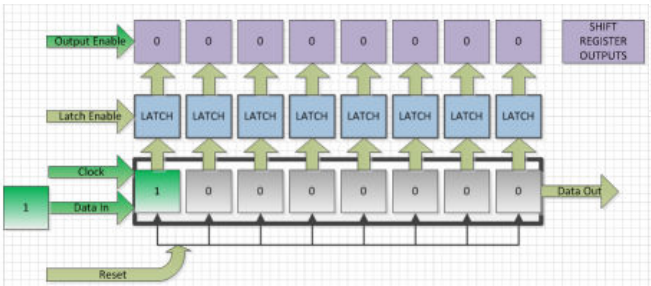


Output Shift Register Block Diagram

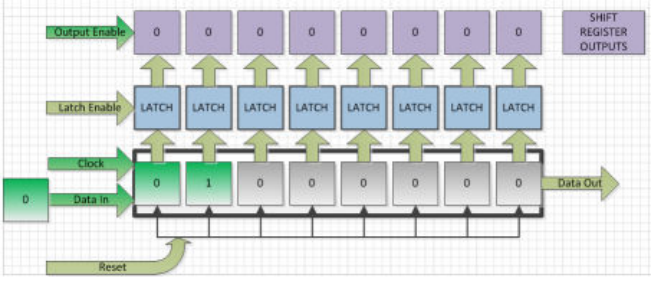
Now it maybe that this is a desirable affect, after all "running light" displays as they are known are quite common. However, in some applications this affect is certainly not desirable. The 595 device has a series of latches, one for each output of the shift register chain, and these latches act like a tiny memory for each output.

The following diagrams shows the basic internal working of a 595 output shift register.
Data (logic "1's" and "0's") are presented on the Data In pin. Each time the Clock pin is pulsed, the entire chain moves one bit to the right and then the data on the "Data In" pin is placed in the first location.

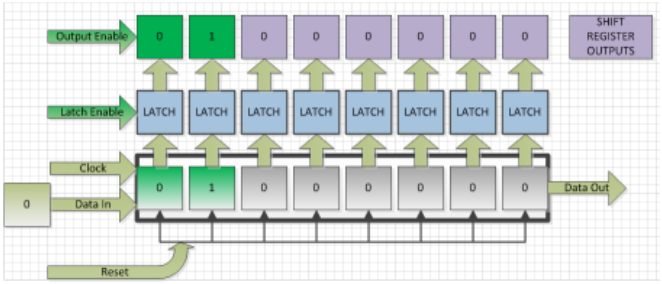
The diagram below shows what happens after a logic "1" is clocked into the shift register. Notice that the outputs are still all showing "0". Also notice that the "Output Enable" is active. When enabled, the contents of the output latches are connected to the output pins on the IC. When disabled the output pins are isolated and so "float" or are of "high impedance".



Next, a "0" is clocked into the shift register and once again, the chain shifts one bit to the right but all the outputs still stay at logic "0".

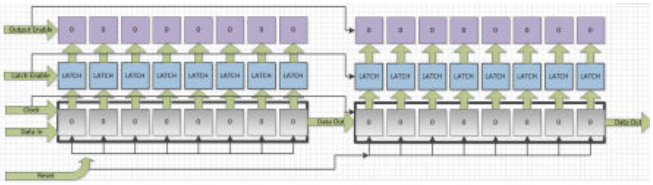


Now the Latch Enable is pulsed and the data in the shift register chain flows into the output latches so is now visible.



Longer Shift Register Chains

The shift register described above is an 8-bit register because it can accommodate 8-bits of data before it overflows. However, you can chain a theoretical unlimited number of these devices together to form chains of hundreds or even thousands of bits long.



The above diagram shows how two or more shift registers can be connected in series to create longer chains. Notice how all the signals are connected in parallel (Output Enable, Latch Enable, Clock and Reset) BUT the "data in" on the second register, is fed by the "data out" from the first.

I hope that you can see some of the great advantages to using shift registers. I/O pins on PICs and other MPUs are often in short supply, but if you need more than four or five control outputs, a shift register could be ideal.

Of course, there's no free lunch with shift registers and one of the problems is that the longer the chain, the longer it takes to set the desired pattern so for some applications long shift register chains aren't practical.

Input Shift Registers

The compliment device to the 595 is the 597 which is a PISO device; Parallel In, Serial Out and classed as an input shift register. Having input latches, these devices are perfect for capturing data and then serially transferring it to the MPU and once again, incredibly long chains can be constructed.

The datasheets for the two devices can be found on the Texas Instruments website as well as others.

http://www.ti.com/lit/ds/symlink/cd74hc595.pdf
http://www.ti.com/lit/ds/symlink/cd74hc597.pdf

Practical Application

Whilst this has been a very brief introduction and overview of shift register basics, they do have many practical uses in electronics and one of these is expanding the I/O capabilities of PICs and AVRs etc, as they allow for a handful of I/O lines can be expanded to control a great many devices.

