ENCODER_MODE = 4 * ENCODER_PULSES / ENCODER_DETENTS

Wie man das bei einer Software Lösung berücksichtigt, zeige ich im entsprechenden Kapitel.

Wie man am Bild 11 gut erkennen kann, unterscheiden sich Encoder auch noch in der mechanischen Anordnung der Rasten zwischen den Schalterstellungen.
Das muss u.U. berücksichtigt werden, da es andernfalls dazu führen kann, dass der Rastpunkt an einer Status-Grenze zum „Flattern“ neigt. D.h. die ausgelesenen Bits nicht stabil sind.
Dies wird hier allerdings nicht weiter behandelt. Näheres findet sich z.B. hier:
http://www.mikrocontroller.net/articles/Drehgeber
Ich habe hier nur die wichtigsten Encoder Parameter genannt, soweit sie für die weiteren Ausführungen wichtig sind.

3 Auswertung mit TCPWM component

3.1 Ohne Debouncing

Projekt 049_RotaryEncoder_HW_not_debounced_1

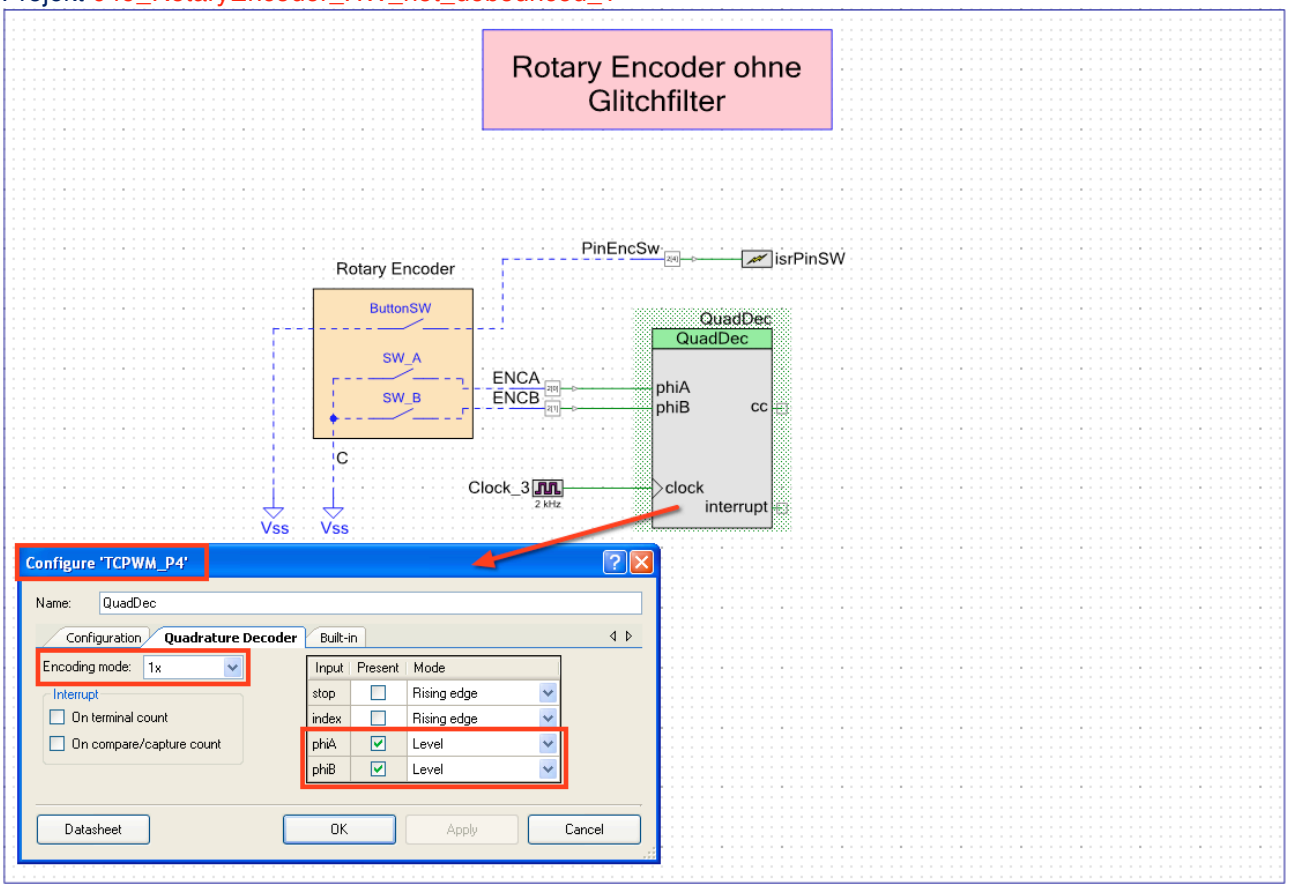


Bild 12: Verdrahtung Rotary Encoder ohne Debouncing und TCPWM compon**ent**

Da die Encoder Eingangs Pins gegen GND geschalten werden, werden sie im Resistive pull up Mode betrieben.

Achtung!
Da ButtonSW ebenfalls gegen GND geschaltet wird, wird beim Einschalten(reset) der zugehörige Interrupt ausgelöst, obwohl der Taster nicht gedrückt wurde.

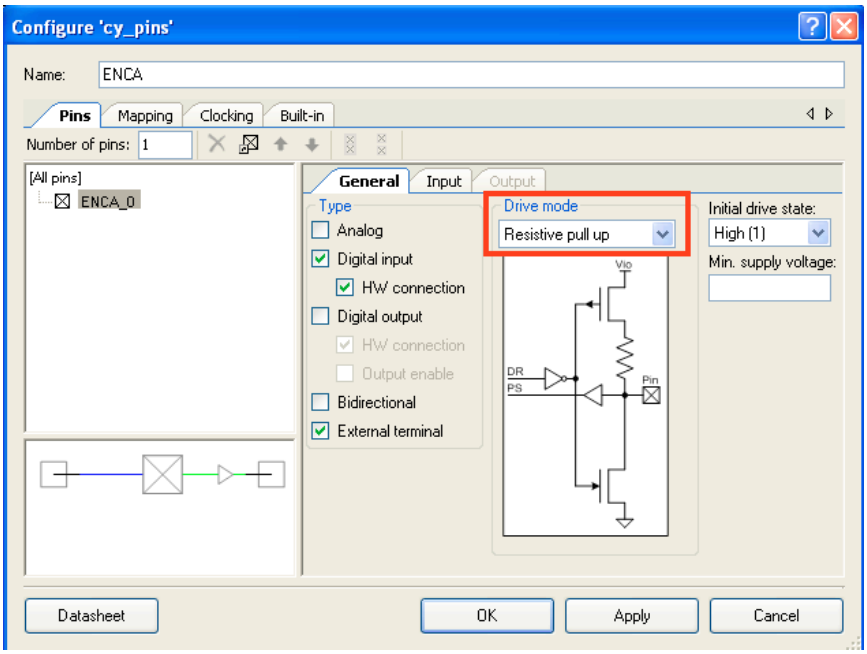
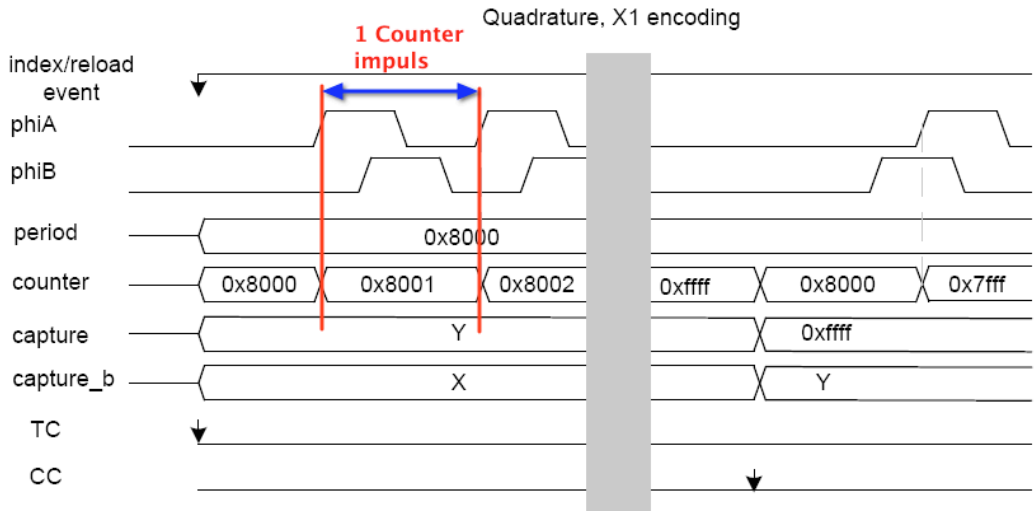


Bild 13: Eingangs PIN Konfiguration

Encoding Mode (siehe auch 2.5)
Die TCPWM Komponente kann einen Encoder-Impuls zu 1,2 oder 4 counter-Impulsen auswerten.



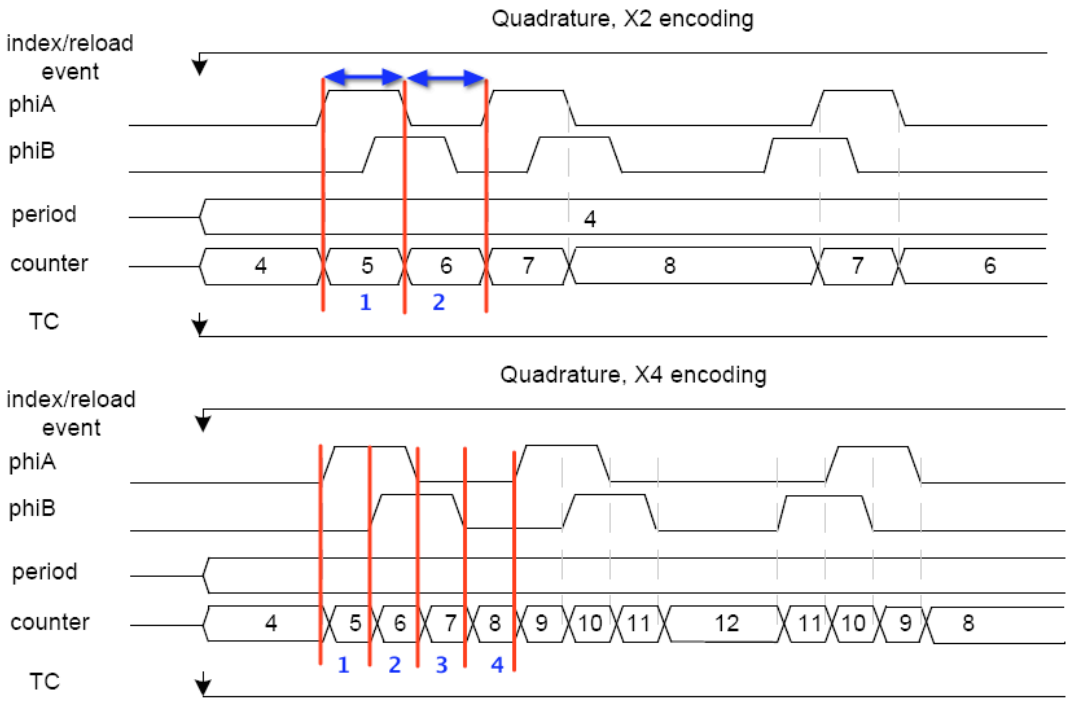


Bild 14: encoding Mode

Der **Stop** und **Index** Eingang wird beim normalen Einsatz als Drehgeber **nicht benutzt**.
Für die Eingänge **phiA** und **phiB** sollte ausschließliche der **Level-Mode** benutzt werden sagt das Datasheet.

Der Encoder hat ein Raster, sodass die Zwischenstufen gar nicht sichtbar sind. Dennoch werden diese Zwischenstufen korrekt durchlaufen, was man durchaus sehen kann, wenn man den Encoder ganz langsam dreht.

Das Beispiel ist sehr einfach gehalten und soll für die Experimente mit dem Rotary Encoder dienen.

Hardwareanforderung
049_RotaryEncoder_HW_not_debounced_1 ist für das CY8CKIT-049-0042 als Bootloadabe Projekt erstellt.
Zur Ausgabe wird ein 4*20 HD44780 komp. Display als Anzeige benutzt. Die aktuelle Anschlussbelegung des Displays kann im Projekt im .cydwr file eingesehen und ggf. angepasst werden.
Als Rotary Encoder wird ein Typ mit integriertem Switch verwendet. Getestet wurde mit den Typen:
Alps EC11B sowie dem Panasonic EVEQDBRL416B (die preiswerten Pollin-Teile).

Angezeigte Werte
enc: aktueller Counterwert der TCPWM-Komponente
em: aktueller Encoder Mode der TCPWM-Komponente
dir: Drehrichtung +(Uhrzeigersinn) –(gegen den Uhrzeiger)
counts: aktueller Counterwert der TCPWM-Komponente – Initial Counterwert (0x8000)

Bild 15: Anzeige RotaryEncoder 0 No Glitch

Bedienung
Mit dem Encoder-Switch wird der Encoder Mode der TCPWM-Komponente zwischen 1,2,4 durchgeschaltet.

Um die Daten eines unbekannten Encoders zu ermitteln, kann man folgendermaßen vorgehen:

1. Den **Encoder-Mode** solange durchtasten, bis der Zähler (enc, counts) mit jedem Rastenschritt um 1 erhöht/vermindert wird.

2. Encoder drehen bis counts bei 0 steht.
3. Encoder eine volle Umdrehung im Uhrzeigersinn

In **counts:** steht jetzt der Wert **detents.**
Pulses ergibt sich jetzt aus

$$pulses = \frac{encoderMode * detents}{4}$$

Dabei sollte die Drehung in normaler Geschwindigkeit erfolgen, da die Benutzung ohne Debouncing bei schnellen Drehungen zu Fehlern führen kann.

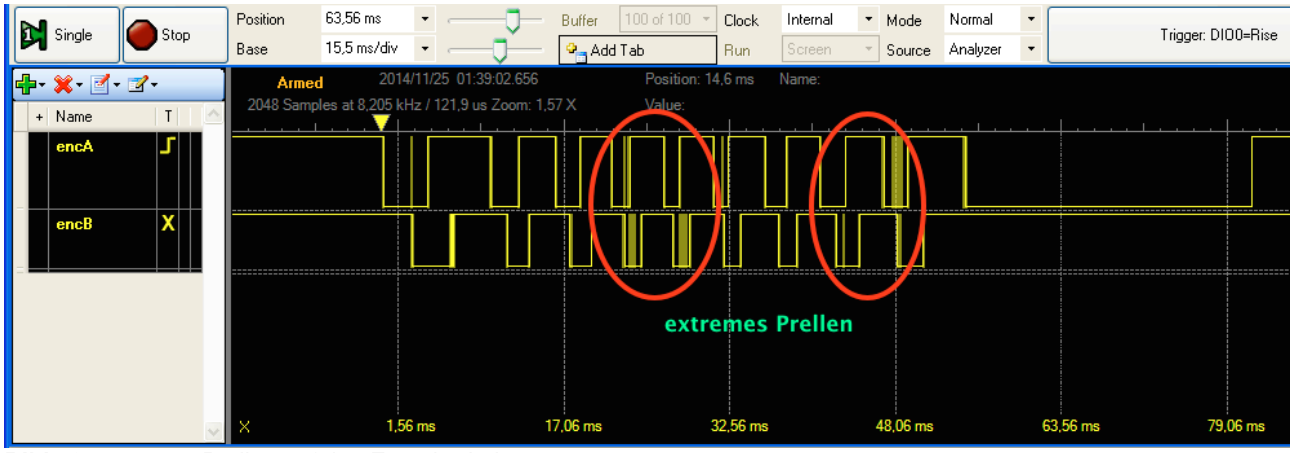


Bild 16: extremes Prellen auf den Encoder-Leitungen

Trotz des extremen Prellens auf den Encoderanschlüssen (**ohne RC Glieder**) zählt die TCPWM Komponente sehr exakt.
Entscheidend ist die clock Frequenz der TCPWM Komponente.
Sie muss einerseits niedriger sein als typische Prellimpulse und andererseits noch hoch genug, damit auch bei schnellen Encoder-Drehungen noch alle Impulse erfasst werden.
Sie hängt zudem noch davon ab, in welchem **encoding Mode** die **TCPWM** Komponente betrieben wird.

