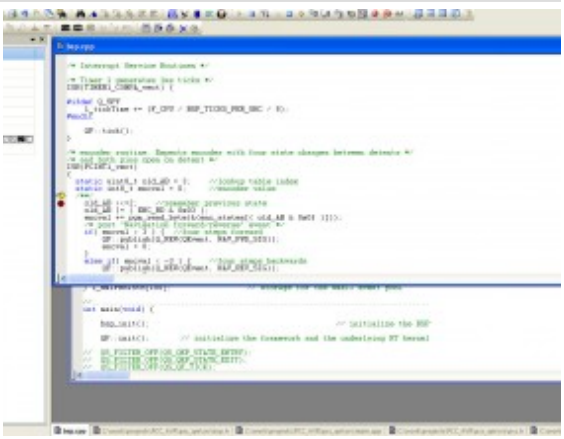
« Using HID device for RC control

Arduino USB Host 2.0 library progress report

Rotary encoder interrupt service routine for AVR micros

By Oleg Mazurov



Encoder ISR

About a year ago, I posted an article about 2-channel rotary encoder interfacing with MCU using lookup table. With its linear program flow, inherent noise immunity and small processing requirements, this method produces extremely efficient code. The article attracted quite a bit of attention with many interesting comments and several code modifications. Today, I want to demonstrate slightly different way of reading the encoder – via interrupt service routine (ISR).

I'm currently involved in a project built with QP Event-Driven Real Time Framework running on 8-bit AVR micro. Rather than implementing periodic polling, I wanted to generate encoder events as they occur. AVR Atmega, as well as many other micros have external interrupt source called "Pin Change", ideally suited for an encoder. As the name implies, pin change interrupt occurs when certain digital port pin changes state from low to high or vice versa. When an encoder is rotated, one pin changes state from say low to high, then other goes from low to high, than the first one goes from high to low, then the other one goes from high to low. The cycle then repeats. In order to properly react to every state change we want to be able to generate interrupt on every transition on each pin and that is exactly how pin change interrupt works.

I am developing for custom built AVR-based board and all code snippets are pasted directly from project files. This code can be used on bare metal Atmega as well as Arduino board with no changes and can also be adopted for other micros with only minor modifications, such as port and pin names. Encoder's A and B pins are connected to Port B pins 0 and 1, common pin is connected to ground. To prevent floating inputs, internal pull-ups are turned on.

The code consists of two main parts – initialization and interrupt service routine. The following code snippet shows the firs part:

```
1
2   ...
3
4   /* Atmega644p MCU */
5   /* encoder port */
6   #define ENC_CTL DDRB    //encoder port control
7   #define ENC_WR  PORTB   //encoder port write
8   #define ENC_RD  PINB    //encoder port read
9   #define ENC_A 0
10  #define ENC_B 1
11
12  ...
13
14    ENC_WR |= (( 1<<ENC_A )|( 1<<ENC_B ));     //turn on pullups
15    PCMSK1 |= (( 1<<PCINT8 )|( 1<<PCINT9 ));  //enable encoder pins interrupt source
16
17  ...
18
19    /* enable pin change interupts */
20    PCICR |= ( 1<<PCIE1 );
21
22
```

The snippet comes from 3 different parts of the code. On **lines 5-9** I'm giving meaningful names to MCU peripherals. **Line 13** turns on internal pull-up resistors on MCU pins connected to encoder pins A and B. **Line 14** sets pin change interrupt sources to those pins. Finally, **line 19** enables pin change interrupt. From this moment, the ISR (shown below) will be called on each pin change.

The interrupt service routine looks very much like the one explained in the article linked to at the beginning of this post. A lookup table which represents eight valid transitions of the encoder is used to increment or decrement a variable holding encoder count. An array is filled with values of -1, 1, or zero (for invalid transitions) and `old_AB` variable which holds present and previous encoder states is used as an index for the array. Let's take a look at ISR code:

```
1
2
3   /* encoder routine. Expects encoder with four state changes between detents *
4   /* and both pins open on detent */
5   ISR(PCINT1_vect)
6   {
7     static uint8_t old_AB = 3;  //lookup table index
8     static int8_t encval = 0;   //encoder value
9     static const int8_t enc_states [] PROGMEM =
10    {0,-1,1,0,1,0,0,-1,-1,0,0,1,0,1,-1,0};  //encoder lookup table
11    /**/
12    old_AB <<=2;  //remember previous state
13    old_AB |= ( ENC_RD & 0x03 );
14    encval += pgm_read_byte(&(enc_states[( old_AB & 0x0f )]));
15    /* post "Navigation forward/reverse" event */
16    if( encval > 3 ) {  //four steps forward
17      QF::publish(Q_NEW(QEvent, NAV_FWD_SIG));
18      encval = 0;
19    }
20    else if( encval < -3 ) {  //four steps backwards
21      QF::publish(Q_NEW(QEvent, NAV_REV_SIG));
22      encval = 0;
    }
}
```