So, as a practical application I've given details of a programmable seven-segment display controller based on an 18F25K22 PIC, and programmed with the free AMICUS BASIC compiler. The circuit given (page 4) could be expanded to control up to 255 digits with no additional use of PIC I/O pins.

The problem with long shift register chains is the length of time it takes to send all the data along the chain and this can induce an amount of flicker if the chain isn't clocked fast enough. Fortunately, the PIC is plenty capable of driving the chain at sufficient speed so there is no perceptible flicker, however, as the chain gets longer, the PIC needs to dedicate more CPU time to driving the chain, leaving fewer clock cycles available for other activities.

PIC Firmware operation

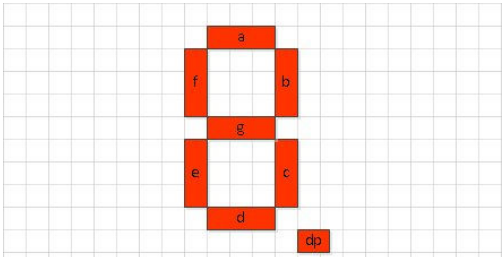
To keep things simple, the PIC firmware listens on PIN PORTC.7 for a 9600 Baud, No Parity serial data stream containing data to be displayed.

Firmware Mode

On power-on, the PIC displays "rdy." and waits for a single byte to be received. This byte specifies the operating mode for the firmware and once sent, can only be changed if the PIC is reset. If the mode is valid, the display is changed to "rdy0" or "rdy1" depending on the mode.

Sending a "0" (ASCII 048) sets to firmware to process and display ASCII characters 0 to 9, "A" to "F" and "a" to "f". Any code outside of this range results in a single decimal point (dp) being displayed.

Sending a "1" (ASCII 049) sets the firmware to process and display raw binary values.



	dp	g	f	e	d	c	b	a	
	128	64	32	16	8	4	2	1	
63	0	0	1	1	1	1	1	1	0
6	0	0	0	0	0	1	1	0	1
91	0	1	0	1	1	0	1	1	2
79	0	1	0	0	1	1	1	1	3
102	0	1	0	0	1	1	1	0	4
109	0	1	1	0	1	1	0	1	5
125	0	1	1	1	1	1	0	1	6
7	0	0	0	0	0	1	1	1	7
127	0	1	1	1	1	1	1	1	8
103	0	1	0	1	1	1	1	1	9
119	0	0	1	1	0	1	1	1	A
124	0	1	1	1	1	1	0	0	B
57	0	0	1	1	1	0	0	1	C
94	0	1	0	1	1	1	1	0	D
121	0	1	1	1	1	0	0	1	E
113	0	1	1	1	0	0	0	1	F
128	1	0	0	0	0	0	0	0	?

The picture of the 7-segment digit shows how the binary value sent will be translated by the display.
Sending ASCII code 064 would just illuminate the "g" segment. Sending ASCII code 009 will illuminate the "a" and "d" segments, an so on.

In either mode, after each ASCII code is received by the PIC, the entire contents of the display chain is shifted one digit to the right, and the new code is placed in the most left hand position. Characters "over-flow" off the end of the chain if there are no LED digits to accommodate them and are lost.

The PIC firmware maintains an internal display buffer and automatically handles the display refresh, so the user only needs to concern themselves with sending values to the display and not the actual display refresh.