Mit folgenden Werten habe ich gute Erfahrungen gemacht:

Alps Encoder mit 20 Rasterungen	
encoding Mode	clock
x 1	1kHz
x 2	10kHz
x 4	100kHz

Tabelle 1: clock Frequenz je nach encoding Mode

Beschreibung der relevanten Softwarefragmente
Zum genauen Verständnis sollten sie die **main.c** der Projektes anschauen. Die relevanten Decoderinformationen werden im Beispiel in einer struct gehalten.

```
#define INITIAL_COUNTER 0x8000    //TCPMW InitialCounter
/*--------rotary struct----------*/
typedef struct {
   uint32 counterValue;           //actual counter
   volatile uint8 swichPressed;   //status of encoder switch
   uint32 direction;              //encoder direction
} ENCODER_t;
```

Der INITIAL_COUNTER wird durch die Komponente auf 0x8000 gesetzt, was genau der Hälfte des 16bit Counters entspricht.
Natürlich kann der Counter mit der API-Funktion TCPWM_WriteCounter() anders gesetzt werden, dabei ist aber zu beachten, dass die Komponente den Initialwert auch nach einem Counterüberlauf auf 0x8000 setzt.
Die Initialisierung der TCPWM-Komponente für den Betrieb als Quadraturdecoder ist etwas tricky.

```
QuadDec_Start();

//* see http://www.cypress.com/?app=forum&id=4749&rID=97375*//
QuadDec_TriggerCommand(QuadDec_MASK, QuadDec_CMD_RELOAD);

QuadDec_SetQDMode(QuadDec_MODE_X1); //initial mode
```

Nach dem Start der Komponente **muss** die API-Funktion _TriggerCommand() aufgerufen werden. Aus dem datasheet ist dieses Verhalten nicht ersichtlich. Ohne diese Funktion wird die interne Counter nicht inkrementiert.
Die API-Funktion _SetQDMode() setzt den Encoder Mode auf 1. Diese Zeile ist eigentlich überflüssig, da man den Encoder Mode ja bereits direkt in den Komponenten Eigenschaften setzten kann. Ich will hier nur den Zugriff per API zeigen, da in diesem Beispiel der Encoder Mode bei unbekannten Encodern geändert werden kann.
Den Encoder wird in der while() Loop ausgelesen.

```
myEncoder.counterValue = QuadDec_ReadCounter();  //read counter
myCounter = myEncoder.counterValue - INITIAL_COUNTER;
myEncoder.direction = QuadDec_ReadStatus() & 1; /* read direction
                                                    bit0 = QuadDec_STATUS_DOWN
                                                    bit1 = QuadDec_STATUS_RUNNING
                                                         direction ist _STATUS_DOWN
                                                          see TRM PSoC 4200 S. 176
                                                   */
```

Die Variable myCounter entspricht dem Counterwert der Komponente vermindert um den Initialwert. Er beginnt somit bei 0.
Falls die Richtung des Encoders benötigt wird, kann die natürlich aus der Differenz des vorhergehenden zum aktuellen Counterwert ermittelt werden.
Es besteht allerdings auch die Möglichkeit die letzte Drehrichtung mit der API-Funktion _ReadStatus() aus der Komponente zu lesen.
Hat die **isr** einen Tastendruck registriert, wird der EncoderMode entsprechend weitergeschaltet.

```
if (myEncoder.swichPressed){  //set encoder mode
    encoderMode++;
    if (3 == encoderMode) { //only 3 different modes
        encoderMode = 0;
    }
    switch(encoderMode){
    case 0: QuadDec_SetQDMode(QuadDec_MODE_X1); break;
    case 1: QuadDec_SetQDMode(QuadDec_MODE_X2); break;
    case 2: QuadDec_SetQDMode(QuadDec_MODE_X4); break;
    }
```

```
    myEncoder.swichPressed = 0;
}
```

Bei der Anzeige des Encoder Mode muss noch berücksichtigt werden, dass der Zähler encoderMode von 0 bis 2 zählt was aber den tatsächlichen Encoder Modes 1,2,4 entspricht.

```
LCD_Char_PrintNumber(1 << encoderMode);  //encoder mode is 1,2,4
```

Zusammenfassung
Normale Rotary Encoder können durchaus ohne debouncing-Maßnahmen mit der TCPWM Komponente eingesetzt werden. Dabei muss allerdings eine passende clock-Frequenz gewählt werden.

Um den **Encoder Taster** sinnvoll zu verwenden, sollte dieser unbedingt debounced werden

Kurzanleitung zum Einsatz

1. Komponente konfigurieren clock je nach Auflösung (siehe Tabelle)
2. Software Implementation

rotary struct

```
/*--------rotary struct----------*/
typedef struct {
    uint32 counterValue;           //actual counter
    volatile uint8 swichPressed;   //status of encoder switch -- if you want
    uint32 direction;              //encoder direction  -- if needed
} ENCODER_t;

ENCODER_t myEncoder = {
    .counterValue = 0,
    .swichPressed = 0,
    .direction = 0,
};
```

Componentenstart

```
QuadDec_Start();
/*see http://www.cypress.com/?app=forum&id=4749&rID=97375*/
QuadDec_TriggerCommand(QuadDec_MASK, QuadDec_CMD_RELOAD);
```

Encoder auslesen

```
myEncoder.counterValue = QuadDec_ReadCounter(); //read counter
myEncoder.direction = QuadDec_ReadStatus() & 1; /*read direction
                                          bit0 = QuadDec_STATUS_DOWN
                                          bit1 = QuadDec_STATUS_RUNNING
                                          direction ist _STATUS_DOWN see
                                          TRM PSoC 4200 S. 176
                                         */
```

Ressourcenbedarf
Speziell beim Einsatz von PSoC 4 muss man die Hardwareressourcen immer im Blick haben, schließlich gibt es da nicht so viele. Zwar gibt es ganze 4 TCPWM Components in PSoC 4, aber die will man sicher nicht für Eingabedrehschalter verbraten.

Der Ressourcenbedarf der TCPWM Komponente ist wie folgt angegeben.
Neben der eigentlichen Komponente werden 572Bytes an Flash und 2 Bytes an SRAM verbraucht.

Configuration	PSoC 4 (GCC)			
	PSoC 4000		PSoC 4100/PSoC 4200	
	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes
Unconfigured	1144	2	1036	2
Timer / Counter	752	2	752	2
Quadrature Decoder	572	2	572	2
PWM	1128	2	1020	2

3.2 mit Glitchfilter als debouncer

Projekt: 049_RotaryEncoder_HW_debounced_1.
Sinn des Glitchfilter-Einsatzes ist es, die Prellimpulse an den Encoderkontakten zu beseitigen und damit eine sicherere Arbeitsweise zu gewährleisten. Das kann sinnvoll sein, wenn eine 100% genaue Zählung der Encoderimpulse sichergestellt werden muss.
Allerdings wird dies mit einem höheren Hardwareaufwand erkauft.

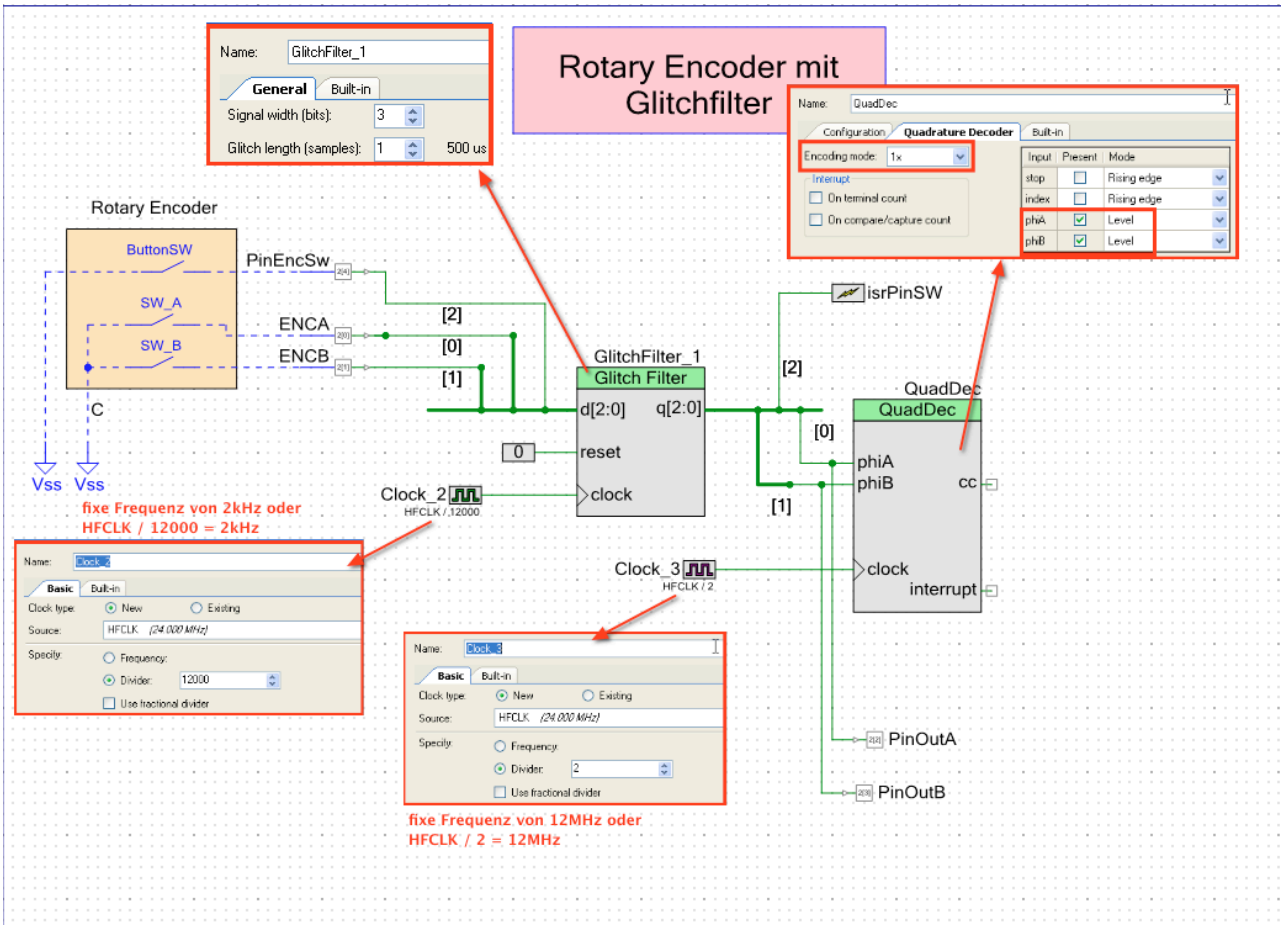


Bild 17: Verdrahtung Rotary Encoder mit Glitchfilter und TCPWM component

Die Software ist identisch zur Version ohne Debouncing.

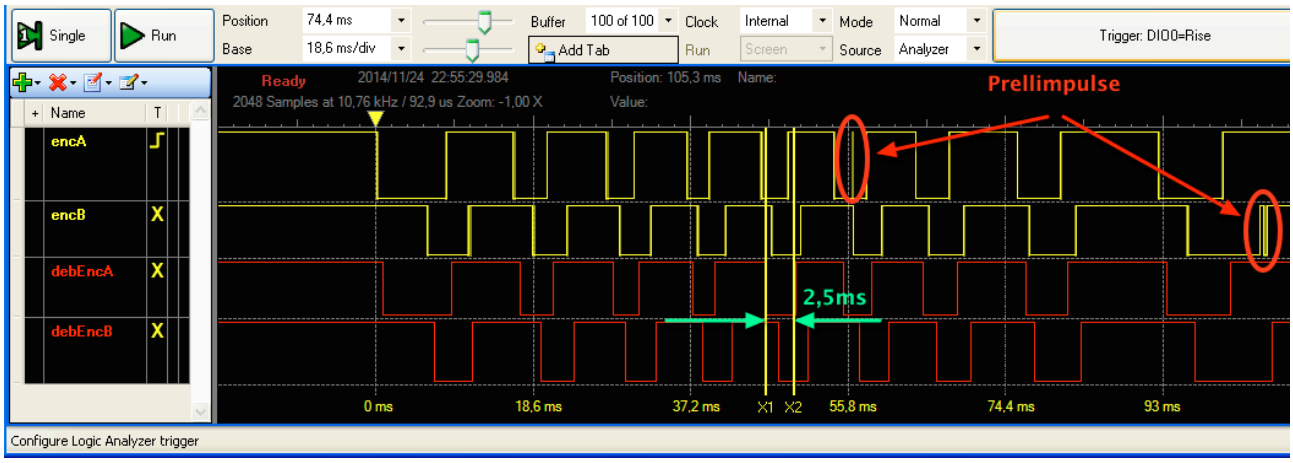


Bild 18: Sehr schnelle Drehung von Hand, **2kHz clock an TCPWM**

Bei Bild 18 betrugen die Einstellungen:

Glitch Lenght (sample): 500us
clock TCPWM: 2 kHz

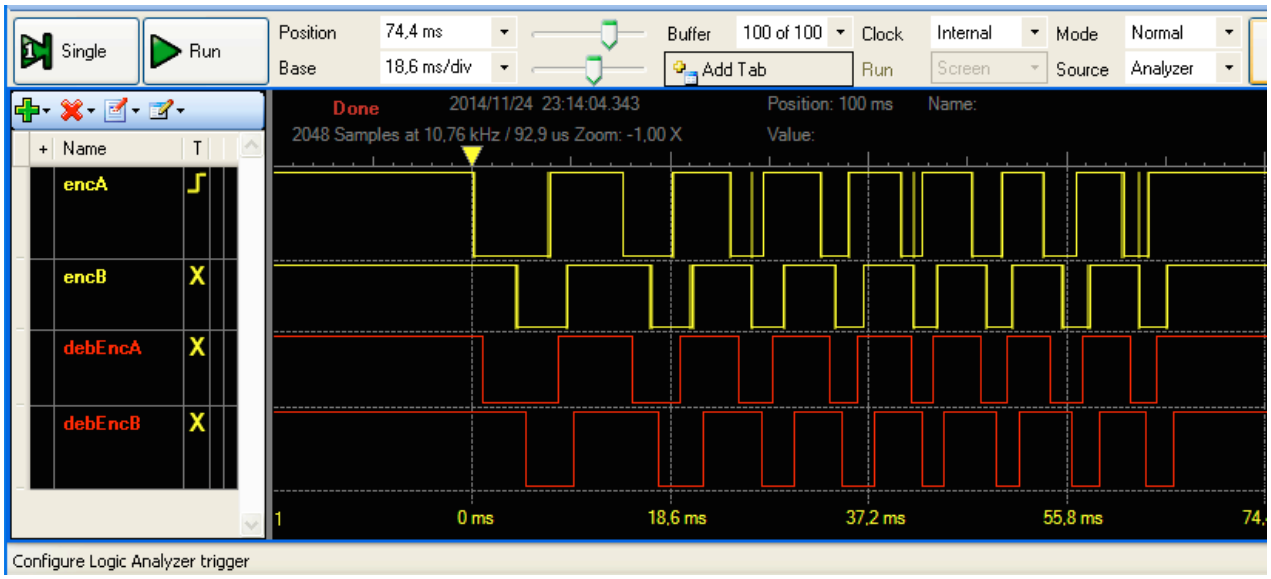


Bild 19: Sehr schnelle Drehung von Hand, **separater 12MHz clock an TCPWM**

An Bild 18 und Bild 19 kann man gut erkennen, dass die Prellimpulse sicher beseitigt werden und die Größe des TCPWM clock keine Rolle mehr spielt.

Ressourcenbedarf
Speziell beim Einsatz von PSoC 4 muss man die Hardwareressourcen immer im Blick haben, schließlich gibt es da nicht so viele. Zwar gibt es ganze 4 TCPWM Components in PSoC 4, aber die will man sicher nicht für Eingabedrehschalter verbraten.

Neben dem Speicherverbrauch der TCPWM Komponente werden für das Glitchfilter noch die folgenden Ressourcen verbraucht.

Configuration	Resource Type					
	Datapath Cells	Macrocells	Status Cells	Control Cells	DMA Channels	Interrupts
Glitch Length ≤ 8	–	N * (L + 1) [1]	–	–	–	–
8 < Glitch Length ≤ 256	N	N	–	–	–	–

1. N – Signal width; L – Filtering length

Inklusive des Tasters werden 3 * (1 + 1) = **6 Macrocells** von 32, die in PSoC 4 zur Verfügung stehen, verbraucht.

Zusammenfassung
Durch die Vorschaltung eines Glitchfilters kann ein Prellen der Encoderkontakte sicher beseitigt werden und die Zuverlässigkeit somit erhöht werden. Die clock-Frequenz der TCPWM-Componente spielt dann keine dominate Rolle mehr.
Das Glitchfilter kann die Encoder-Taste zudem mit debouncen.
Allerdings wird der Komfort mit zusätzlicher Hardware erkauft und die ist speziell bei PSoC 4 knapp.