Line 3 is standard avr-gcc pin change 1 interrupt vector macro which places the subsequent code at certain memory address specified as ISR address. **Line 5** declares old_AB variable which holds encoder history – note that I initialize it to 3 instead of zero because I'm using Bourns PEC-12 encoder which has "both pins open" state at detent (encoder sold by Sparkfun has "both pins connected to common" state at detent). **Line 6** declares a variable which holds encoder value, it increments when encoder is rotated counterclockwise and decrements when encoder is rotated clockwise. The `encval` can be placed outside the ISR if access to encoder value is desired. **Line 7** declares lookup table. Note that in order to save 16 bytes of precious RAM the array is placed in program memory.

Next 3 lines perform the actual work. On **line 10**, the previous state of encoder pins is shifted left. After that, the current encoder state is read and placed where the old one was in **line 11**. Port read is performed atomically in order to prevent a situation when encoder gets turned between individual pin reads.

After this is done, a combination of previous and current states forms an index to a lookup table, which is read in **line 12** to find out whether encoder was moved forward (1), backward(-1) or was just bouncing (0). The value from lookup table then gets added to `encval`.

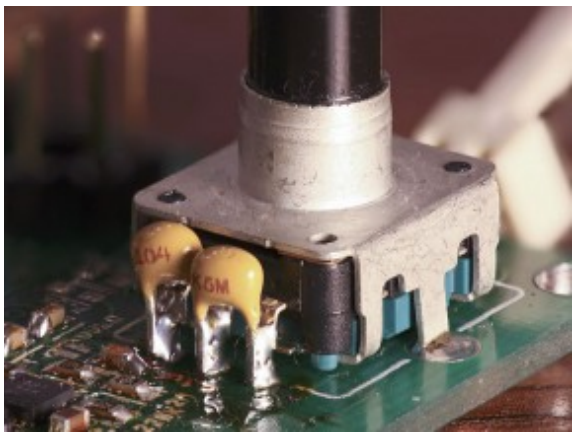
Next lines are part of my QP-based project. I wanted to generate "encoder forward" or "encoder reverse" events when encoder travels from one detent to the next. There are four state changes between detents; all I do is check if 4 steps has been made, generate event using `QF::publish()` function, and then reset `encval` to start counting from zero. This piece of code is not important for understanding the lookup table technique; I included it here because output of this part will be used to explain algorithm's "natural debouncing" properties.

Bouncing is intermittent contact closure due to mechanical properties of a contact – it's actually a spring which tends to jump a little after landing on the mating surface. As a result, the ISR will fire reacting to a pin change. In this situation, it would be reasonable to expect false readings, however, they won't occur due to two reasons. First, as can be seen from a lookup table, not every combination of previous plus current state is valid: ones that contain "0" won't change `encval`. Second, even if a valid combination has been produced during bouncing, it it impossible to "make" two steps in the same direction. Only one of two pins would change state during single encoder step; the other stays in the same state and therefore won't bounce. Consequently, it is only possible to make one accidental step and only backwards, after which a step forward will ultimately happen when bouncing finally stops and a firm contact gets established.

Using QP framework's QSPY facility, I was able to produce very granular trace of real-time events inside the application. Let's take a look at this listing:

```
1
2
3
4   0394883704 User000:
5   Enc: 21
6   Encval:   1
7   old_AB: BD
8   0394982674 User000:
9   Enc: 20
10  Encval:   2
    old_AB: F4
```

```
0394983175 User000:
Enc: 20
Encval:   2
old_AB: D0
0394983682 User000:
Enc: 20
Encval:   2
old_AB: 40
0394984188 User000:
Enc: 20
Encval:   2
old_AB: 00
0395726873 User000:
Enc: 20
Encval:   2
old_AB: 00
0395727374 User000:
Enc: 20
Encval:   2
old_AB: 00
0395727875 User000:
Enc: 20
Encval:   2
old_AB: 00
0395728376 User000:
Enc: 22
Encval:   2
old_AB: 00
0395727377 User000:
Enc: 22
Encval:   2
old_AB: 00
0395727878 User000:
Enc: 22
Encval:   3
old_AB: 02
0395784009 NEW    : Evt(Sig=00000004,Obj=00000000, size=    2)
0395784218 User000:
Enc: 23
Encval:   0
old_AB: 0B
0395784785 Disp==>: Obj=000006B4 Sig=00000004,Obj=000006B4 Active=00003EC5
0395784960 Intern : Obj=000006B4 Sig=00000004,Obj=000006B4 Source=00003EC5
```