Firmware Options

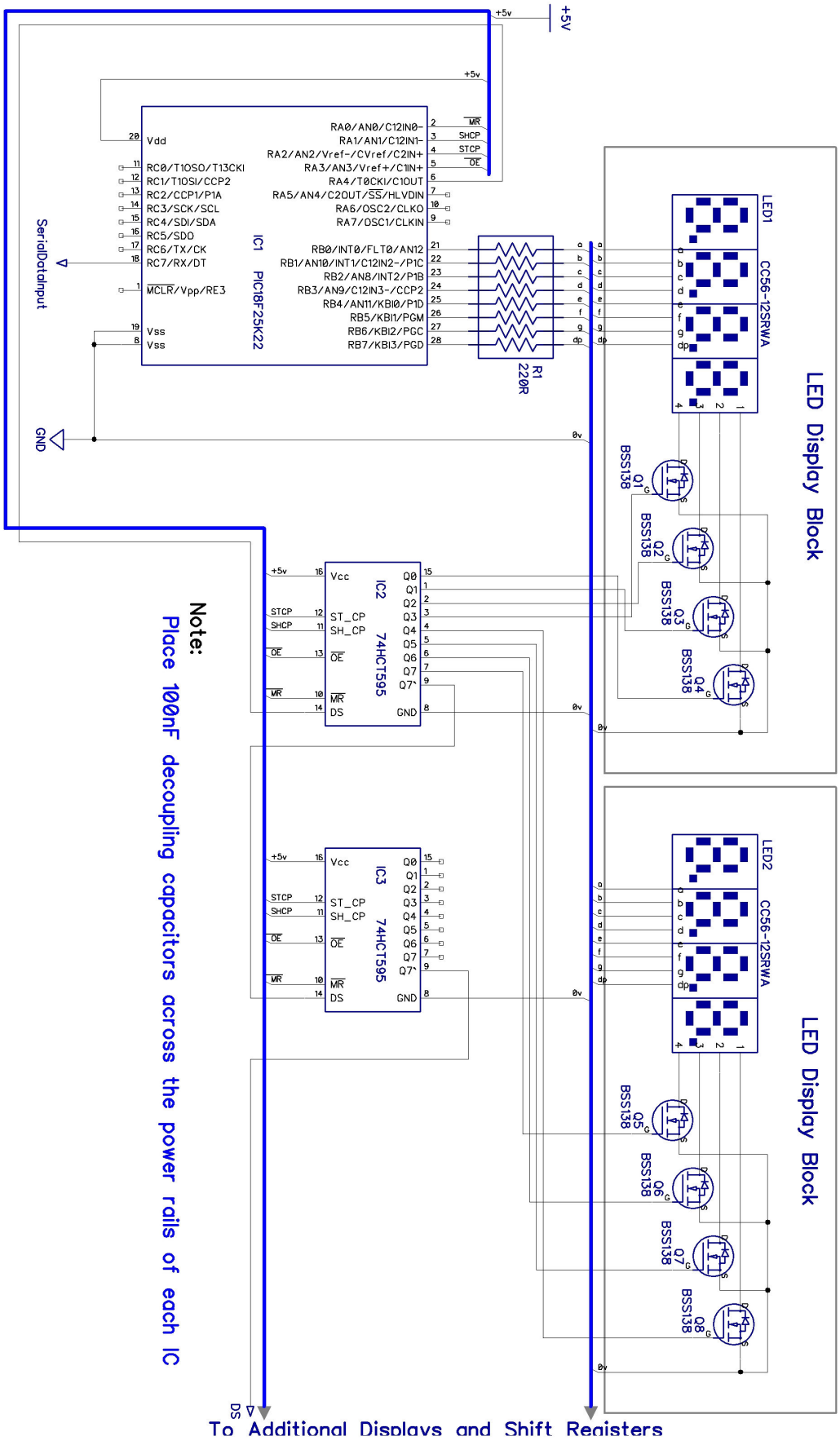
The "bytPatternLookup" array contains the translation look-up table for the display and is located near the start of the program.
If required, additional entries could be defined for this table. So, if for example there was a requirement to be able to display a "minus sign", a new entry, number 18 could be added that held the value "64" - which is the value for segment "g".
It would then be down to the user to assign a "code" for this minus sign, for example "045".
This is done by updating the translation CASE statement; shown below.

```
Select Case bytInput
Case 48 To 57  ' 0 to 9
  bytValue = bytInput - 48

Case 65 To 70  ' A to F (Uppercase)
  bytValue = bytInput - 55

Case 97 To 102  ' a to f (Lowercase)
  bytValue = bytInput - 87

Case 45  ' Minus Sign
  bytevalue = 18  ' Points to 18 in the
translation table.

Case Else
  bytValue = 16  ' Unknown ASCII code

End Select
```

The above code snippet shown in Red, shows the changes required to translate the incoming ASCII code 45, to 18 which then points into the translation table.

Conclusion

The firmware for this project can easily be customised to meet the developers requirements and is just an example of how a display could be controlled.


+5v
+5V
Vdd
IC1
PIC18F25K22
RA0/AN0/C12IN0-
RA1/AN1/C12IN1-
RA2/AN2/Vref-/CVref/C2IN+
RA3/AN3/Vref+/C1IN+
RA4/T0CKI/C1OUT
RA5/AN4/C2OUT/SS/HLVDIN
RA6/OSC2/CLKO
RA7/OSC1/CLKIN
RC0/T1OSO/T13CKI
RC1/T1OSI/CCP2
RC2/CCP1/P1A
RC3/SCK/SCL
RC4/SDI/SDA
RC5/SDO
RC6/TX/CK
RC7/RX/DT
MCLR/Vpp/RE3
Vss
Vss
RB0/INT0/FLT0/AN12
RB1/AN10/INT1/C12IN2-/P1C
RB2/AN8/INT2/P1B
RB3/AN9/C12IN3-/CCP2
RB4/AN11/KBI0/P1D
RB5/KBI1/PGM
RB6/KBI2/PGC
RB7/KBI3/PGD
MR
SHCP
STCP
OE
SerialDataInput
GND
0v
R1
220R
LED1
CC56-12SRWA
LED Display Block
BSS138
Q1
Q2
Q3
Q4
IC2
74HCT595
Vcc
ST_CP
SH_CP
OE
MR
DS
GND
Q0
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q7'
LED2
Q5
Q6
Q7
Q8
IC3
DS
To Additional Displays and Shift Registers
Note:
Place 100nF decoupling capacitors across the power rails of each IC