**SPI and shift-register switches - spiserialparallelSW.[ch]**

**A. General Strategy**

The spi reads from the input shift register while writing to the output shift register. The input shift register loads in parallel from switch inputs, and the output shift register drives FETs that drive LEDs. The switch inputs have pullup resistors and the switch contact closure pulls the input to ground. Therefore, open switches present a one bit. The output shift register drives a FET which pulls a LED via a resistor to gnd, so a one bit in the output turns the LED on.

The spi sends/receives two bytes (currently in the 2/20/2020 repo) then causes an interrupt. The interrupt callback deals with the logic of the switches and restarts another send/receive cycle. With the spi set to the slowest rate the interrupt rate is about 1004 cycles per 100 ms, i.e. approximately 10 cycles per ms. Net, the state of the switches are sampled at about 10K per second.

To deal with this rate, the interrupt callback handling xor’s the input half-word (16b, i.e. two bytes) with the previous half-word, producing a word with bits that are set for switch bits that have changed. If there is no change, no further processing is needed and the next spi cycle is started. If there are changes the switches with changes are updated.

Since a time source is needed for timing the debouncing the spi is used, since the spi interrupt is already present and the rate is relatively constant, the variation only due to the time variations between interrupt and restarting the next cycle. The interrupts are counted down to yield a timing rate of about 100 per sec which is used to time for switch debouncing.

The interrupt handling, therefore, 1) checks for changes in switch bits, handling only those that have changed, and 2) when the countdown counter expires, updates the timing for switches that actively being debounced.

There can be two types of switches. One is the on/off, or pushbutton type, i.e. SPST, and the other is a SPDT which appears to the spi/shift-register as a pair of on/off switches. For the switch pair the combination of the two inputs is used to determine the switch state and debouncing of a single contact is not needed, though at some point timing the indeterminate states as a hardware error check might be added.

**B. Switch structs**

A switch that is to be used is “instantiated” by a FreeRTOS task by calling an initialization routine. The routine adds a struct to the heap and returns a pointer for that switch to the calling task. The arguments to the initialization routine provides the specifications--

(In the following, the text between “ “ is the element name in the struct.)

(“$” designates arguments in the call to instantiate a switch.)

1. Pointer to next instantiated switch struct, i.e. linked list (“pnext”)

NULL = last on the list of instantiated switches

2. Pointer to next switch that is actively being time for debouncing (“pdbnx”)

NULL = last on the list of active debouncing switches

3. $ Task handle (“tskhandle”)

Null = use the handle of calling task.

4. $ Notification word bit (“notebit”)

This specifies the bit used when the task is notified of a switch status change.

Zero = skip notification

5. $ Switch bit position (“switchbit”)

The bit position in the spi read word for this switch.

6. $ Switch bit position1 (“switchbit1”)

The bit position in the spi read word for this switch for the second in a switch pair, If zero, then the switch is not a pair.

7. $ Type (“type”) (Redundant, unless other possibilities are conjured up.)

0 = SPST, on/off (pushbutton)

1 = SPDT, switch pair

8. Switch status not debounced (“on”)

SPST:

0 = switch closed;

1 = open

SPDT:

0 = 00 both closed \*

1 = 01 combination

2 = 10 combination

3 = 11 both open \*

\* debounced state remains unchanged

9. Switch status debounced (“db\_on”)

0 = switch closed;

1 = open

10. $ Mode for debouncing (“mode”)

Here are the current definitions. Only the first three are implemented--

#define SWMODE\_NOW 0 // Immediate, w debounce minimum

#define SWMODE\_WAIT 1 // Wait until debounce ends

#define SWMODE2\_RS 2 // Switch pair R-S flip flop mode

#define SWMODE2\_RS\_00 3 // Switch pair 00 debounced

#define SWMODE2\_RS\_11 4 // Switch pair 11 debounced

#define SWMODE2\_RS\_0011 5 // Switch pair 00 & 11 debounced

11. Debounce working timing counter (“db\_ctr”)

This counter counts down the spi time ticks for debouncing.

12.$ Debounce time when closing: (“db\_dur\_closing”)

This specifies the number of time ticks for a closing event.

13. $ Debounce time when opening: (“db\_dur\_opening”)

This specifies the number of time ticks for an opening event.

Example of instantiation of a pushbutton switch--

// Control Lever Fullscale Normally open. \*/

struct SWITCHPTR\* psw\_cl\_fs\_no = switch\_pb\_add(

NULL, /\* task handle = this task \*/

0, /\* Task notification bit \*/

CL\_FS\_NO, /\* 1st sw see shiftregbits.h \*/

0, /\* 2nd sw (0 = not sw pair) \*/

SWTYPE\_PB, /\* switch on/off or pair \*/

SWMODE\_WAIT, /\* Debounce mode \*/

SWDBMS(20), /\* Debounce ms: closing \*/

SWDBMS(20)); /\* Debounce ms: opening \*/

Note in the above example “SWDBMS” (SwitchDeBounceMilliseconds) is a macro that converts milliseconds into spi time ticks. The spi time tick is presently 10 ms, so values less than 10 convert to one tick. If no debouncing is to be applied, enter zero instead of the macro. Zero time ticks is handled as a special case and bypasses executing code for the debouncing.

**C. Modes**

Converting the spi switch bit samples into switch status is different between the the SPST and SPDT switches.

**C1. SPST**

The switch contacts usually bounce up open and closure, and the duration of bouncing may differ between open and closure so a debounce time is specified for each.