Hardware debouncer

The beginning of interrupt record looks like **line 1**; the 'User000' pattern is an indicator. Actual output gets generated at the end of interrupt routine after possible `QF::publish()` call and subsequen zeroing of `encval`. The number to the left is a timestamp, the resolution is 8 system clock cycles. In my case, one system clock cycle is 83ns, therefore, to find time between two events in microseconds, subtract one timestamp from another and multiply the difference by 0.664, discarding negative sign, if present.

Next line shows encoder port and we are only interested in two lowe bits of it since this is where our encoder is connected. Next line shows `encval`, and the next one shows `old_AB`. We are only interested in lower nibble of old_AB.

Line 45 shows new event generated as a result of `QF::publish()` function call. Last two lines are irrelevant. Interrupts where `encval` has not been changed from its previous value is bounce. You can see that often times a glitch is so short that by the time it's read the port already contains the previous value. The code can't be fooled by that, however, and only advances `encval` when valid pin state combination has been read. Studying the listing, you'll see many interesting situations – sometimes I forced the encoder to stop between detents to produce a lot of bounce.

Finally, I want to share a trick that I use from time to time. As you can see from the listing, "debouncing" takes quite a bit o processing and in some situations we can't afford to waste precious CPU cycles. Or maybe we want to reduce time spent in a critical section (with interrupts locked) to a minimum. In this case, a very good encoder debouncing can be achieved with a pair of 0.1uF ceramic capacitors which can be soldered directly to encoder pins. A very accurate explanation on

how this debouncer works has been given by “Grumpy_Mike” Cook in this Arduino forum thread. In my experience, cheap low-quality encoders, even equipped with capacitors would still bounce occasionally but the amount of extra processing will be greatly reduced. On the other hand, cheap high-quality encoders, like PEC-12 featured in the picture, would stop bouncing completely.

I used the code presented in this article in several projects – it has small processing footprint and works reliably. Study it, try it, and if you have any questions or want to add your opinion, please leave a comment!

Oleg.

No related posts.

March 30th, 2011 | Tags: AVR, interrupt, ISR, rotary encoder | Category: MCU, programming | 36 comments

36 comments to Rotary encoder interrupt service routine for AVR micros

Markus Gritsch
April 8, 2011 at 1:55 am · Reply

Thank you for sharing your code. I especially like your elegant approach using the enc_states array. One little quirk which was easily solved: The setup section is a bit inconsistent, since it uses PORTB, but then uses PCMSK1, PCINT8, PCINT9, PCIE1, and PCINT1_vect which all require using PORTC.

After changing the above SFRs to their PORTB relatives, it worked fine. However, I discovered one issue: In case one or more transitions get lost (Either because the 100 nF caps make it too slow or the ISR simply isn’t fast enough), the algorithem fails. Suppose encval is equal to 2 (due to lost transistions) in the detented position. Now turning one detent down yields -2 (no QF::publish takes place), turning again one detent up yields 2 (again, no QF::publish takes place).

I find this 4 steps between detents quite annoying, causing this whole trouble. Do you have any idea how to solve this?

oleg
April 8, 2011 at 9:45 am · Reply

Thanks for catching on setup inconsistency – I forgot to mention that I was using Atmega644p so port/pin/interrupt register names are valid for this MCU. I added a comment to the code snippet stating this.

You can deal with skipping steps in several ways. One is demonstrated in the article: you generate one step per detent. If you skip a step, just keep rotating the encoder in the same direction (that’s what people would do in this situation, anyway). As soon as 4 pulses are generated, non necessarily on detent, the algorithm will fix itself.

There exist plenty of encoders without detent – they will work just fine for things like volume control, where accurate positioning is less relevant. For menu navigation I still prefer a detented one.

Markus Gritsch
April 8, 2011 at 11:50 am · Reply

I do not agree, that the algorithm will fix itself. Once the encval is out of sync with the detents, it will stay out of sync with the current implementation. Please reread the second paragraph of my previous comment.

Let’s again assume envcal is 2 at a detent posision due to lost changes. Turning one detent down results in an encval of -2 (no QF::publish). If, like you suggested we turn another detent down, encval will reach -4 during halfway between the detents, QF::publish gets fired, encval gets set to 0, and at the detent position it will be -2. If we now turn one detent up, encval will be 2 and no QF::publish gets fired.