4 Auswertung mit Quadrature Decoder component

Der Einsatz der Quadratur Decoder Component ist bei PSoC 4 nicht sinnvoll, da die UDB Komponente enorme Ressourcen benötigt:

Resources

The Quadrature Decoder component is placed throughout the UDB array. The component utilizes the following resources.

Configuration	Resource Type					
	Datapath Cells	Macrocells	Status Cells	Control Cells	DMA Channels	Interrupts
8-bit, resolution 1x, no glitch filtering, use index	1	22	2	1	–	–

Configuration	Resource Type					
	Datapath Cells	Macrocells	Status Cells	Control Cells	DMA Channels	Interrupts
16-bit, resolution 2x, glitch filtering, use index	2	31	2	1	–	–
32-bit, resolution 4x, glitch filtering, use index	2	32	2	1	–	1

Note The PSoC 4200 family can support an 8-, 16-, or 32-bit counter size with glitch filtering disabled. Other configurations are too large for this family.

Der Einsatz des internen Componenten Glitch-Filters ist bei der PSoC 4200 Familie hier gar nicht mehr möglich.

5 Realisierung in Software

Für die Auswertung in Software gibt es, trotz der in den PSoC Bausteinen enthaltenen Quadratur Decoder, gute Gründe.

- PSoC 4 hat nur recht begrenzte Hardwareressourcen.
- Die UDB Quadraturdecoder ist unter PSoC 4 unverhältnismäßig ressourcenhungrig.
- Die 4 TCPMW – Komponenten sind zwar hervorragend geeignet, aber da gibt es eben nur 4 davon.
- Bei den fertigen Hardware-Komponenten ist es schwierig, ggf. eine Dynamik zu implementieren.
- Die Realisierung, besonders wenn es sich um einfache mechanische Drehgeber handelt, die für simple Steuerungsaufgaben verwendet werden, kann recht einfach gestaltet werden und nimmt dann kaum mehr Speicherplatz in Anspruch als die gezeigten Hardwarelösungen.
- Sollen mehrere Encoder eingesetzt werden, ist man unter PSoC 4 mit den Hardwarekomponenten ohnehin schnell am Ende.
- In Software lässt sich eine Dynamik recht einfach implementieren.
- Die Softwarelösung ist i.d.R. leichter auf andere Plattformen portierbar.
- Die Softwarelösung kann auch problemlos für den Anschluss mehrerer Encoder eingesetzt werden.

Speziell für den Ersatz der Hardwarekomponente macht es sicher Sinn, die Hardwarekomponente quasi in Software nachzubauen. Diese besteht intern im Wesentlichen aus einem Decoder und dem Counter.

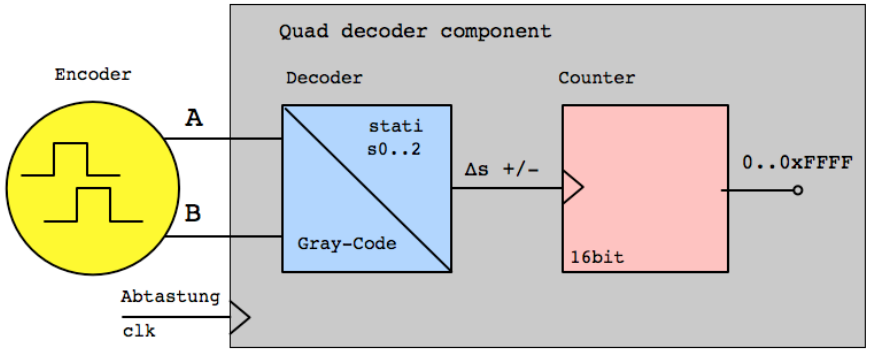


Bild 20: Ersatzschaltung der Hardwarekomponente (nur die wesentlichen Elemente)

Für die Realisierung in Software haben sich 2 verschiedene Verfahren etabliert.

A. **Das Abtastverfahren (IR Polling)**
   Dabei werden die Encoder-Anschlüsse A und B zyklisch abgetastet (Polling) und deren Status ausgewertet. Sinnvoller Weise geschieht dies per Timerinterrupt (IR Polling). Das Prellen der Kontakte wird hier bereits durch die Wahl der Abtastfrequenz (<= 1kHz für Drehencoder) nahezu eliminiert.
   Das Verfahren ist einfach zu implementieren. Allerdings wird die ISR des Timers auch dann durchlaufen, wenn überhaupt keine Drehbewegung des Encoder erfolgt. Ein Timerinterrupt wird idR. ohnehin für viele andere Aufgaben innerhalb eines Programmes benötigt, sodass der Mehraufwand kaum ins Gewicht fällt.
B. **Die Interruptauswertung (Interruptverfahren)**
   Hintergedanke dieses Verfahrens ist es, dass sich der Encoder per Interrupt meldet, wenn er bewegt wird, wodurch die Prozessorlast gesenkt werden kann.
   Durch geschickte Programmierung kann hier ebenfalls der Aufwand recht gering gehalten werden.

Über die Vor- und Nachteile beider Verfahren werden zum Teil erbitterte Kriege geführt;-)
z.b. hier: https://www.mikrocontroller.net/topic/378062#new
Ich will hier einfach beide Verfahren vorstellen und als Beispiel implementieren. Welches Verfahren dann jeweils genutzt wird kann anhand der konkreten Applikation entschieden werden.

5.1 Das Abtastverfahren (IR Polling)

Im Grunde sind 3 Teilaufgaben zu lösen.

1. Abtastung der Dekoderimpulse
2. Decodierung des Graycode (stati) mit Ermittlung der Differenz zum vorhergehenden Status
3. Ein Counter der je nach Differenz hoch oder runtergezählt wird.

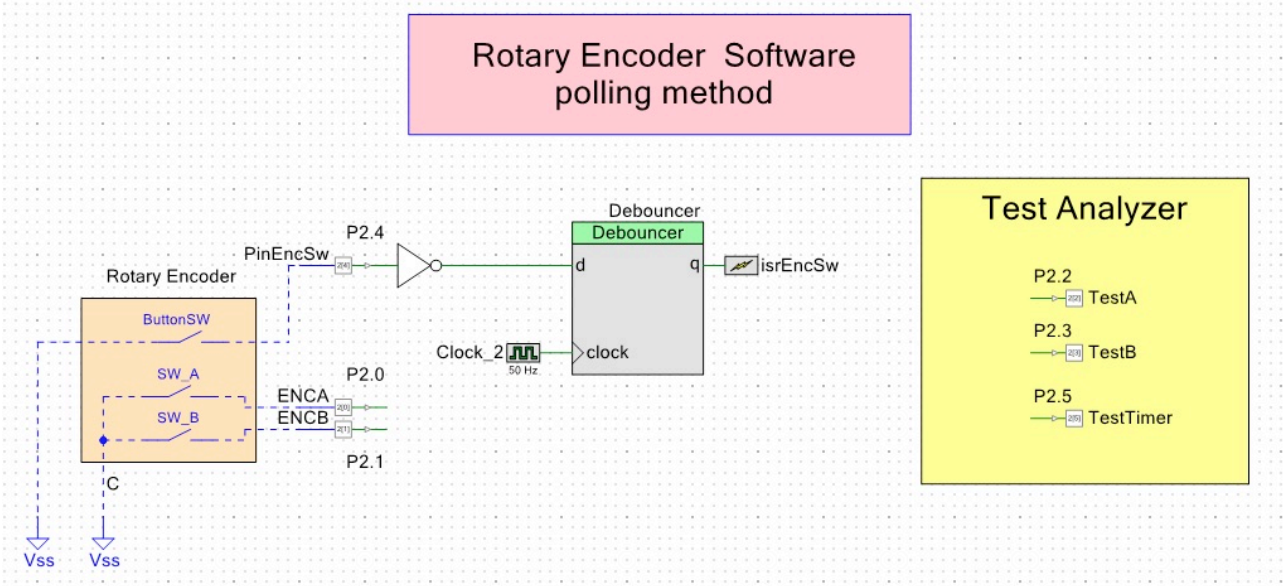


Bild 21: Encoder Ankopplung nach der polling Methode, ohne Debouncing

Die Abtastung des Encoder geschieht im Abstand von 1ms durch den SystickTimer des M0.

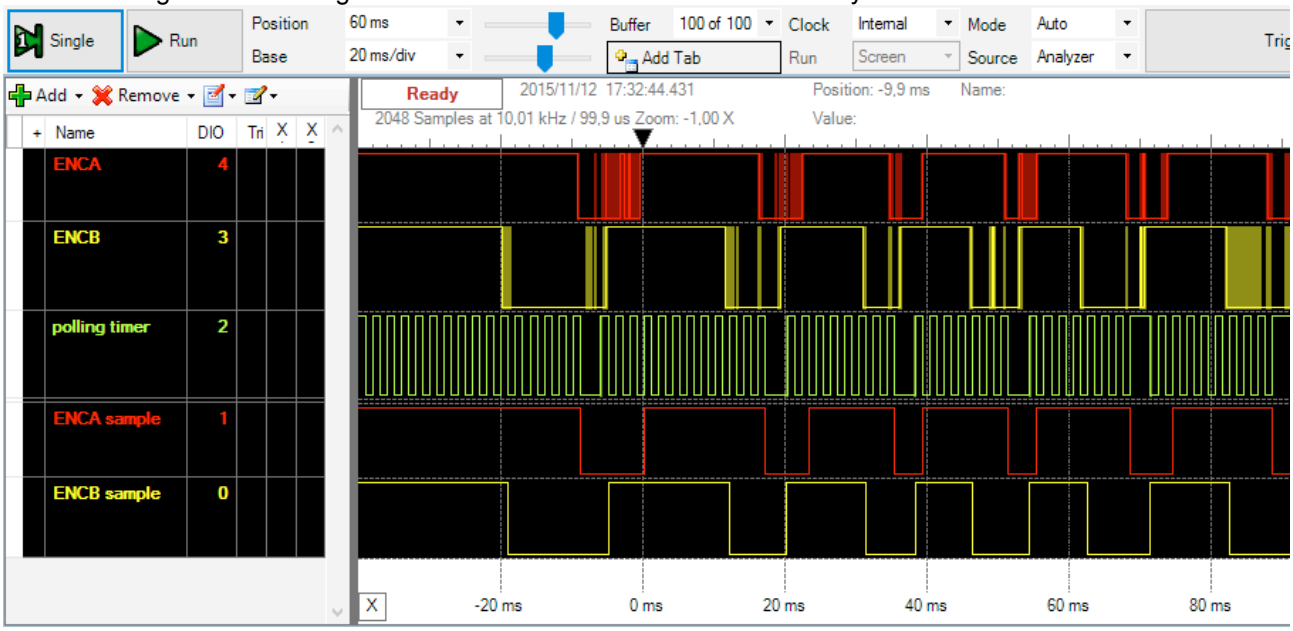


Bild 22: Abtastung des Encoder durch einen 1ms Impuls

Da die Auswertung des Timerablaufes hier in der Main-Loop erfolgt(die isr setzt lediglich ein Flag), werden Timerabläufe ausgelassen, wenn eine LCD-Ausgabe erfolgt.
Praktische Auswirkungen hat das nicht, wie man sieht. Aber natürlich kann man die Encoder-Auswertung auch direkt in die ISR des SysTimers platzieren, dann haben andere Funktionen in der Main-Loop keinen Einfluss mehr.
Es handelt sich um einen mechanischen Alps-Encoder mit 20 Rasten pro Umdrehung, der sehr stark prellt.

Number of detent = 20
Number of pulse = 20

Der Encoder wurde mit normaler Geschwindigkeit gedreht.

Der Encoder ist direkt an den Anschlüssen **encA** und **encB** angeschlossen, die hier auch direkt oszillografiert sind. (Bild 21) Er wird nicht entprellt.
In der Timerbehandlung werden die Encoder Eingänge gelesen, dekodiert und die Testsignale (**TestA**, **TestB**) zum oszillografieren gesetzt.

Man erkennt ganz deutlich, dass die Störimpulse allein durch die Abtastung nahezu komplett beseitigt sind.

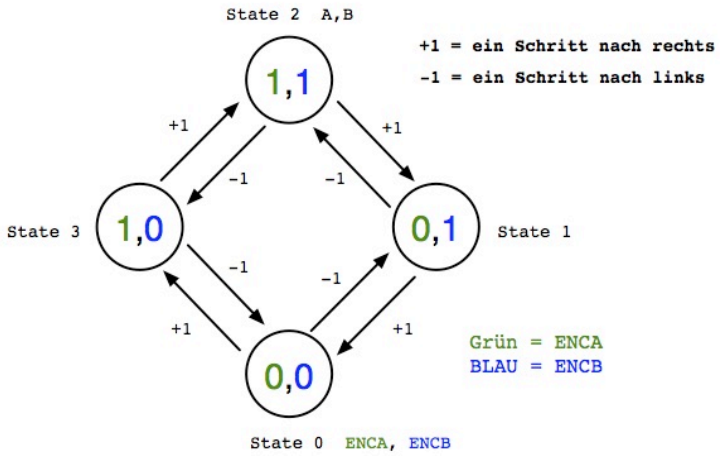
Hinweis
Für sehr schnelle Drehungen von Hand ist die Abtastfrequenz von 1kHz zu klein. Sollte dies nötig sein sollte muss die Abtastfrequenz größer gewählt werden.

Natürlich sind zur Abtastung der Dekoderimpulse viele andere Möglichkeiten denkbar. Und statt des Systick Timers kann selbstverständlich auch eine Timerkomponente eingesetzt werden.
Mit dem hier vorgestellten Verfahren habe ich aber sehr gute Erfahrungen gemacht.

Der Decoder Algorithmus
Für die Implementierung in Software finden sich im Netz schier endlose Möglichkeiten. Eine sehr gute Anlaufstelle ist z.B.: http://www.mikrocontroller.net/articles/Drehgeber

Für die Erfassung der Encoderimpulse benutze ich hier einen Timer der jede ms die Eingänge ENCA und ENCB abfragt.
Die Auswertung der Drehimpulse und der Drehrichtung kann dann z.B. in einer **state machine** erfolgen. Die state machine hält Status und Folgestatus in einer Tabelle. Die Drehrichtung und die Gültigkeit des Statuswechsels kann so direkt aus der Tabelle abgeleitet werden. Ein Beispiel findet sich im o.g. Artikel.
Eine besonders clevere Variante ist von **Peter Dannegger** (ebenfalls in o.g. Artikel). Diese Variante kommt ohne die klassische (tabellenorientierte) state machine aus, indem sie den Folgestatus aus dem Gray Code berechnet. Genau genommen handelt es sich hierbei also ebenfalls um eine state machine.
Ich zeige hier eine angepasste Variante.

Wie arbeitet das Verfahren nun genau? Dazu noch einmal das state Diagramm des Encoders.



Gray	State	State-binär
AB	S	S
00	0	00
01	1	01
11	2	10 <– /bit0
10	3	11 <– /bit0

Bild 23: Zusammenhang zwischen Gray-Code und den einzelnen Stati

Für die Umsetzung mit einer klassischen state machine spielt die Nummerierung der einzelnen stati keine Rolle, da der Zusammenhang zwischen einem Status und dem Nachfolgestatus fest in einer Tabelle codiert ist.
Für das Verfahren nach **Peter Dannegger** ist es wichtig, dass die stati fortlaufend nummeriert sind. Zumindest lässt sich so eine einfache Implementierung realisieren. Die Zuordnung erfolgt wie in der Tabelle von Bild 23 zu sehen.
Wie man sieht bildet der Gray-Code den Status nicht direkt ab. Bei den stati 3 und 4 ist das bit0 im Gray-Code negiert.
Der Gray-Code muss also wie in der Tabelle zu sehen, in einen einfachen Binärcode (0..3) gewandelt werden. Eine einfache Umwandlung, die nur binäre Operationen verwendet kann so aussehen:

```
s = encB;
```

```
if (encA) {s ^= 3;}  //s = encB ^3

/*
  encB == 0  -> S = 0^3 =  00
                          ^11
                          ---
                           11 = 3
  10 -> 11

  encB = 1  -> S = 1^3 =  01
                         ^11
                         ---
                          10 = 2
  11 -> 10
*/
```

Man erkennt, wie mit Hilfe der XOR Operation das Bit 0 negiert wird.
Man kann jetzt mit dem stati s weiterrechnen.