If the switch changes state during active debounce timing, the debounce timer is re-initialized.

There are two modes for debouncing, “SWMODE\_NOW”, and “SWMODE\_WAIT”. Though not a mode, a third mode is “not debounced” which is executed when the debounce time is zero. A switch bit change is not placed on the list for debounce timing and the output (debounced) state follows the switch open/close. Since debounce times for open and close are specified separately this “zero debounce” can be one or both directions.

The “now” mode the first closure changes the debounced output state and starts the debounce timing. If the switch opens and closes during the debounce timing the output state does not change. Therefore, a very short press of a pushbutton gives one output pulse of duration of the debounce timer. If the press is long the output shows closure immediately and continuously as long as the button remains closed. The logic works the same with the reverse sense of open of close, but the debounce time duration will depend on the duration specified in the instantiation call arguments.

With the “wait” mode output state does not change until the debounce period has expired. A press of a pushbutton, or multiple presses, during the debounce timing do not produce any change in the output state. At the end of the debounce time the output changes to the state of the switch. Since the debounce time is extended each time the switch changes state, rapid pressing of the pushbutton with “wait” mode will indefinitely postpone output as long as the switch transitions take place before the debounce period expires.

**C2. SPDT**

**SWMODE2\_RS**

The two contacts, i.e. two switches in the spi/shift-register context, determine the change of switch state. In the static state the switch will have one contact closed and the other open. In the transition between states the both may be briefly closed, or both, depending whether the switch is BBM (Break Before Make, Form C), or MBB (Make Before Break, Form D). The BBM case both contacts go open as the switch changes state. In the MBM case both contacts are closed as the switch changes state. Since the contacts might bounce during the transition have the output state only change when there is a open|close or close|open situation and remain the state when there is open|open or close|close, the contact bounce.

If a transition state remains for a long duration that is likely to indicate a hardware failure. For example, both switches open for a long period would indicate that one or both of the switches are not connected. Both closed for a long period would indicate something missing with the pullup resistors.

Adding additional modes to the code to handle these types of situations can be added later.

**D. Linked lists & bit mapping**

**D1. Instantiated switch list**--

A “head” pointer points to the first switch struct. (NULL is when the list is empty upon bootup.) The number of switch structs possible can be none (no deboucing, task notification), to as many as there are bit positions, i.e. currently 16 SPST switches possible, or one-half the number of bit positions if all the switches are SPDT.

Individual switch status can be accessed at any time via the pointer (“p→on”) when the switch was instantiated. The word with the spi read word (“spisp\_rd[0].u16”) can be read at any time, but it only has the bit status for each shift-register input and SPDT switches will show up as two separate bits, whereas “p->on” will show a number 0-3 for the four possible combinations of the SPDT switch.

**D2. Debouncing list**--

The “head” to this list (“struct SWITCHPTR\* phddb”) is NULL when there are no switches actively debouncing. When a switch bit changes, that switch is added to the the debouncing linked list (provided the specified debouncing time is not zero). Each spi time tick counter, this linked list, if not empty, is traversed, and the debouncing working counters updated. When a counter reaches zero the logic is applied that updates the debounced switch status (“db\_on”), and notifies the task, if “notebit” is not zero. A task can access “db\_on” at any time via the pointer (“p->db\_on”)when the switch was instantiated.

**D3. Bit versus struct mapping**--

An array of 16 pointers holds pointers to the switch struct so that the position in the array corresponds to the bit position of the switch, as read by the spi.

struct SWITCHPTR\* pchange[NUMSWS] = {0};

The spi interrupt xor’s the newly read word with the previous, thus setting bits for switches that have changed. In a single machine cycle the CLZ (count leading zeros) assembly instruction converts the bit position in the xor’d word to the number which is then used an index into the mapping array, thus providing a fast path to the struct for dealing with the change in the switch contact status.

**LAST. Items--**

Miscellaneous thoughts about the topic.

**Array versus heap as needed**

Two ways of implementing the structs for each switch is to, 1) define the maximum possible number of structs in common memory, or 2) add structs as needed (via calloc) on the heap. If the structs are placed on the heap, they are not necessarily in consecutive memory locations. To access the structs, given the switch bit position, without a search, an array of pointers that maps bit position to struct location is needed.

However, with all possible structs pre-defined, the mapping array is not needed, saving 4 bytes per switch position, i.e. with 16 switches, 64 bytes. Another 4 bytes are saved in the struct since the list linking the structs is not needed. Net, 128 bytes. Each struct is reduced to 20 bytes. This is weighted against heap method by the number switches actually instantiated, and the number of SPST versus SPDT type switches as the SPDT switch only needs one struct on the heap.

**Task versus interrupt handling**

The current routine handles the switches during the interrupt. The interrupt priority is set to low priority. The processing is normally very short as most of the time there are no switches changing state and no switches requiring debounce timing. If the application has a large number of switches that change at the same time then some attention to the worst-case interrupt duration and how it might affect RTOS tasks is needed.

If the switch handling is done with a task, then each time the spi has a time tick or switch contact change it has to place the xor’d word on a queue for a task to do the handling. That assures short interrupt handling, but if some of the switches are critical, the switch handling task could be blocked by higher priority tasks. Furthermore, all the spi time ticks and changes result in task context switches.

**Extending beyond 16b**

Additional shift-registers can be added. This changes the word size in the routines. Beyond 32, the routine has to handle two words when doing xor’s of new versus previous readings since the CLZ instruction works on register, i.e. 32b.