In this out-of-sync situation, when changing the turn direction, one will always have to turn two detents in the new direction to get a QF::publish fired.

I think I found a simple solution to bring encval in sync with the detents:

```
#include
#include
#include

/* encoder port */
#define ENC_CTL DDRB //encoder port control
#define ENC_WR PORTB //encoder port write
#define ENC_RD PINB //encoder port read
```

```
#define ENC_A 0
#define ENC_B 1

void setup()
{
ENC_CTL &= ~(( 1<<ENC_A )|( 1<<ENC_B )); //inputs
ENC_WR |= (( 1<<ENC_A )|( 1<<ENC_B )); //turn on pullups
PCMSK0 |= (( 1<<PCINT0 )|( 1<<PCINT1 )); //enable encoder pins interrupt sources

/* enable pin change interupts */
PCICR |= ( 1<<PCIE0 );

Serial.begin(115200);
}

void loop()
{
}

/* encoder routine. Expects encoder with four state changes between detents */
/* and both pins open on detent */
ISR(PCINT0_vect)
{
static uint8_t old_AB = 3; //lookup table index
static int8_t encval = 0; //encoder value
static const int8_t enc_states [] PROGMEM =
{0,-1,1,0,1,0,0,-1,-1,0,0,1,0,1,-1,0}; //encoder lookup table
/**/
uint8_t AB = ENC_RD & 0×03;
//Serial.print("AB="); Serial.println(AB, DEC);
old_AB < 3 ) {
Serial.println(“inc”);
encval = 0;
} else if ( encval < -3 ) {
Serial.println("dec");
encval = 0;
} else if ( AB == 3 ) {
//bring the encval in sync with the detents in case of lost transitions
encval = 0;
}
}
```

Markus Gritsch
April 8, 2011 at 11:52 am · Reply

I do not agree, that the algorithm will fix itself. Once the encval is out of sync with the detents, it will stay out of sync with the current implementation. Please reread the second paragraph of my previous comment.

Let’s again assume envcal is 2 at a detent posision due to lost changes. Turning one detent down results in an encval of -2 (no QF::publish). If, like you suggested we turn another detent down, encval will reach -4 during halfway between the detents, QF::publish gets fired, encval gets set to 0, and at the detent position it will be -2. If we now turn one detent up, encval will be 2 and no QF::publish gets fired.

In this out-of-sync situation, when changing the turn direction, one will always have to turn two detents in the new direction to get a QF::publish fired.

I think I found a simple solution to bring encval in sync with the detents:

```
#include
#include
#include

/* encoder port */
#define ENC_CTL DDRB //encoder port control
#define ENC_WR PORTB //encoder port write
#define ENC_RD PINB //encoder port read
#define ENC_A 0
#define ENC_B 1

void setup()
{
ENC_CTL &= ~(( 1<<ENC_A )|( 1<<ENC_B )); //inputs
```

```
ENC_WR |= (( 1<<ENC_A )|( 1<<ENC_B )); //turn on pullups
PCMSK0 |= (( 1<<PCINT0 )|( 1<<PCINT1 )); //enable encoder pins interrupt sources

/* enable pin change interupts */
PCICR |= ( 1<<PCIE0 );

Serial.begin(115200);
}

void loop()
{
}

/* encoder routine. Expects encoder with four state changes between detents */
/* and both pins open on detent */
ISR(PCINT0_vect)
{
static uint8_t old_AB = 3; //lookup table index
static int8_t encval = 0; //encoder value
static const int8_t enc_states [] PROGMEM =
{0,-1,1,0,1,0,0,-1,-1,0,0,1,0,1,-1,0}; //encoder lookup table
/**/
uint8_t AB = ENC_RD & 0x03;
//Serial.print("AB="); Serial.println(AB, DEC);
old_AB < 3 ) {
Serial.println("inc");
encval = 0;
} else if ( encval < -3 ) {
Serial.println("dec");
encval = 0;
} else if ( AB == 3 ) {
//bring the encval in sync with the detents in case of a lost transitions
encval = 0;
}
}
```

oleg
April 8, 2011 at 5:11 pm · Reply

This could be valid. However, I haven't seen it happen myself and I need to conduct more tests before giving an explanation. The dump from ISR -> http://www.circuitsathome.com/wp/wp-content/uploads/2011/03/bounce.txt shows that "NEW" event (the result of QF::publish()) always happens at detent, i.e. when "Enc" AKA PORTB equals x3 so apparently the code never misses a pulse, even though bounces and "back-steps" are aplenty, as can be seen from the same dump.