Die Differenz zum vorhergehendem Status ergibt sich aus:

```
s -= slast;
```

Für bit0 und bit1 ergeben sich folgende Differenzen:

Vorwärts:
s - slast
00 - 11 = 01 (1) b00000000 – b00000011 = b11110101
01 - 00 = 01 (1)
10 - 01 = 01 (1)
11 - 10 = 01 (1)

Rückwärts:
s - slast
00 - 01 = 11 (3)
01 - 10 = 11 (3)
10 - 11 = 11 (3)
11 - 00 = 11 (3)

Die Änderung kann also **nur 1** (vorwärts) **oder 3** (rückwärts) sein. Alle anderen Änderungen können nur Störungen sein (oder der status wurde gar nicht gewechselt) und sind ungültig.
Falls es sich also um einen gültigen Statuswechsel handelt **(**Bit 0 muss gesetzt sein **S & 1)** wird zuerst der status als letzter status (slast) gespeichert. Da der inzwischen als Differenz zum alten Status vorliegt **(S -= slast)**, muss die Differenz auf den letzten Status addiert werden um den aktuellen Status zu speichern.
Wie man an den obigen Differenzen sieht, entspricht **bit1** der Drehrichtung 0 -> vorwärts, 1-> rückwärts.
Wird bit0 gelöscht (S & 2) erhält man 0 -> vorwärts, 2 -> rückwärts.
Und mit -1 schließlich -1 -> vorwärts und 1 -> rückwärts.
Der Wert **counterValue** entspricht direkt einem zur Drehung proportionalem Counter.

```
If (s & 1 ) {    //if bit0 set (diff is 1 or 3) or shorter if (s)
   slast += s;   //s is now s-slast (S -= slast) -> slast + s – slast = s

   /*
    bit 1 is the direction bit
    01->forward   & 2 = 00 -> 00 - 1 = -1
    11->backward  & 2 = 10 -> 10 - 1 = +1
   */

   counterValue+= (S & 2) -1; //clear bit -> (0 fw; 2 bw)-1
}
```

Hinweis
Wenn die Encoder-Pins als PinBlock definiert sind, spart man sich einen Pin_Read(), da der Block als ganzes ausgelesen wird.

```
s = encAB_Read();
```

Um bit0 zu negieren muss der XOR nun mit 1 (bit0) erfolgen, da bit1 ja schon in s enthalten ist

```
if ((s & 2) >>1) {s ^= 1;}  //if bit1 -> /bit0  alternativ: if (s!=0){s^=1;}
```

Im Beispiel **049_RotaryEncoder_SW_Polling_1_B** wird dies gezeigt.

Wie man sieht, wird jeder Statuswechsel registriert. Der kann jetzt dazu benutzt werden einen Counter (counterValue) zu inkrementieren/dekrementieren. Das wird in der Originalversion von Peter Dannegger auch so gemacht.
Dabei kann es aber vorkommen, dass **der Counter nicht in 1er Schritten** zählt, **falls detents != pulses** (siehe Bild 4).
Je nach dem Verhältnis von detents/pulses (encoding modes) zählt der counter in 1er, 2er oder 4er Schritten pro Rasterung.
Natürlich kann der Counterwert wieder durch den encoding mode geteilt werden, dann zählt der Counter immer in 1er Schritten.
Das hat allerdings 2 Nachteile.

1. Der Counterwert ist u.U. vom Encodertyp abhängig
2. Es kann u.U. nicht mehr die volle Counterauflösung genutzt werden.

Besser ist es, nur die Rasten statt der stati zu zählen. Das geht zwar nicht direkt, weil die Rasten ja keine Kontakte sind, die man irgendwie registrieren könnte.
Nach Bild 14 bedeutet der encoding mode ja wieviel stati pro pulse gezählt werden.

encoding mode = 1 - 1 counter-wert pro pulse
encoding mode = 2 - 2 counter-werte pro pulse
encoding mode = 4 - 4 counter-werte pro pulse

Innerhalb eines pulses können eine, zwei oder vier Rasten liegen
Bei einer Raste pro pulse (detent = pulses) muss der **encoding mode 4** sein damit **nur jeder 4. Stati gezählt wird.**
Bei zwei Rasten pro pulse (detent = 2 * pulse) muss der **encoding mode 2** sein damit **nur jeder 2. Stati** gezählt wird.
Bei vier Rasten pro pulse (detent = 4 * pulse) muss der **encoding mode 1** sein damit **jeder Stati** gezählt wird.
Daraus ergibt sich:

$$encodingMode = \frac{4 * pulse}{detent}$$

Statt

```
    counterValue += (S & 2) -1; //clear bit -> (0 fw; 2 bw)-1
```

kann man den encoder mode wie folgt berücksichtigen:

```
if (0 == slast % (1 << encoderMode){ // only the right state 0,1,2->1,2,4
     counterValue += (s & 2) - 1;
      direction = ((s & 2) >> 1) ? '+':'-';  //bit1 = encoder direction
}
```

Der Encoding mode 1,2,4 wird hier in der Form 0,1,2 benutzt, das ist nützlich, da sich die einzelnen mode 1,2,4 dann einfach als bit-Verschiebung um 1,2 oder 3 Stellen realisieren lassen.

Hinweis
In einer normalen Applikation ergibt sich der encoding mode ja aus den Encoderkenngrößen detent und pulse. Lediglich in meinem Beispiel kann der encoding mode durchgeschalten werden um die Auswirkung sichtbar zu machen und ggf. die Kenngrößen unbekannter Encoder zu ermitteln.

Im **Projekt 049_RotaryEncoder_SW_Polling_1** habe ich die vorangegangenen Beispiele als reine Softwareimplementierung realisiert. Lediglich die Encoder Taste wird durch eine Debouncer-Komponente entprellt.
Als Timer benutze ich hier wie bereits erwähnt den SysTick-Timer des M0 Prozessors. Zur einfachen Handhabung setze ich diesmal die Community-Komponente SysTimers ein.
Siehe **Exkurs: Die SysTimers Komponente**.

Hinweis
Wie die SysTick-Timer direkt benutzt wird habe ich z.B. im Dokument **CY8CKIT-049-4xxx PSoC 4 Taster mit Debouncer.pdf** beschrieben.

Im Vergleich zu den vorhergehenden Varianten kann im Verhalten eigentlich kein Unterschied festgestellt werden

Hinweis
Die komplette encoder Funktionalität wurde in diesem Beispiel in ein eigenes Modul **encoder_sw_single.c/.h** ausgegliedert. Im Grunde beinhaltet das Modul neben der encoder struct lediglich die Funktionen **encoder_init()** und **encoder_update().** Für den Einsatz der Hardware-Komponenten ist es sinnvoll, sich ebenfalls ein Modul mit identischer encoder struct und den beiden Funktionen zu schreiben. In Applikationen kann so leicht je nach Bedarf die Hard- oder Softwareversion benutzt werden.

Die Implementierung
Zur Übersicht hier nach einmal die Implementierung als Gesamtzusammenhang.

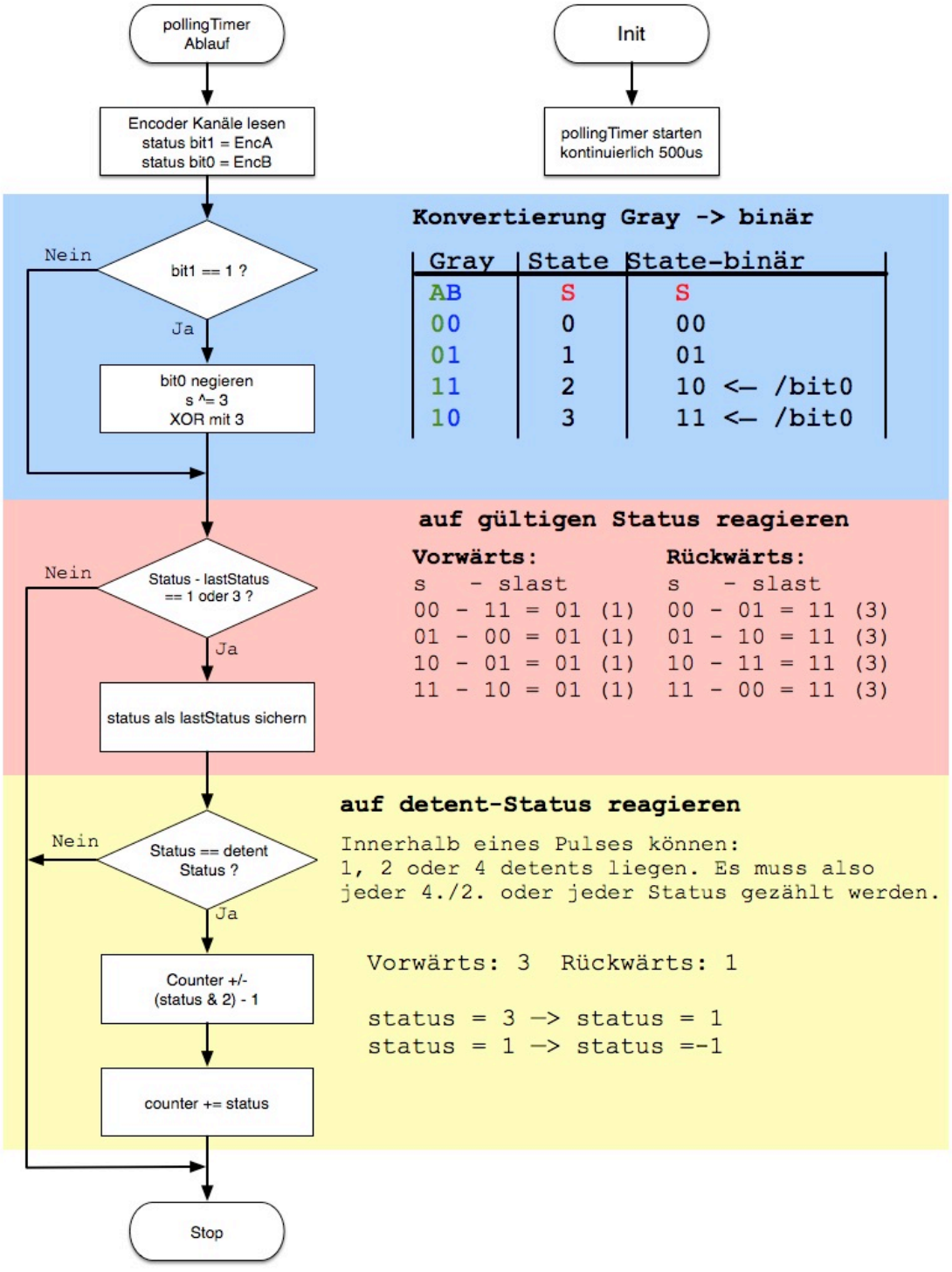
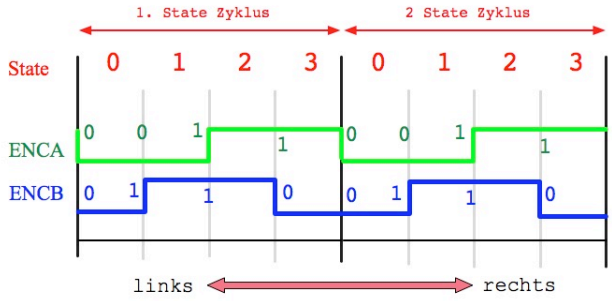


Bild 24: Die Implementierung des Pollingverfahrens

5.2 Softwarelösung nach dem Interruptverfahren

Beispiel Projekt **049_RotaryEncoder_SW_IR_not_debounced_1**
Bei der folgenden Betrachtung bleibt die Prellproblematik einstweilen außen vor.
Zum Verständnis des Interruptverfahrens hier noch einmal eine etwas modifizierte Ansicht der beiden Encoderkanäle je nach Drehung rechts/links.



Gray	State	State-binär
AB	S	S
00	0	00
01	1	01
11	2	10 <– /bit0
10	3	11 <– /bit0

Bild 25: Encoderkanäle in Abhängigkeit der Drehrichtung

In Bild 25 zeigt wieder die bekannte Aufteilung eines pulses in 4 Stati. Man erkennt, dass sich ein Stati auch über die Flanke eines Kanals und den Zustand des anderen Kanals zu diesem Zeitpunkt definieren lässt.
Eine steigende Flanke repräsentiert 1 und eine fallende Flanke eine 0.
Damit ergibt sich genau die gleiche Tabelle, wie aus Bild 23. Ist auch logisch, schließlich hat sich die Arbeitsweise des Encoders ja nicht geändert.
Der Unterschied besteht lediglich im Zeitpunkt der Stati-Erkennung. Ein Stati wird hier durch den ir einer Flanke eingeleitet. Um welchen Stati es sich handelt, bestimmt dann der Zustand des anderen Kanals.

Das bedeutet, dass zur Auswertung des Encoders im IR-Verfahren prinzipiell die gleichen Softwareroutinen wie beim Polling-Verfahren benutzt werden können. Lediglich die Art der Status-Erkennung ändert sich.
Allerdings wird beim IR Verfahren statt der state machine Auswertung häufig auch direkt die Reihenfolge der auftretenden Kanal IRs ausgewertet.
Ich bleibe aber bei der state machine, weil sich beide Verfahren so sehr gut vergleichen lassen.

Wie man am Bild 25 gut erkennen kann, werden die Flanken beider Kanäle jeweils im Wechsel benötigt um die 4 Stati eines pulses erfassen zu können. Es werden also Interrupts beider Kanäle benötigt. Innerhalb der isr wird dann ermittelt, ob es sich um eine rising oder falling edge handelte.

Im Pollingverfahren wurde das Kontaktprellen auf 2 Arten eliminiert.

1. Durch den Sample-Abstand verschwanden die meisten Störimpulse allein dadurch, dass sie innerhalb der ersten Sample-Pause auftreten und somit nicht erfasst werden.
2. Treten dennoch Störimpulse auf, führten sie dazu, dass der counter u.U. um einen Schritt weiter und mit der 2. Flanke des Störimpulses wieder zurück gezählt wird. Siehe Bild 7.

Beim IR Verfahren wird nicht gesampled, sodass die Störimpulse hier voll wirken könnten.
Das Verfahren an sich funktioniert korrekt. Das Beispiel **049_RotaryEncoder_SW_IR_0**
Wurde jeweils mit einem Panasonic EVEQ sowie mit einem Alps EC11B getestet.
Die isr sind wie folgt aufgebaut:

```
//isr_ENCA
CY_ISR(isr_ENCA){
    /* test the right bit from  port status register*/
```

```
    if (ENCA_INTSTAT & (1u << ENCA_SHIFT)){

        isrENCA_Disable();  //disable isrENCA

        read_encoder();     //inline function for read and decode encoder

        /*clear old ir on encb ans allow ir on encb*/
        isrENCA_Enable();
        ENCA_ClearInterrupt();
    }
}

//isr_ENCB
CY_ISR(isr_ENCB){
   /* test the right bit from  port status register*/
   if (ENCB_INTSTAT & (1u << ENCB_SHIFT)){

        isrENCB_Disable();  //disable isrENCA

        read_encoder();     //inline function for read and decode encoder

        /*clear old ir on enca ans allow ir on enca*/
        isrENCB_Enable();
        ENCB_ClearInterrupt();
   }
}

/**
 * Function:    read_encoder
 * Description:
 *     inline function decode gray-code and count encoder detents

 @param        void
 @return       void
*/
inline void read_encoder(void){
    uint8 encA = ENCA_Read();
    uint8 encB = ENCB_Read();
    uint8 s = encB;         //stati

    if (encA) {s ^= 3;}     //S = ENCB ^3

    s -=  myEncoder.sLast;
    if (s & 1) {              //if bit0 set (diff is 1 or 3)
        myEncoder.sLast += s; //s is now s-sLast -> sLast + s - sLast = s
        if (0 == myEncoder.sLast % (1 << myEncoder.encoderMode) ) {
            uint32 lastCounterValue = myEncoder.counterValue;
            myEncoder.counterValue += (s & 2) - 1;   //clear bit -> (0 vw; 2 bw)-1
            myEncoder.direction = (lastCounterValue < myEncoder.counterValue) ?
                                    '+':'-'; //encoder direction
        }
    }
}
```

Die Sperrung des IR innerhalb der ISR ist nötig, um zu verhindern dass der IR bei Glitches sofort erneut ausgeführt wird. Deswegen werden auch GPIOs verschiedener Ports benutzt, da sich der IR nur für den kompletten Port sperren/freigeben lässt.
Wie man sieht, ist die Routine zum lesen des encoders identisch mit dem polling – Verfahren.