oleg
April 8, 2011 at 5:24 pm · Reply

Your idea is actually pretty cool. For this encoder, we can be certain that when PORTB & 0×03 equals 0×03 and transition is valid, i.e., enc_states[(old_AB & 0x0f)] is non-zero, then encoder is at detent. The ISR then becomes much shorter because we don't need to count steps and make these stupid checks anymore. I'll check this idea and come back

Marcin
August 21, 2012 at 2:37 am · Reply

I like your solution. But I would change the encval check this way:

```
if (AB == 3) {
if (old_AB > 3 ) {
Serial.println("inc");
encval = 0;
} else if ( encval < -3 ) {
Serial.println("dec");
encval = 0;
}
}
```

That way the encval is checked only on detent (so less CPU cycles is used on encval check) and it syncs itself automatically.

Marcin
August 21, 2012 at 2:40 am · Reply

Sorry, there is an error in my previous comment (I was checking if old_AB is greater than 3 which is wrong)

Corrected code:

```
if (AB == 3) {
  if (encval > 3 ) {
    Serial.println("inc");
    encval = 0;
  } else if ( encval < -3 ) {
    Serial.println("dec");
    encval = 0;
  }
}
```

Markus Gritsch
April 9, 2011 at 3:33 am · Reply

When using 100 nF debounce caps (to prevent from excessive ISR calls), it was quite easy to miss a transition due to the too large time constant of the RC combination. Without the decounce caps, a transition is much harder to miss.

I do not yet see, how the ISR can be made much shorter. IMO counting steps is always necessary to determine the direction of the rotation. Looking forward to your investigation

oleg
April 9, 2011 at 11:21 am · Reply

We don't need to count steps – lookup table entries contain direction information. Here is new ISR:

```
ISR(PCINT1_vect)
{
static uint8_t old_AB = 3; //lookup table index and initial state
uint8_t encport; //encoder port copy
uint8_t direction;
static const int8_t enc_states [] PROGMEM =
{0,-1,1,0,1,0,0,-1,-1,0,0,1,0,1,-1,0}; //encoder lookup table
/**/
old_AB <<=2; //remember previous state
encport = ENC_RD & 0x03; //read encoder
old_AB |= encport; //create index
direction = pgm_read_byte(&(enc_states[( old_AB & 0x0f )])); //get direction
if( direction && ( encport == 3 )) { //check if at detent and transition is valid
/* post "Navigation forward/reverse" event */
if( direction == 1 ) {
QF::publish(Q_NEW(QEvent, NAV_FWD_SIG));
}
else {
QF::publish(Q_NEW(QEvent, NAV_REV_SIG));
}
}//if( direction && encport == 3...
}
}
```

The compiled size is 6 bytes less than the ISR from the article. It is insensitive to step loss, unless a step to detent is lost, which is unlikely since human-operated encoder tends to stay longer on detent. Even in this case direction information is extracted correctly – if you rotate encoder one detent clockwise, say, lose this pulse and then rotate encoder one detent counterclockwise, you'll get correct direction.

Markus Gritsch
April 9, 2011 at 1:57 pm · Reply

I will keep my modified version of your original code. This new ISR has the drawback of sending way too much events in case of contact bouncing.

Teis
January 10, 2012 at 4:07 pm · Reply

Hi Markus, I'd like to compare the different versions of the code, but your code is incomplete. Will you post the full code?

RT
April 19, 2011 at 9:30 am · Reply

I studied a rotary encoder that, I think, is very similar to yours. A 2-bit grey code rotary encoder doesn't require counting. I have tried different methods, with and without debouncing, and concluded that for a manually operated rotary encoder, a timer interrupt and/or a little debouncing will work better than an external interrupt. Also, I never count steps. I wait for something to happen on pin 1 and then check the state of pin 2. The whole interrupt routine only requires 12 useconds and can be optimized to 8 useconds.
The only drawback is the possibility to overflow the decoding, missing a state change here and there. But again, this is fore manually operated rotary encoder.
Take a look at http://practicalusage.com/?p=246 and http://practicalusage.com/?p=267

Rotary Encoder: H/W, S/W or No Debounce? « H i F i D U I N O
May 18, 2011 at 12:20 pm · Reply

[...] And here is the right way to do it (from circuits@home) [...]

Michael Schwager
July 19, 2011 at 4:58 pm · Reply

At the beginning of this article, you mention how your previous article had "inherent noise immunity". I don't understand how you can say that. I did some testing, and determined that the "noise immunity" was because the Serial.print() statements took about 1ms to complete at 115200 bps (115200 bps == 14400 char ps [8-bit], 17-19 chars to print the counter value). As a matter of fact, I was nervous about it because 1.) I would not have any Serial.print() debugging code in my production code and 2.) a simple interrupt showed that, by removing the Serial.print() statements, the counters would most certainly be inaccurate. See this code:

```
/* Rotary encoder read example */
#define ENC_A 2
#define ENC_B 3
#define ENC_PORT PIND
#define RSHIFT 2 // Put the ports at the LSB

void setup()
{
/* Setup encoder pins as inputs */
pinMode(ENC_A, INPUT);
digitalWrite(ENC_A, HIGH);
pinMode(ENC_B, INPUT);
digitalWrite(ENC_B, HIGH);
attachInterrupt(0, doInterrupt, FALLING);
Serial.begin (115200);
Serial.println("Start");
}

static volatile int8_t pulses=0;

void loop()
{
static uint8_t counter = 0; //this variable will be changed by encoder input
int8_t tmpdata;
/**/
tmpdata = read_encoder();
if( tmpdata ) {
Serial.print("Counter value: ");
```

```
Serial.print(counter, DEC); Serial.print(" -");
Serial.println(pulses, DEC);
counter += tmpdata;
pulses=0;
}
}

/* returns change in encoder state (-1,0,1) */
int8_t read_encoder()
{
static int8_t enc_states[] = {0,-1,1,0,1,0,0,-1,-1,0,0,1,0,1,-1,0};
static uint8_t old_AB = 0;
/**/
old_AB <>RSHIFT & 0x03 ); //add current state
return ( enc_states[( old_AB & 0x0f )]);
}

void doInterrupt()
{
pulses++;
}
```

oleg
July 19, 2011 at 5:35 pm · Reply

The trace in this article was produced using QSPY – it doesn't introduce any delay. You can see how it works with bouncing encoder.

Michael Schwager
July 19, 2011 at 9:31 pm · Reply

…Got it, thanks! You have created a very clever technique; I've spent all evening studying your two articles and playing on my Arduino. Thanks for all your hard work!

Carlos Ladeira
October 13, 2011 at 4:56 pm · Reply

Hi Oleg,

I'm writing to say thank you for this ingenious code and to report some problem with the second ISR version.

I've been testing your code with and encoder I order from Farnell (http://pt.farnell.com/jsp/search/productdetail.jsp?SKU=1653380) using an arduino pro mini from sparkfun.

The original version works like a charm, however, the new ISR version doesn't work continually and make several jumps in the counting.

Here my code:

```
/*
Rotary encoder read example
using interrupts and an Arduino pro mini from sparkfun
*/

#include

#define ENC_CTL DDRC //encoder port control
#define ENC_WR PORTC //encoder port write

#define ENC_RD PINC //encoder port read
#define ENC_A 0
#define ENC_B 1

volatile uint16_t counter=0;

void setup()
{
ENC_WR |= (( 1<<ENC_A )|( 1<<ENC_B )); //turn on pullups
PCMSK1 |= (( 1<<PCINT8 )|( 1<<PCINT9 )); //enable encoder pins interrupt sources
PCICR |= ( 1<<PCIE1 ); //enable pin change interupts
```

```
Serial.begin (115200);
Serial.println("Start");
}

void loop()
{
static uint16_t tmpdata = 0;

if( counter != tmpdata ) {
Serial.print("Counter value: ");
Serial.println(counter, DEC);

tmpdata = counter;
}
}

ISR(PCINT1_vect)
{
static uint8_t old_AB = 3; //lookup table index
static int8_t encval = 0; //encoder value
static const int8_t enc_states [] PROGMEM = {0,-1,1,0,1,0,0,-1,-1,0,0,1,0,1,-1,0};
//encoder lookup table

old_AB < 3 ) { //four steps forward
counter++;
encval = 0;
}
else if( encval < -3 ) { //four steps backwards
counter--;
encval = 0;
}
}

/*
*****************************
WITH THIS ISR DON'T WORK FOR ME
*****************************
ISR(PCINT1_vect)
{

static uint8_t old_AB = 3; //lookup table index and initial state
uint8_t encport; //encoder port copy
uint8_t direction;
static const int8_t enc_states [] PROGMEM = {0,-1,1,0,1,0,0,-1,-1,0,0,1,0,1,-1,0};
//encoder lookup table
old_AB <<=2; //remember previous state
encport = ENC_RD & 0x03; //read encoder
old_AB |= encport; //create index
direction = pgm_read_byte(&(enc_states[( old_AB & 0x0f )])); //get direction
if( direction && ( encport == 3 )) { //check if at detent and transition is valid
// post "Navigation forward/reverse" event

if( direction == 1 ) {
counter++;
}
else {
counter--;
}
}//if( direction && encport == 3...
}
*/
```

oleg
October 13, 2011 at 5:30 pm · Reply

Are you using debouncing caps on encoder pins? Some encoders give nasty bounces and you need to tame them a little.