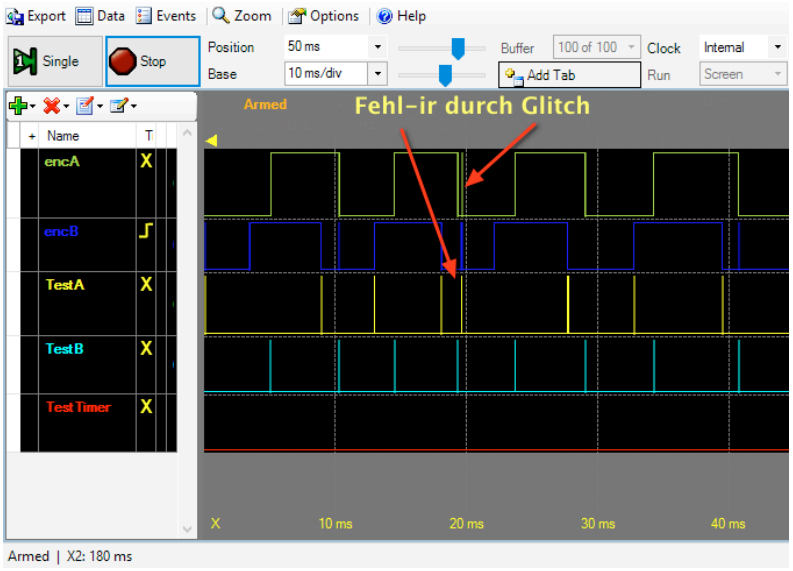


Bild 26: Das IR-Verfahren mit einem Panasonic EVEQ

In Bild 26 wurde jeweils die isr mit einem Testpin (TestA/TestB) oszillografiert. Der Encoder prellt kaum. Das Beispielprogramm zählt i.d.R. korrekt. Allerdings kommt es gelegentlich zu Fehlzählungen durch zusätzliche Interrupts, die durch Glitches ausgelöst werden wie im Bild zu sehen.

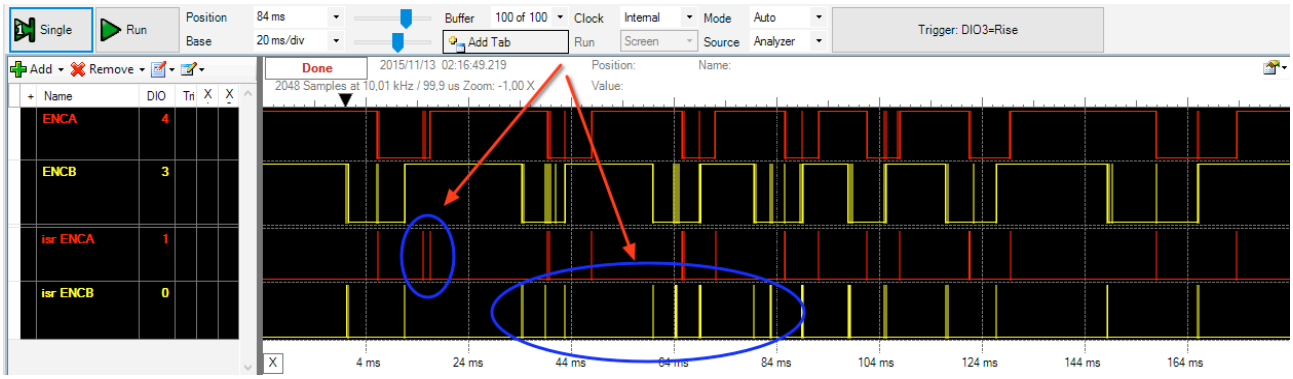


Bild 27: Das IR-Verfahren mit einem extrem prellenden Alps EN11B

Bild 27 zeigt die gleiche Situation mit einem extrem prellendem Alps EN11B. Hier arbeitet das Verfahren nicht mehr sinnvoll. Die Prellimpulse können außerdem zu einem extremen Interruptaufkommen führen.

Im Vergleich, der gleiche Encoder ausgelesen im Pollingverfahren mit Beispielprogramm **049_RotaryEncoder_SW_Polling_1.** Siehe Bild 22
Hier wird der Encoder akkurat ausgelesen.

Ganz anders sieht die Situation aus, wenn der Encoder z.B. durch ein Glitchfilter entprellt wir. Das erhöht zwar etwas den Hardwareaufwand, aber es mag Situationen geben, wo diese Variante die beste Wahl ist. Beispielprogramm **049_RotaryEncoder_SW_IR_debounce_1** zeigt eine solche Variante.

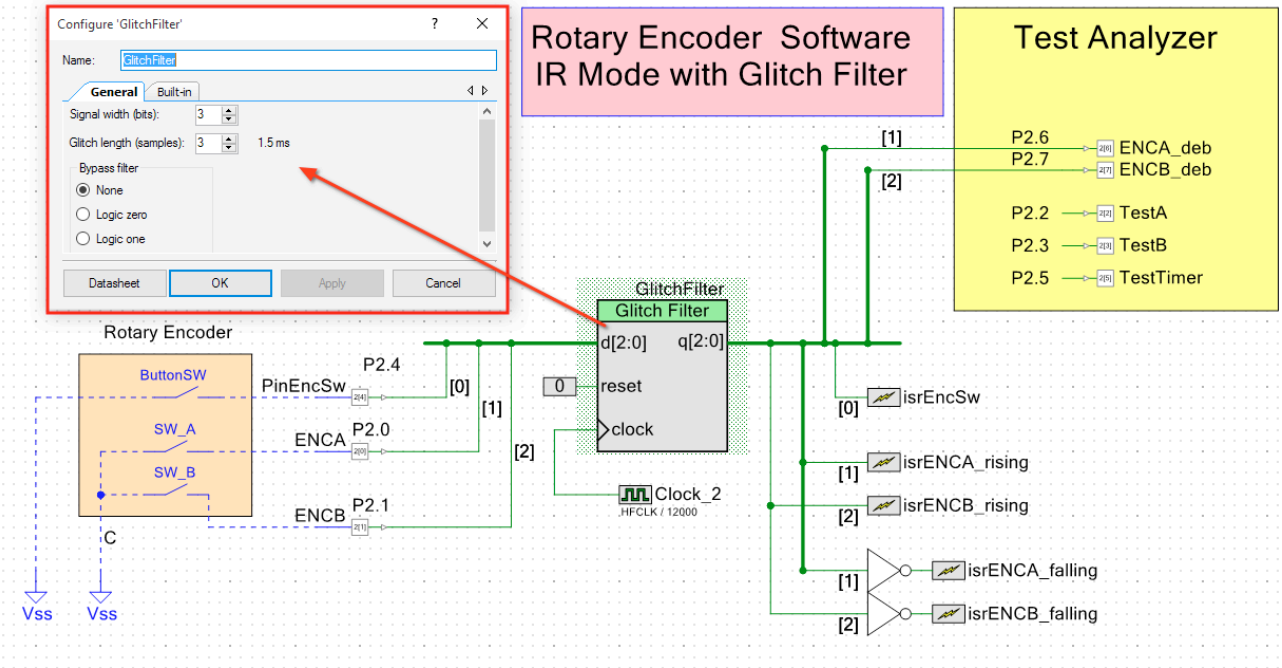


Bild 28: Encoder lesen im IR-Verfahren mit Debouncing

Zur Entprellung wird hier die Glitchfilter Komponente eingesetzt, die auch gleich die Encoder-Taste entprellt. Die **Glitch length** wurde für den extrem prellenden Encoder auf 1,5ms eingestellt. Je nach Encoder kann hier ggf. eine kürzere Zeit gewählt werden, was dann auch den Ressourcenbedarf der Glitchfilterkomponente reduziert.
Der Interrupt muss nun an den Flanken der entprellten Impulse abgenommen werden.
Da die Standard Interrupt Komponente nur RISING_EDGE als Flankeninterrupt anbietet, werden hier 2 Komponenten pro Flanke eingesetzt, wovon die eine negiert wird. Der da eingestellt RISING_EDGE-IR triggert also tatsächlich auf FALLING_EDGE.
Da die ISR nicht mehr zwischen FALLING- und RISING-EDGE unterscheiden muss, bedienen beide IR-Komponenten die gleiche ISR.

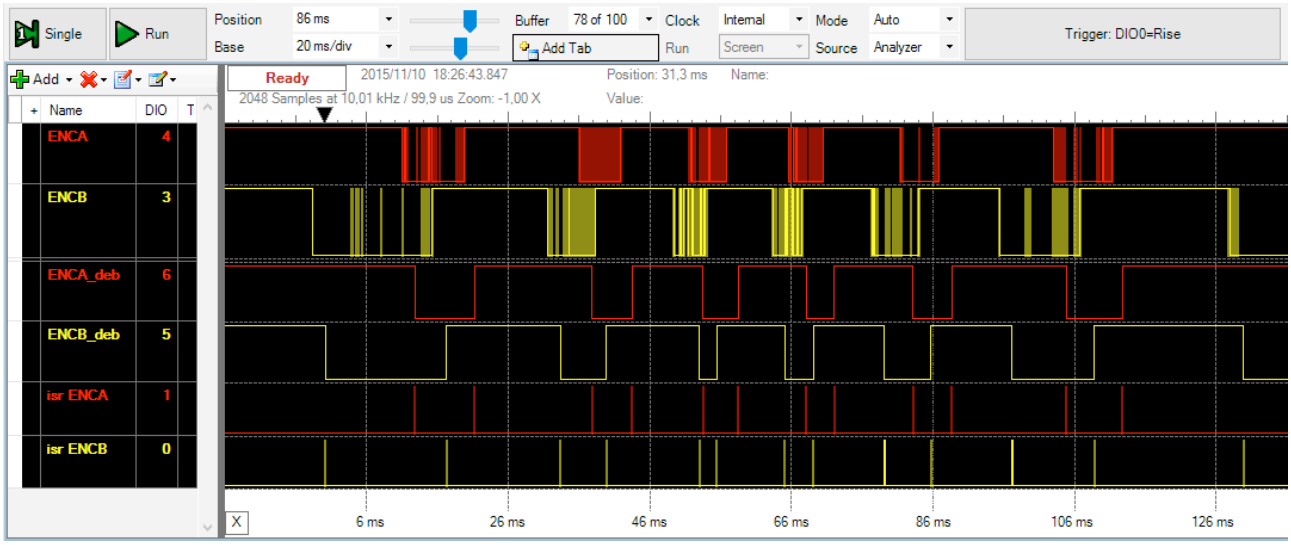


Bild 29: Oszillogramm nach Bild 28

Man erkennt deutlich, dass der Encoder perfekt entprellt wird. Es entstehen bei jeder Flanke die entsprechenden Interrupts.
Der Encoder arbeitet im Testprogramm einwandfrei. Für sehr schnelle Drehungen muss die Glitchlänge aber reduziert werden. Der Encoder arbeitete auch noch mit einer Glitchlänge von 500us perfekt, wobei dann wesentlich schnellere Drehungen möglich sind.

```
//isr_ENCA
CY_ISR(isr_ENCA){
```

```
   read_encoder();     //inline function for read and decode encoder
}

//isr_ENCB
CY_ISR(isr_ENCB){
   read_encoder();     //inline function for read and decode encoder
}
```

Die read_encoder() – Funktion ist identisch, mit der im vorherigen Beispiel.

Hinweis
Setzt man einen Encoder ein, bei dem pulses und detents gleich sind, kommt man mit je eine RISING_EDGE IR pro Kanal aus. Wie in Bild 27 zu sehen, kommt pro pulse immer eine steigende Flanke pro Kanal vor. Und da man nur einen count pro pulse benötigt reicht diese Flanke.

5.3 Kombination von Polling- und IR-Verfahren

Vergleicht man beide Softwarelösungen, stellt man fest, dass beide Varianten ihre Vor- und Nachteile haben. Die wichtigsten sind:

Polling
+ i.d.R keine Entprellung nötig
+ leicht erweiterbar für Multiplexbetrieb mehrerer Encoder (nächster Abschnitt)
+ Verfahrensbedingte Korrektur verbliebener Spikes durch den Gray-Code
- Konstante Prozessorbelastung, auch wenn keine Betätigung erfolgt

IR-Verfahren
+ Keine Prozessorbelastung wenn keine Betätigung erfolgt
- externe Entprellung nötig

Da liegt es nahe, beide Verfahren so zu kombinieren, dass die Vorteile beider Verfahren vereint werden können.
Das sollte auch gar nicht so schwer sein. Dazu kann das Polling Verfahren benutzt werden, welches aber durch einen Interrupt an den Encodereingängen aktiviert wird.
Wird der Encoder einige Zeit nicht mehr benutzt, kann der Polling Timer wieder gestoppt werden.
Das sollte sich einfach realisieren lassen. Der Polling Timer kann einen entsprechenden Zähler (TimeoutCounter) inkrementieren. Beim Timerablauf kann nun geprüft werden, ob eine gültige Statusänderung vorliegt.
Falls ja, wird der TimoutCounter auf 0 gesetzt und falls nicht, wird der Polling Timer gestoppt, wenn der PollingCounter eine voreingestellte Größe überschreitet.
Die Prozessorlast ist zwar geringfügig höher als beim reinen IR-Verfahren, dafür entfällt aber das unnötige Polling wenn der Encoder nicht benutzt wird.
Da der IR der Encodereingänge nur dazu benutzt wird, den PollingTimer zu starten, können die Eingänge diesmal im gleichen Port liegen. Und es ist auch nur eine gemeinsame ISR für die Encoder Eingänge nötig.
Das Beispielprojekt Projekt **049_RotaryEncoder_SW_IR_Polling_1** demonstriert die Verfahrensweise.

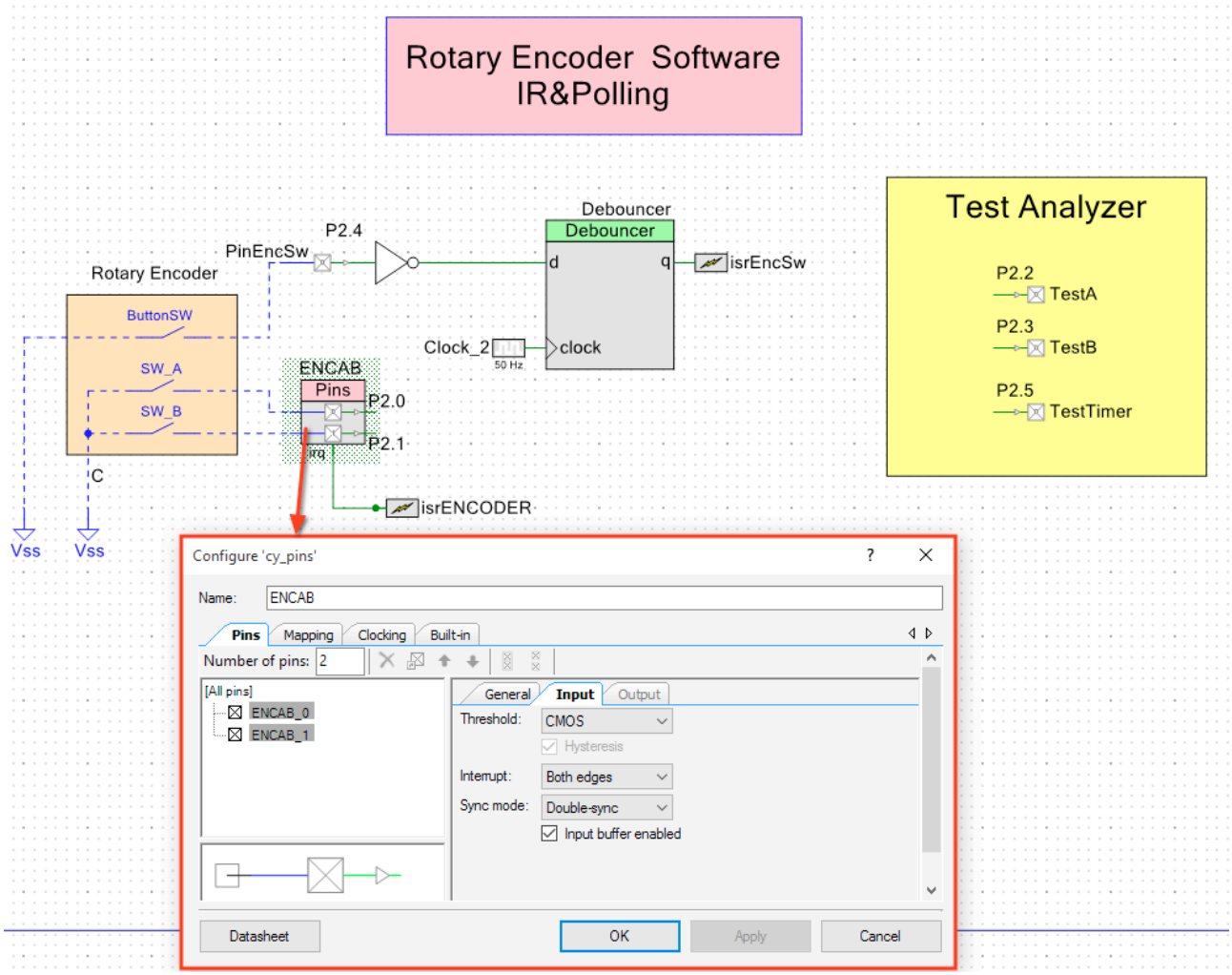


Bild 30: Schaltung für die Kombination von Encoder IR und Polling

Die Eingangs-Pins werden diesmal als Pinblock realisiert. Das hat den Vorteil, dass nur eine Interruptkomponente für beide Eingänge nötig ist.
Innerhalb der ISR muss auch nicht ausgewertet werden, welcher Pin/Flanke den IR verursacht hat. Es genügt zu wissen, dass ein IR auftrat um den Polling Timer zu starten.