Carlos Ladeira
October 14, 2011 at 4:21 pm · Reply

Adding 100nF caps at the encoder pins both ISR work without problems in my case. Thanks.

scrubscrout
March 22, 2012 at 3:16 pm · Reply

Great code!
I'm using an encoder with 564 counts/revolution, but using a uint8_t only gives up to 255 counts. Going to a uint16_t for the encval gives me enough counts, but when reversing it jumps several hundred or even thousand counts. Is that a debouncing issue if it can read correctly when going one direction but not when rotating back and forth slowly?

sommerfeld
March 25, 2014 at 12:21 pm · Reply

I have the same issue with 400 p/rev encoder. Works well for counting up, backwards it looses control. What could be the reason for this behaviour?

Mike
August 3, 2012 at 4:43 pm · Reply

Dear oleg,
beautiful idea ! I used your code on a Teensy card, I only had to adapt a little bit because PCINTs are shared with PWM and SPI pins which ones I used in my design. So I simply used INT0, INT1 for "A" and "B" signals :
init :
EICRA &= ~(1<<ISC01); // trigger INT0 on rising or falling edge
EICRA |= (1<<ISC00);
EICRA &= ~(1<<ISC11); // trigger INT1 on rising or falling edge
EICRA |= (1<<ISC10);
EIMSK |= (1<<INT0)|(1<0 1->-1 2->1 3 ->0
However I do not understand the next values. enc_states[4] is 1. If I do not make any mistake, 4 = 1000b, this means the previous state was 01 and the actual state is 00. I do not understand why this state indicates we have to go right. Can you explain it ?

Best Regards,
Mike

Mike
August 3, 2012 at 4:46 pm · Reply

Dear oleg,
beautiful idea ! I used your code on a Teensy card, I only had to adapt a little bit because PCINTs are shared with PWM and SPI pins which ones I used in my design. So I simply used INT0, INT1 for "A" and "B" signals :

```
init :
EICRA &= ~(1<<ISC01); // trigger INT0 on rising or falling edge
EICRA |= (1<<ISC00);
EICRA &= ~(1<<ISC11); // trigger INT1 on rising or falling edge
EICRA |= (1<<ISC10);
EIMSK |= (1<<INT0)|(1<<INT1);
sei();

//launch main interrupts
ISR(INT1_vect)
{
rotary_func();
}

ISR(INT0_vect)
{ rotary_func();

}
```

where "rotary_func()" is a function which contains the code that you placed in the interruption. This perfectly works, whereas hardware debouncing did not satisfactory work.

Unfortunately there is something that I do not understand : how did you obtain the values inside the static “enc_states” table ? I do understand very well the 4 first values : in the case we just start, the previous state is “00″ and the actual state can be any of 00, 01, 10, 11. Well, in this case, we can say that the rotary goes left with 01 and right with 10. All that lead to the 4 first values of the table : 0->0 1->-1 2->1 3 ->0
However I do not understand the next values. enc_states[4] is 1. If I do not make any mistake, 4 = 1000b, this means the previous state was 01 and the actual state is 00. I do not understand why this state indicates we have to go right. Can you explain it ?

Best Regards,
Mike

oleg
August 3, 2012 at 5:11 pm · Reply

4 is 0b0100.

Mike
August 3, 2012 at 10:32 pm · Reply

Ok thank you for your answer. Of course 4 is 0100b … I finished to understand that all that’s in the table are all the possible transitions on a rotary encoder’s chronogram.
Best Regards, thank you for the job !
Mike

David
September 22, 2012 at 7:07 am · Reply

Seems to me that using a table is highly inefficient. All you need is the following (here just the interrupt service routine):

```
/* set up INT0 for PD2 and INT1 for PD3 interrupts on rising edge ...*/

void enc_getdir()
{
// put pin values in their own vars to avoid bit manipulations for every calculation
if (PIND & (1<<PIND3)) encA = 1; else encA = 0;
if (PIND & (1<<PIND2)) encB = 1; else encB = 0;

// count only if both pins are equal ie 00 or 11 and only one of the pins has changed
if (encA == encB) {
// a change in the A pin means CW otherwise CCW
if (oldencA ^ encA)
encCCW++; // count CCW steps
else if (oldencB ^ encB)
encCW++; // counts the CW steps
}
oldencA = encA; oldencB = encB;
}

ISR(INT0_vect)
{
CLI();
enc_getdir();
SEI();
}

ISR(INT1_vect)
{
CLI();
enc_getdir();
SEI();
}
```

oleg
September 22, 2012 at 12:38 pm · Reply

Define “inefficient”.

jaz
October 23, 2012 at 3:01 pm · Reply

oleg. thank you very much for sharing all of this
where can i get the complete code ?

HANDYJO
October 24, 2012 at 3:31 pm · Reply

I am designing a directional people sensor and counter in a room using two infrared sensors (S1 and S2) which have a quadrature output such as the rotary encoder.What I want is the first sequence on breaking S1 and S2 the change should signal entry and increment a counter whilst breaking the beam in the second sequence S2 first and S1 will signal exit and decrement a counter.I wish also to be able to timestamp this data and pass it to a remote computer by wifi or bluetooth.
Any ideas on how to code this on an ARDUINO or PIC i will appreciate?

Thanks.

James
December 4, 2012 at 3:05 pm · Reply

Thanks Oleg for this ingenious code.Just a quick question,where and how do you code the function to display the count on a 3 digit 7 segment multiplexed common cathode display instead of serial printing it?

prinwibowo
October 4, 2013 at 1:54 am · Reply

Thanks Oleg for sharing. my question is how to read multiple rotary encoder at least 16 encoders on a microcontroller. I am not sure using shift register chip or matrix scanning will work, probably you can give a little idea for that.

denimjeans
October 13, 2013 at 5:30 am · Reply

well done, now with inline asm then we are good to go

oleg
October 13, 2013 at 9:50 am · Reply

you won't gain much with asm here – I looked at compiler-generated assembly of this code and it's already pretty tight.

chadbone
January 24, 2014 at 2:27 pm · Reply

Thanks so much for taking the time to share this!
I am trying very hard to learn how to use the Sparkfun rotary encoder to work as a menu selector for an Arduino Uno. I am very new to this type of code and am trying to crate a simple sketch that will read the encoder (on an interrupt basis so other code doesn't block it) and then print out the encoder value. Could you please please post a simple version like this… or maybe correct mine? My stabs in the dark don't do anything and I admit I have no idea what I am doing.

I have the encoder A&B wired to the Arduino Uno A0 and A1 while the encoder C is wired to ground. I have some Caps on order but would be happy with even a noisy signal at this point.

Something like this (though that actually worked) would be awesome!

//Rotary encoder read on interupt example

```
#define ENC_CTL DDRB //encoder port control
#define ENC_WR PORTB //encoder port write
#define ENC_RD PINB //encoder port read
#define ENC_A 0
#define ENC_B 1
```

```
volatile uint16_t counter=0;

void setup(){
ENC_WR |= (( 1<<ENC_A )|( 1<<ENC_B )); //turn on pullups
PCMSK1 |= (( 1<<PCINT8 )|( 1<<PCINT9 )); //enable encoder pins interrupt sources
PICR |= ( 1<<PCIE1 ); //enable pin change interupts

Serial.begin (115200);
Serial.println("Start");
}

void loop(){
static uint16_t tmpdata = 0;
if( counter != tmpdata ) {
Serial.print("Counter value: ");
Serial.println(counter, DEC);
tmpdata = counter;
}
}

ISR(PCINT1_vect){
static uint8_t old_AB = 3; //lookup table index and initial state
uint8_t encport; //encoder port copy
uint8_t direction;
static const int8_t enc_states [] PROGMEM =
{0,-1,1,0,1,0,0,-1,-1,0,0,1,0,1,-1,0}; //encoder lookup table
old_AB <<=2; //remember previous state
encport = ENC_RD & 0×03; //read encoder
old_AB |= encport; //create index
direction = pgm_read_byte(&(enc_states[( old_AB & 0x0f )])); //get direction
if( direction && ( encport == 3 )) { //check if at detent and transition is valid
/* post "Navigation forward/reverse" event */
if( direction == 1 ) {
counter++;
}
else {
counter–;
}
}
}
```

THANKS SO MUCH!

Johan
February 20, 2014 at 6:58 am · Reply

Is it possible to run this code on Arduino, to get interrupt handling instead of this?
http://www.circuitsathome.com/mcu/programming/reading-rotary-encoder-on-arduino