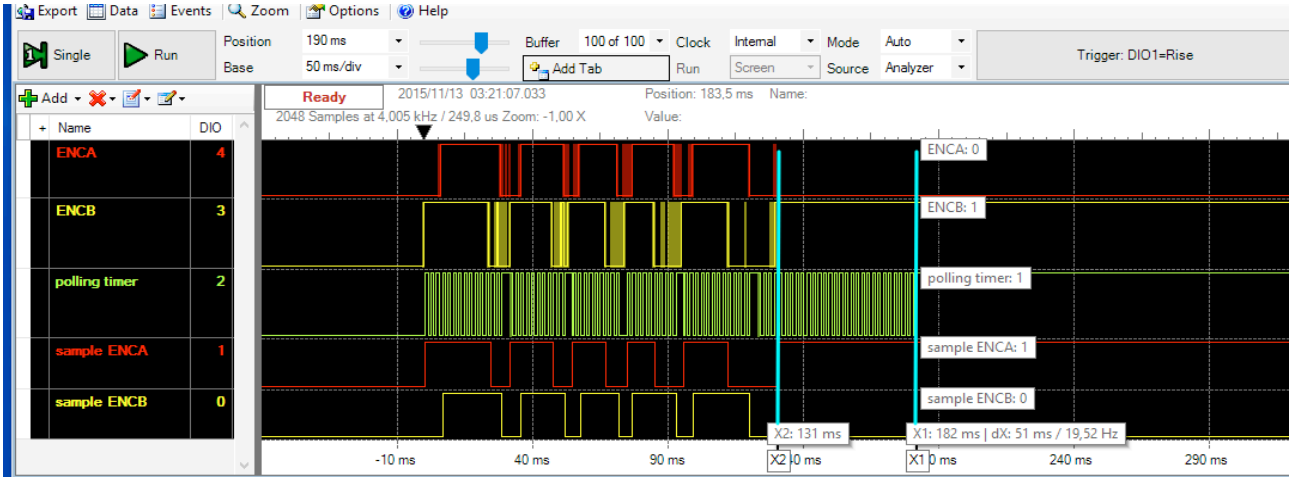


Bild 31: Oszillogramm der Kombination von Encoder IR und Polling

Im Bild 31 ist gut zu erkennen, dass der Polling Timer mit der ersten Flanke startet und 50ms nach der letzten Flanke beendet wird.
Eine Entprellung des Encoders ist jetzt i.d.R. nicht mehr nötig.

Hinweis
Die Timeout Zeit kann ohne weiteres weiter verringert werden. Selbst wenn sie kleiner als die Zeit von Raste zu Raste, ist funktioniert das Verfahren weiter korrekt, weil der Pollingtimer ja mit der nächsten Flanke sofort wieder gestartet wird.

Implementierung
Die Implementierung besteht im Wesentlichen aus 3 Komponenten:

- Der Initialisierung, bei der nur der Encoder Interrupt gestartet wird, nicht aber der polling Timer.
- Der Encoder ISR, die den Encoder Interrupt stoppt und den polling Timer startet.
- Der eigentlichen Encoder read/decoder Routine, die diesmal um die Behandling eines TimeoutCounters ergänzt ist.

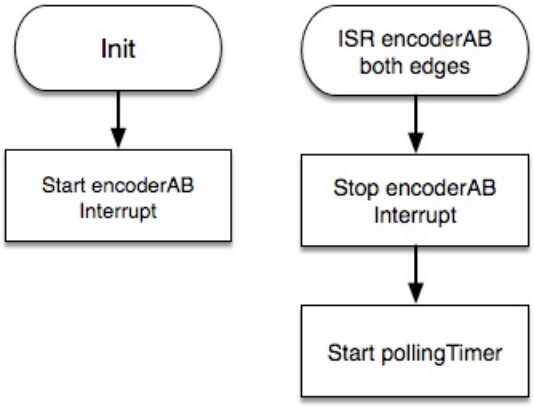


Bild 32: Initialisierung und encoder ISR beim kombinierten IR/Polling Verfahren

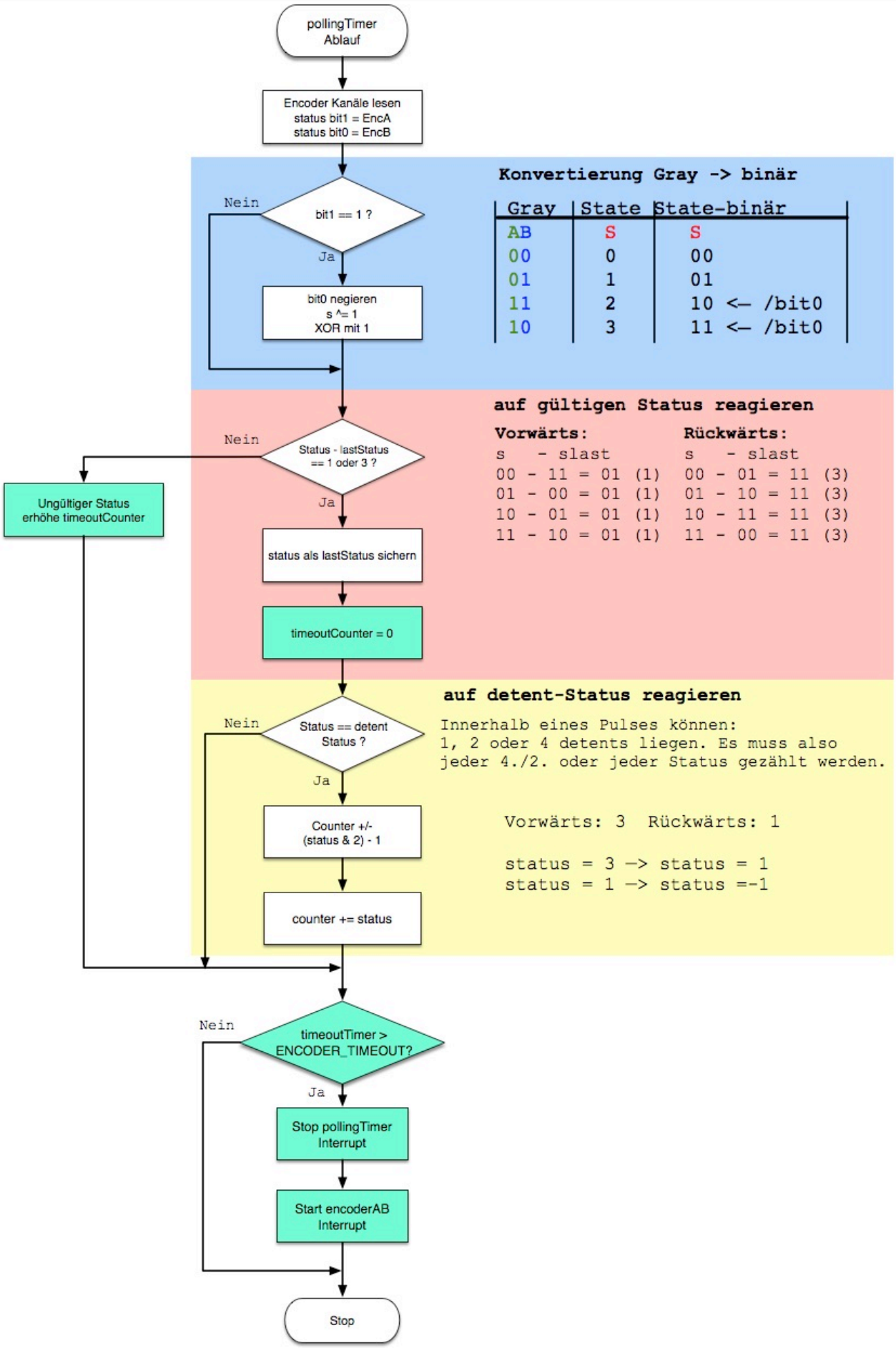


Bild 33: Polling Timer Ablauf beim kombinierten IR / Polling Verfahren

5.4 Berücksichtigung der Drehgeschwindigkeit (Dynamik)

Drehregler besitzen i.d.R. < 50 Rasten. Berücksichtigt man weiter, dass man ohne nachzufassen ca. eine halbe Umdrehung hinbekommt, bekommt man mit einem Handgriff eine Counteränderung von weniger als 25 hin.
Muss man sehr große Zahlen genau eingeben können, dann muss man also entsprechend lange drehen.
Da wäre es wünschenswert, wenn sich die Counteränderung mit zunehmender Drehgeschwindigkeit erhöhen ließe.
Dazu ist es nötig, den Faktor Zeit zu integrieren, um die Geschwindigkeit ermitteln zu können.

5.4.1 Dynamik mit Zeitmessung zwischen den Rasten

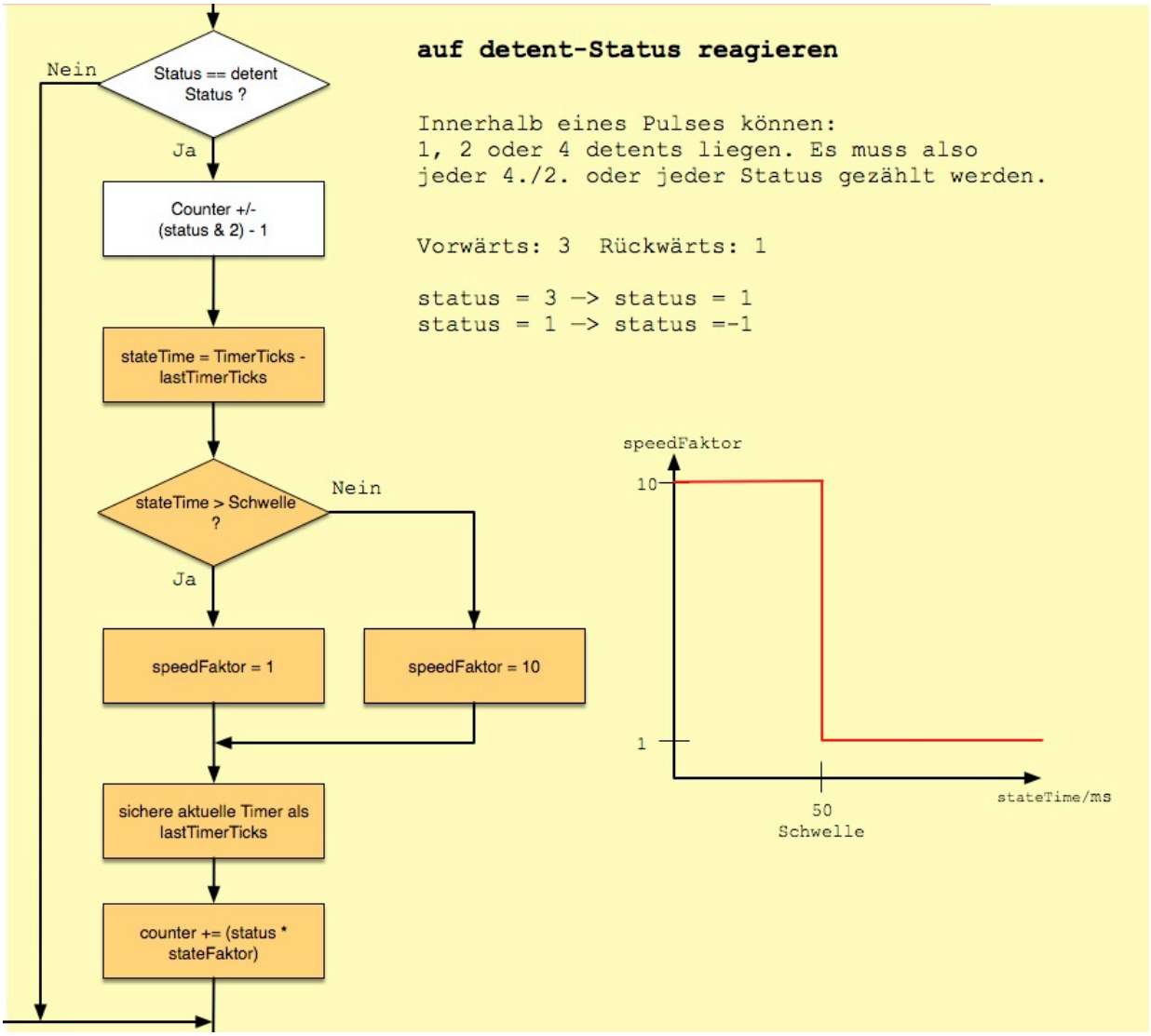
Projekt **049_RotaryEncoder_SW_Dynamik_1**
Bei diesem Verfahren wird die Zeit zwischen 2 Stati (oder besser noch zwischen 2 Rasten) gemessen. Das sollte sich leicht realisieren lassen, indem die Ticks eines Timers zwischen 2 Stati/Rasten gezählt werden.
Die Auswertung kann nun so erfolgen, dass die gemessene Zeit mit fest hinterlegten Zeiten verglichen wird und in Abhängigkeit dieser Schwellen ein Faktor für den Encoder Counter benutzt wird.
Dabei ist allerdings zu beachten, dass dieser Faktor eine Beziehung zu den detents des Encoders hat. Denn je mehr Rasten der Encoder hat, desto größer ist ja die Zahl der Counts, die bei gleichem Drehwinkel erreicht werden können.
Statt fester Schaltschwellen ist es natürlich auch möglich, die Abhängigkeit als Funktion der Drehgeschwindigkeit zu implementieren.
Die beschriebene Softwarelösung nach dem Polling- oder dem kombiniertem Verfahren bietet sich geradezu dafür an, da der Polling Timer ja schon zur Verfügung steht.
Da muss also lediglich ein Timerwert gespeichert werden.
Bei der Inkrementierung/Dekrementierung des Counterwertes muss dann auf die gemessene Zeit reagiert werden.
Wurde der Einfluss der Drehgeschwindigkeit als Funktion implementiert, ergibt sich die Änderung des Encoder-Counters unmittelbar aus der Funktion. Je nach Funktion ist hier natürlich ggf. Rechenaufwand nötig.
Arbeitet man hingegen mit Geschwindigkeitsschwellen reicht da eine Vergleichsstruktur.
Bei Drehreglern mit so wenig Rasten reicht ein Schwelle aus. Bei mehr Schwellen fehlt das Gefühl, wie schnell der Drehregler gedreht werden muss um einen entsprechenden Faktor zu erwischen.
Bei hochauflösenden Encodern mag das anders aussehen.
Ich habe im Beispiel eine Dynamik mit einer Schwelle implementiert. Dabei wird der Counter in 10er Schritten gezählt, sobald der zeitliche Abstand zwischen 2 Rasten < 50ms wird.
Damit lässt sich hinreichend präzise arbeiten. Die Zeit habe ich empirisch bei einem Encoder mit 20 detents ermittelt.
Die 50ms entsprechen also einem detent. Für den Vollkreis gilt dann: 50ms * 20 = 1000ms
Es macht Sinn, die Zeit normiert auf den Vollkreis zu verwenden, da diese Zeit im Programm leicht für Encoder unterschiedlicher Rastenzahl angepasst werden kann.

```
#define ENCODER_DYNAMIK_TRESHOLD 1000 // ms. per revolution
```

Bei einer SysTick Auflösung von 250us werden bei 1000ms genau 1000ms / 20detent *4 = 200 Ticks gezählt.
Allgemein formuliert beträgt die Schwelle in Ticks:

```
/* SysTimers Resulution = 250us = 1ms/4 */
#define ENCODER_DYNAMIK_TICKS ( 4 * ENCODER_DYNAMIK_TRESHOLD
                                / NUMBER_OF_DETENT)
```

Wie in Bild 34 zu sehen, können alle erforderlichen Programmerweiterungen direkt zusammenhängend an der Stelle vorgenommen werden, wo der Encoder Counter weitergezählt wird.
Da bietet es sich an, diese Funktionalität in eine if-Struktur zu legen, womit die Dynamik dann einfach ein- und ausgeschalten werden kann.



Die Implementierung der relevanten Teile im Quellcode befindet sich dann komplett in einer if-Abfrage.

```
/*only the right state*/
if (0 == myEncoder.sLast % ENCODER_MODE) {
     uint8 speedFaktor = 1; // default per 1
     if ( encoderDynamik ) {
        uint32 timerTicks = SysTimers_GetSysTickValue();  //aktual SysTicks
        uint32 stateTimeDiff = timerTicks - myEncoder.lastTimerTicks;

        /*if time between 2 detents < ENCODER_DYNAMIK_TRESHOLD*/
        if (stateTimeDiff < ENCODER_DYNAMIK_TICKS) {
            speedFaktor = 10;
        }

         myEncoder.lastTimerTicks = timerTicks; //save last SysTicksValue
     }
     myEncoder.counterValue += speedFaktor *((s & 2) - 1);
     myEncoder.direction = ((s & 2) >> 1) ? '+':'-';  //bit1 = encoder direction
     retVal = 1;                                      //counterValue has changed
}
```

Nach dieser Methode lässt sich die Dynamik sehr einfach genau dosieren und problemlos an encoder mit anderer Rastenzahl anpassen.

Hinweis
Wird der Speedfaktor als Exponent einer 2er Potenz definiert, kann man statt der Multiplikation schneller shiften

```
myEncoder.counterValue += ((s & 2) - 1) << speedFaktor;
```

5.4.2 Dynamik nach Lothar Miller

Eine etwas andere Art, Dynamik zu integrieren findet sich hier:
http://www.lothar-miller.de/s9y/categories/54-Encoder

Lothar Miller setzt hier ebenfalls die Decoder Routine nach Peter Dannegger ein. Die Dynamik wird aber anders erzeugt.
Die Lösung von Lothar Miller ist äußerst clever. Sie kommt mit sehr geringen Programmieraufwand und man kann tatsächlich von einem gewissen „haptischen Gefühl“ sprechen, wie Miller dies auf seiner Homepage beschreibt.
Prinzipiell ist hier der Counter-Multiplikator abhängig vom Abstand der einzelnen Stati (oder auch Rasten), weshalb man tatsächlich von einem Beschleunigungseffekt sprechen kann

Zur Arbeitsweise:
Das Verfahren ist durch 3 Konstanten in seiner Verhaltensweise steuerbar.

```
#define DYNAMIK 50      //Wert, mit dem die Beschleunigung mit jeder Raste max.
                        //erhöht/verringert wird
#define MIN_ACCEL 6     //minimaler Beschleunigungsfaktor als Potenz von 2
                        //2^MINIMUM_ACCEL = 6
#define MAX_ACCEL 255   //maximaler Beschleunigungsfaktor
```

Bild 35 zeigt Ablaufplan für die Beschleunigung. Das Verfahren besteht aus 2 Teilen.
Die Entschleunigung vermindert die Beschleunigung direkt durch den Polling Timer, also entsprechend häufiger als der Status gewechselt wird.

Dies erfüllt 2 Aufgaben.
- Die Beschleunigung wird schnell wieder auf 0 gesetzt (zurückgezählt) wenn der Encoder nicht mehr gedreht wird.
- Innerhalb einer Encoder Bewegung wird der aktuelle Beschleunigungswert umso stärker vermindert, je langsamer der Encoder gedreht wird. Je länger die Zeit bis zur nächsten Raste, desto mehr Pollingimpulse vermindern die Beschleunigung.

Die Beschleunigung wird bei der Rastenerkennung erhöht und ausgewertet.
Sofern der maximale Beschleunigungsfaktor noch nicht erreicht ist, wird er um die DYNAMIK erhöht.
Ist der aktuelle Beschleunigungsfaktor kleiner als der minimale Beschleunigungsfaktor, wird der der Encoder um den Wert 1 erhöht/verringert (Standard ohne Beschleunigung).
Andernfalls erfolgt die Encoder Änderung um die aktuelle Beschleunigung dividiert durch die Minimale Beschleunigung.

Hinweise zur Dimensionierung der Konstanten

MAX_ACCEL bestimmt die höchste erreichbare Encoder Counter-Änderung pro Schritt. Größere Werte würden nur Sinn machen, wenn sehr große Encoder-Werte mit kleinem Drehwinkel machbar sein sollen.

DYNAMIK wird man empirisch ermittelt. Wie im vorangegangenen Beispiel könnte es Sinn machen, den Wert auf eine volle Encoderumdrehung zu beziehen. Dann lässt er sich leicht auf Encoder mit anderer Rastenzahl umrechnen.
Der obige Wert für DYNAMIK bezieht sich auf einen Encoder mit 32 detent. Für den Vollkreis gilt dann:

DYNAMIK normiert auf einen Vollkreis = 50 * 32 = 1600

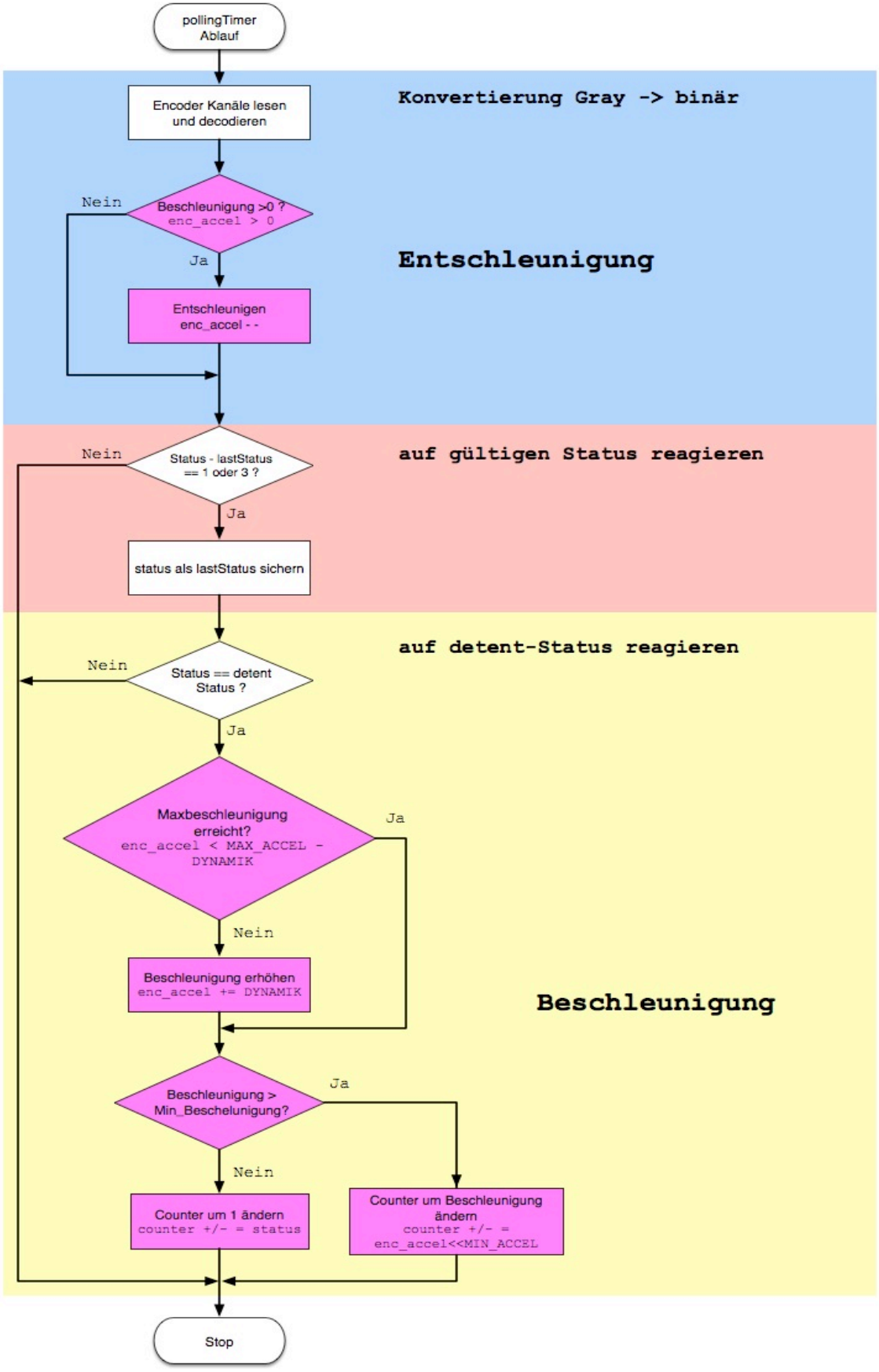
```
#define NUMBER_OF_DETENT 32          //32 Rasten
#define DYNAMIK_PER_REVOLUTION  1600  //Dynamik (50) normiert auf 360°

//Dynamik für je nach Rastenanzahl des Encoders
#define DYNAMIK (DYNAMIK_PER_REVOLUTION  / NUMBER_OF_DETENT)
```

Implementierung im Beispiel Projekt **049_RotaryEncoder_SW_Dynamik_LotharMiller**.

Bild 35: Pollingverfahren mit Beschleuniging nach Lothar Miller

Prinzipiell kann dieses Verfahren natürlich im reinen Pollingverfahren, sowie in der Kombination IR/Polling benutzt werden.
Die Implementierung ist komplett in der Funktion **encoder_update()** realisiert.




```
/*encoder hardware*/
```

```
#define NUMBER_OF_DETENT  32   //32 für Ppansonic EVEQ, 20 für meinen Alps
#define NUMBER_OF_PULSE   16   //16 für Ppansonic EVEQ, 20 für meinen Alps
#define DYNAMIK_PER_REVOLUTION  1600  //Dynamik (50) normiert auf 360°

//Anzahl der Stati pro Raste
#define ENCODER_MODE ((4 * NUMBER_OF_PULSE / NUMBER_OF_DETENT))

//Dynamik auf eine Raste bezogen
#define DYNAMIK (DYNAMIK_PER_REVOLUTION  / NUMBER_OF_DETENT)

#define DYNAMIK 50      //Wert, mit dem der Encoder Counter mit jeder Raste max.
                        //erhöht/verringert wird
#define MIN_ACCEL 6     //minimaler Beschleunigungsfaktor als Potenz von 2
                        //2^MINIMUM_ACCEL = 6
#define MAX_ACCEL 255   //maximaler Beschleunigungsfaktor
```

...

```
 /*Polling Timer*/
 if(SysTimers_GetTimerStatus(encScanTimer)) {

    uint8 encA = ENCA_Read();      //encoder Spur A
    uint8 encB = ENCB_Read();      //encoder Spur B
    uint8 s = encB;                //aktueller status

    if (encA) {s ^= 3;}             //Gray in binär konvertieren
    s -=  myEncoder.sLast;          //letzen Status holen

    // solange >0, pro PollImpuls um 1 "entschleunigen"
    if ((encoderDynamik)&&(myEncoder.enc_accel)) myEncoder.enc_accel--;

    if (s & 1) {                   //wenn bit0 gesetzt (diff ist 1 or 3)
       myEncoder.sLast += s;       //s is now s-sLast -> sLast + s - sLast = s

        //falls Status ein „Rasten-Status"
       if (0 == myEncoder.sLast % myEncoder.encoderMode ) {

          /*neuen Beschleunigungswert berechnen
            solange die Beschleunigung kleiner als  MAX_ACCEL - DYNAMIK
            Beschleunigungswert aufaddieren*/
          if ((encoderDynamik) && (myEncoder.enc_accel<255-DYNAMIK))
             myEncoder.enc_accel += DYNAMIK;

          if (s & 2) {             //Rechtsdrehung (bit1 = 1)
             myEncoder.direction = '+';

            /*Erst Beschleunigungswerte > DYNAMIK sonst währe ggf.
              eine Zählung um 1 nicht mehr möglich*/
             myEncoder.counterValue += 1 + ((myEncoder.enc_accel > DYNAMIK)
                                        ? myEncoder.enc_accel >> 6 : 0);
          } else {                 //Linksdrehung  (bit1 = 0)
             myEncoder.direction = '-';
             myEncoder.counterValue -= 1 + ((myEncoder.enc_accel > DYNAMIK)
                                        ? myEncoder.enc_accel >> 6 : 0);
          }
       }
    }
 }
```

Das Verfahren arbeitet hervorragend. Nachteilig ist höchstens, dass es mehrere Parameter gibt, die die Beschleunigung beeinflussen, sodass evtl. etwas Probieren angesagt ist.
Bild 36 zeigt das Oszillogramm der Beschleunigung in Abhängigkeit der Encoderdrehung. Dazu wurde der Beschleunigungswert in der ISR einfach benutzt um die Spannung mit einem IDAC zu erzeugen.
Projekt **049_RotaryEncoder_SW_Dynamik_LotharMiller_B**.

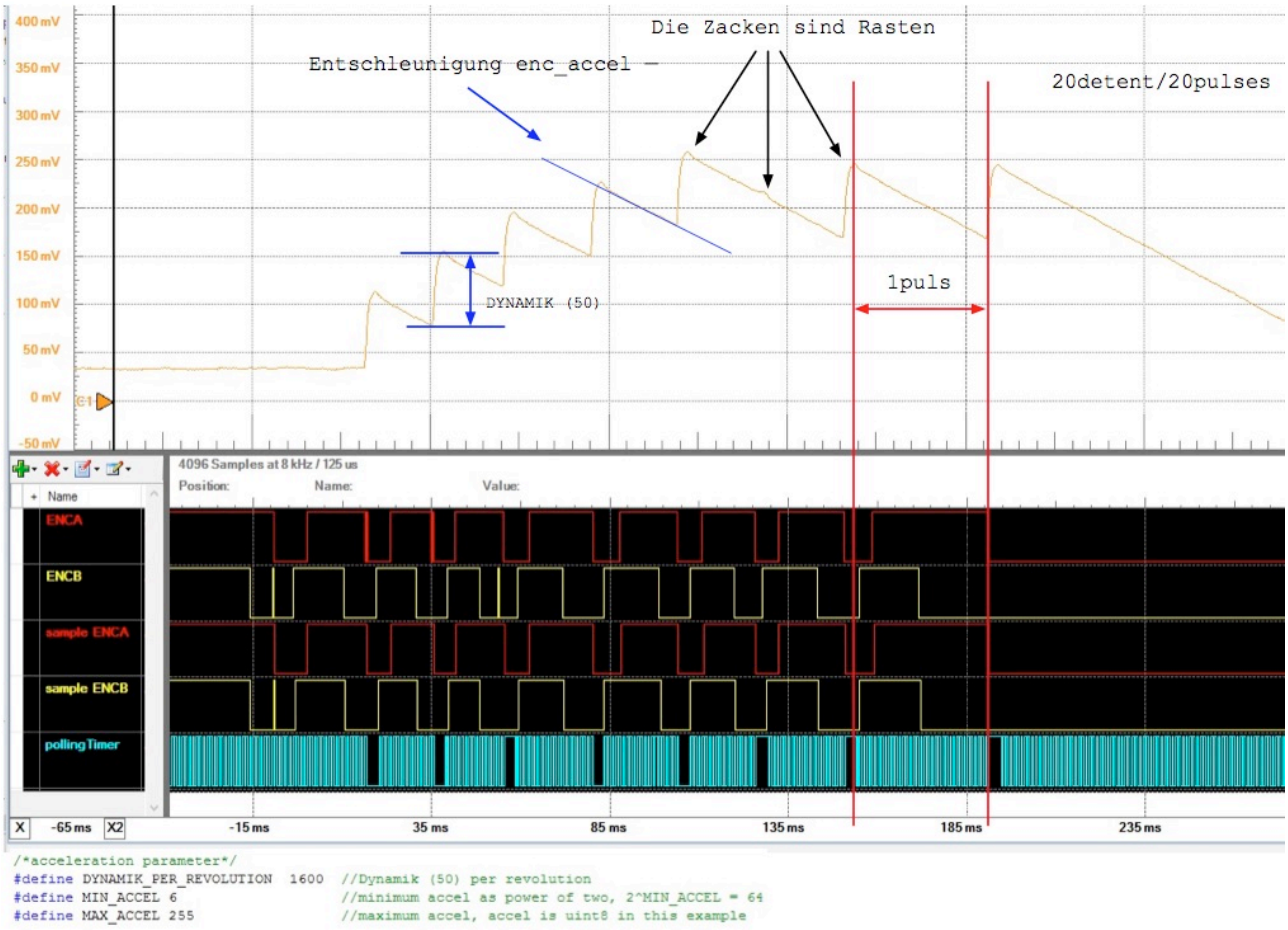


Bild 36: Oszillogramm des Beschleunigungswertes

Man erkennt sehr gut die Wirkung der einzelnen Parameter.
Der Beschleunigungswert wächst mit jeder Raste um den Wert DYNAMIK. Je größer allerdings der Abstand zwischen den Rasten (je langsamer der Encoder gedreht wird), desto mehr wird er wieder entschleunigt. Höher als 255 (MAX_ACCEL) kann der Beschleunigungswert nicht ansteigen. Bei sehr feinauflösenden Encodern, mit denen ein großer Wertebereich überschritten werden soll, kann es Sinn machen, für den Beschleunigungswert **`uint16`** einzusetzten.
Bei encodern mit vielen Rasten ist aber die Zeit zwischen den Rasten recht kurz und endsprechend wenige Polling Impulse liegen zwischen den Rasten. Das hat dann zur Folge, dass die Entschleunigung zwischen den Rasten kleiner wird.
Da kann man aber den Entschleunigungsfaktor erhöhen:

```
#define DECELERATE 2 //Entschleunigungsfaktor
if (enc_accel) enc_accel -= DECELERATE; // until enc_accel>0 "decelerate" with 2
```

Das hat außerdem den Vorteil, dass der Beschleunigungswert schneller wieder auf 0 ist, wenn der Encoder nicht mehr gedreht wird.

5.5 Multiplexbetrieb mehrerer Encoder

Project **RotaryEncoder_1_Software_Mulitplex**
Die Softwarerealisierung (Polling- oder auch die Kombination aus Polling und IR) lässt sich wunderbar für den Einsatz mehrere Encoder im Multiplexbetrieb verwenden.
Sollen mehrere Encoder an einem PSoC Chip verwendet werden, so ist die Hardware-Lösung mittels der Quad-Decoder Komponenten schnell am Ende, da nur eine begrenzte Zahl der Hardwarekomponenten zur Verfügung steht. Außerdem benötigt man 2 Leitungen pro Encoder (ohne Berücksichtigung der Schalterfunktion).
Mit der Software-Version nach Abschnitt 5.1 (Polling) ist das kein Problem. Dabei können die Encoder im Multiplexbetrieb abgefragt werden.

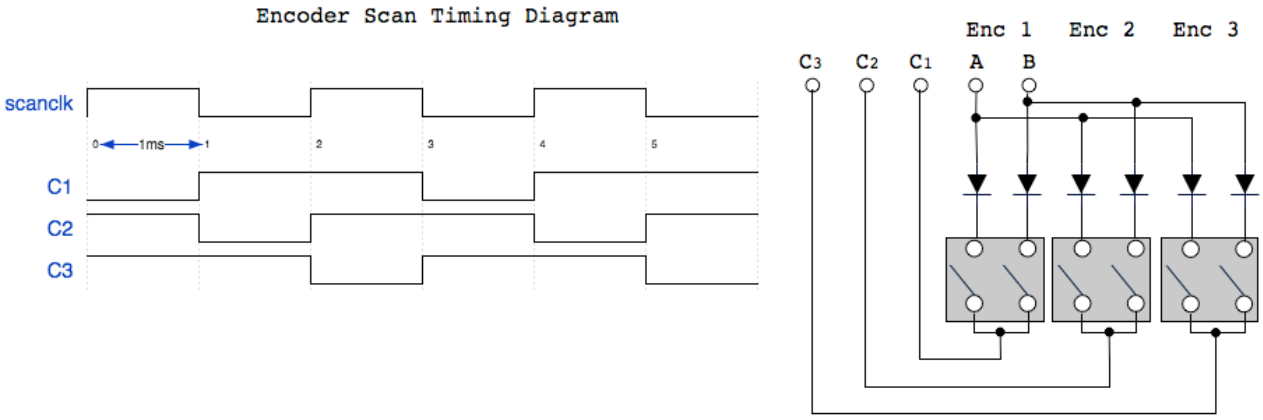


Bild 37: Prinzipschaltung und Timing Dssiagram für eine Multiplaxabfrage

Bild 37 zeigt die prinzipielle Verfahrensweise.
Die Terminalanschlüsse C der einzelnen Encoder werden nacheinander auf 0 geschalten (hier für 1 ms).
Die Anschlüsse A und B der Encoder werden über Entkopplungsdioden parallel geschalten.
Die Pegel an den Eingängen A/B können nur durch den Encoder auf 0 gezogen werden, dessen Terminalanschluss C auf 0 liegt.

Hinweis
Die Entkopplungsdioden sind nötig, da die Anschlüsse A und B andernfalls durch JEDEN Encoder kurzgeschlossen werden könnten, falls dessen Schalter A und B gerade geschlossen sind.

Wie man der Schaltung entnehmen kann, benötigt man pro Encoder eine Leitung und zusätzlich 2 Leitungen für die gemeinsamen A/B Anschlüsse.

$$Anz_{GPIO} = n + 2 \quad | \, für \; n = Anzahl \; der \; Encoder$$

Bei der direkten Ankopplung werden pro Encoder 2 GPIO Anschlüsse benötigt. Bereits bei 3 Encoder wird ein Anschluss gespart.
Die Anzahl der benötigten GPIOs lässt sich weiter reduzieren, wenn die Terminalanschlüsse C über einen externen BCD Decoder angeschlossen werden.
Die Software aus Abschnitt 5.1 muss lediglich um einen entsprechenden Softwarezähler erweitert werden, welcher die GPIOs für die einzelnen Encoder gemäß Bild 37 ansteuert.

Evtl. vorhandene Encoder-Taster werden beim Multiplexbetrieb nicht berücksichtigt. Sollen die ebenfalls Verwendung finden, so sind diese wie normale Taster zu implementieren. Natürlich kann man die ebenfalls im Multiplexbetrieb anschließen um GPIOs zu sparen. Etwa wie dies beim Anschluss von Keypads gemacht wird.
In dieser Beschreibung gehe ich allerdings nicht weiter darauf ein.

Ein kleines Beispiel soll das Verfahren demonstrieren. Dazu werden 3 Encoder simultan betrieben.
Die relevanten Daten der einzelnen Encoder werden jeweils auf einer zugehörigen LCD-Zeile ausgegeben.
Der Anschluss der Encoder erfolgt entsprechend Bild 37.

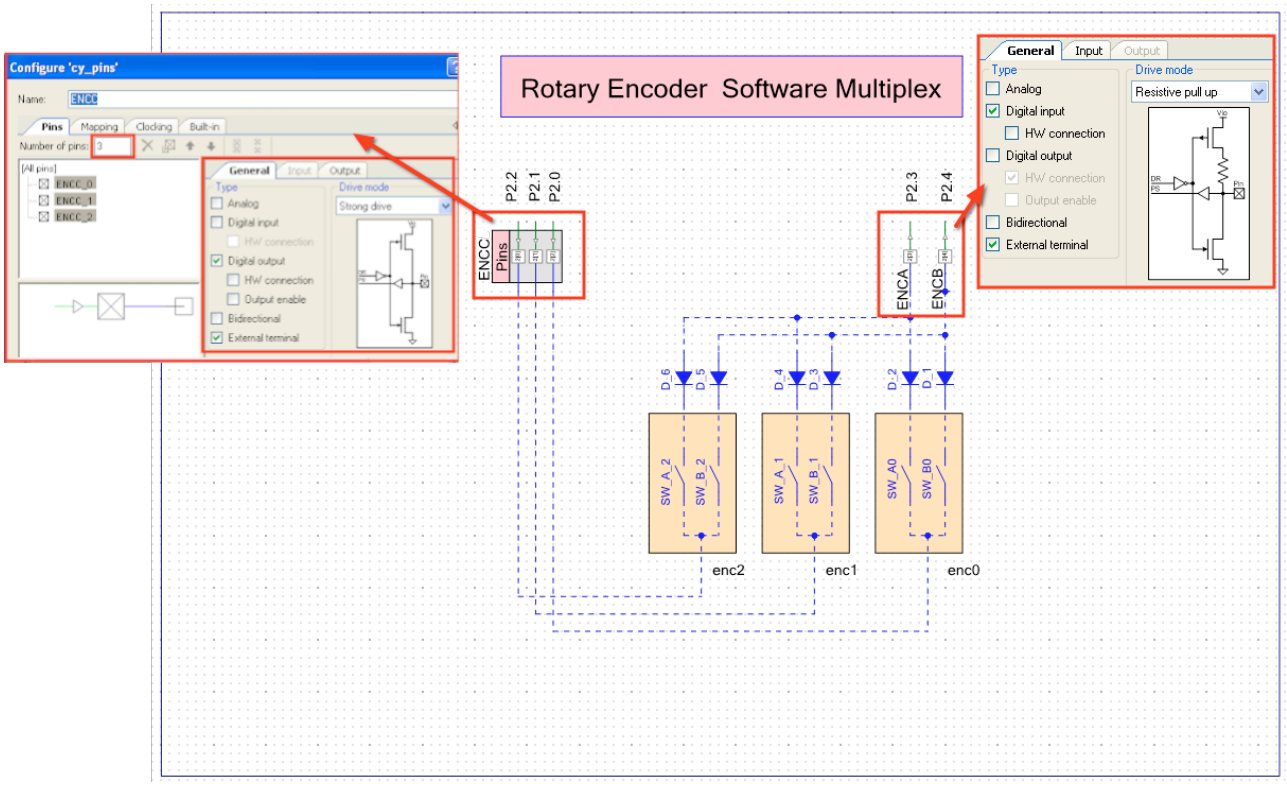


Bild 38: Anschluss der Encoder und Konfiguration der GPIOs

Hilfreich ist hier die Zusammenfassung der 3 Terminal C GPIOs als Pinblock mit 3 Pins.
Vorteilhaft ist hier, dass die Pins dabei gemeinsam mit einem Byte beschrieben werden können. Mit **ENCC_Write(7)** können z.B. alle 3 Pins auf 1 gesetzt werden.
Nachteilig ist, dass die Pins in dieser Betriebsweise in einem Block beginnend bei Pin0 fortlaufend angeordnet sein müssen.
Ist dies nicht möglich, können natürlich auch einzelne Pins wie bei ENCA / ENCB benutzt werden. Die Programmierung ist dann etwas umständlicher.

Die einzelnen Encoder werden als struct verwaltet und in einem Array zusammengefasst.

```
/*encoder hardware*/
#define INITIAL_COUNTER  0x8000     //InitialCounter

/*individual encoder values*/
#define MAX_ENCODERS 3              //1...8

#define ENC0_NUMBER_OF_DETENT  20     //for my Alps
#define ENC0_NUMBER_OF_PULSE   20     //for my Alps
#define ENC0_ENCODER_MODE (4 * ENC0_NUMBER_OF_PULSE / ENC0_NUMBER_OF_DETENT )
```

... weitere Encoder. Maximalanzahl ist 8 in diesem Beispiel. Ließe sich aber einfach erweitern.

```
/*--------rotary encoder struct----------*/
typedef struct encoder{
    uint8 encoderMode;
    char  direction;          //encoder direction (if needed)
    uint8 sLast;              //last state
    uint16 counterValue;      //actual counter
    uint16 lastCounterValue; //last counter
} ENCODER_t;

/*--------array with all encoder structs-*/
extern ENCODER_t encoders[];
```

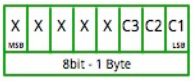
Die Funktion zum Auslesen des Encoder ist fast identisch mit der einfachen Software Version für einen Encoder. Der Unterschied besteht eigentlich nur darin, dass vor dem Auslesen des Encoders mit jedem Scanimpuls der nächste Encoder im Array aktiviert wird.
Die Encoder werden also nacheinander abgetastet, was natürlich die Anzahl der Abtastungen pro Encoder entsprechend verringert. Das Beispiel arbeitet mit einem Abtastimpuls von 1ms bei 3 Encodern mit unterschiedlichen Parametern. Dabei kann ich keine Verschlechterung gegenüber der Standardversion mit einem Encoder erkennen. Die Encoder reagieren flüssig und genau, **auch wenn 2 Encoder gleichzeitig gedreht werden**.

Das Durchschalten der Scanimpulse auf den Terminal C Anschluss des Encoders geschieht mit Ablauf des Scantimers.

```
if(SysTimers_GetTimerStatus(encScanTimer)) { //scan timer expired?
    /*-- rotate Terminal ENCC --*/
    if (++encoderIdx >= MAX_ENCODERS){ //rotate between 0 and MAX_ENCODERS-1
          encoderIdx = 0;
    }
    ENCC_Write(~(1 << encoderIdx));     //write LOW to next encoder terminal

    ...
```

Der encoderIdx (Index für das Array encoders[]) wird erst incrementiert und falls die max. Anzahl überschritten ist, wieder auf 0 zurückgestellt.
Anschließend wird der entsprechende Terminal C-Pin auf LOW gesetzt und die restlichen Pins auf HIGH. Da ein Pinblock mit 3 Pins benutzt wird, können die Pins gemeinsam gesetzt werden.




```
Soll C2 aktiviert werden(encoderIdx=1) bedeutet das:
ENCC_Write(0b101) = ENCC_Write(5)

  010        // 1 << 1(encoderIdx)
~ 010 = 101  // bitweise negation
```

Bild 39: Aktivieren des Terminal C Anschlusses des betreffenden Encoders

Bei der eigentlichen Dekodierung des Encoders muss lediglich darauf geachtet werden, den entsprechenden Encoder zu adressieren.

```
encoders[encoderIdx].lastCounterValue = encoders[encoderIdx].counterValue;
uint8 encA = ENCA_Read(); //read ENCA
uint8 encB = ENCB_Read(); //read ENCB
uint8 s = encB;           //stati

if (encA) {s ^= 3;}  //S = ENCB ^3
   s -=  encoders[encoderIdx].sLast;
   if (s & 1) {          //if bit0 set (diff is 1 or 3)
      encoders[encoderIdx].sLast += s; //s is now s-sLast -> sLast+s-sLast=s

      //only the right state
      if (0==encoders[encoderIdx].sLast % encoders[encoderIdx].encoderMode){

         //clear bit->(0 vw; 2 bw)-1
         encoders[encoderIdx].counterValue += (s&2)-1;
         encoders[encoderIdx].direction = encoders[encoderIdx].lastCounterValue<
                                          encoders[encoderIdx].counterValue) ?
                                          '+':'-'; //encoder direction
      }
   }
}
```

Im Beispiel **049_RotaryEncoder_SW_Mulitplex_1** werden die aktuellen Encoderdaten lediglich in jeweils in einer eigenen LCD Zeile angezeigt.
Die komplette Encoder - Funktionalität ist wieder in ein eigenes Modul **encoder_sw_multiple.c/.h** ausgegliedert.

6 Entscheidungshilfe welches Verfahren für welchen Zweck

In diesem Kapitel stelle ich noch einmal die besprochenen Verfahren zusammen und gebe Hinweise, unter welchen Bedingungen die Variante besonders geeignet/ungeeignet ist.
Natürlich erhebt die Aufstellung keinen Anspruch auf Vollständigkeit und im konkreten Fall können individuelle Gründe zu anderen Entscheidungen führen.

Realisierungs Variante	**Spricht für + / gegen - den Einsatz**
Hardware TCPMW Komponente ohne Entprellung	+ geringste Prozessorbelastung + Komponenten stehen in PSoC 4/4M in unterschiedlicher Zahl zur Verfügung - u.U. keine extremen Zählgenauigkeit bei stark prellenden Encodern (evtl weniger geeignet für extrem prellende Encoder) - maximale Anzahl durch Anzahl der TCPWM Komponenten begrenzt
Hardware TCPMW Komponente mit Entprellung über Debouncer	+ geringste Prozessorbelastung + Komponenten stehen in PSoC 4/4M in unterschiedlicher Zahl zur Verfügung + extreme Zählgenauigkeit (geeignet für extrem prellende Encoder) - maximale Anzahl durch Anzahl der TCPWM Komponenten begrenzt - höherer Hardwareressourcenverbrauch
Hardware UDB QuadDec Komponente	+ Entprellung bereits in der UDB Komponente eingebaut + extremen Zählgenauigkeit (geeignet für extrem prellende Encoder) - hoher UDB Ressourcen Verbrauch - unter PSoC 4 nur ohne die interne Komponentenentprellung nutzbar
Software Polling Verfahren	+ i.d.R. keine Entprellung nötig + sehr gut geeignet für Multiplexbetrieb mehrerer Encoder + lässt sich einfach um eine Dynamik ergänzen - erhöhte Prozessorbelastung durch permanenten Pollinginterrupt
Software IR-Verfahren	+ sehr geringe Prozessorbelastung durch IR nur wenn der Encoder gedreht wird - nur sinnvoll einsetzbar, wenn der Encoder entprellt ist/wird - Hohes Interruptaufkommen, besonders, wenn der Encoder nicht komplett prellfrei ist, macht den Vorteil der geringen Prozessorlast ggf. zunichte.
Software kombiniertes Polling/IR-Verfahren	+ i.d.R. keine Entprellung nötig + sehr gut geeignet für Multiplexbetrieb mehrerer Encoder + lässt sich einfach um eine Dynamik ergänzen + geringe Prozessorbelastung durch Polling-IR nur wenn der Encoder gedreht wird