Andreas Axelsson
Alexander Göransson
Viktor Mann

JÖNKÖPING UNIVERSITY
School of Engineering

2023-05-03

Enchipsdatorer – Laboration 2
Trafikljus

Målsättning

- Konstruera ett tillståndsdiagram (Mealy) och implementera dess beteende i kod.
- Skriva event-driven kod.
- Impelementera interrupt-driven funktionalitet.
- Få dioder att blinka.

Förberedelse

Läst på föreläsningen om tillståndsmaskiner och den om interrupts. Läs igenom kapitel 1 och 2 i detta dokument och gör sedan diagrammen i för 3.1 och 3.2.

Examination

Resultat för laborationen presenteras muntligt för en laborationshandledare under laborationstillfället. Svar på frågor i **fet stil** ska besvaras under redovisningen. Krav på kodstil måste följas.

Genomförande

Laborationen har 4 timmar handledd tid, men kan ta mer tid att genomföra, vilket då görs på egen hand.

Det är viktigt att kodningsarbetet sker individuellt. Det är okej att diskutera problemlösning med andra men det är absolut förbjudet att kopiera kod från andra. De lösningar som tas fram skall förses med bra kommentarer, korrekt indentering och bra variabelnamn. Koden ska m.a.o. vara lätt att läsa och förstå.

Hårdvara

- 1x Kopplingsdäck
- Kopplingskablar (hona-hane)
- Lysdioder vars färger passar trafikljus med övergångsställe
- Motstånd till dioderna
- 1x Aktiv summer

Andreas Axelsson
Alexander Göransson
Viktor Mann

JÖNKÖPING UNIVERSITY
School of Engineering

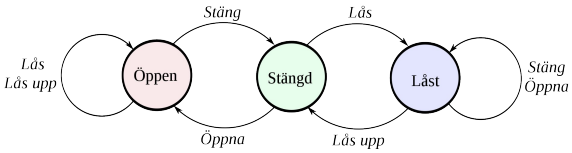
2023-05-03

Enchipsdatorer – Laboration 2
Trafikljus

1. Introduktion

I denna laboration ska du bygga en tillståndsmaskin. Från kursen Introduktion till Elektronik har ni förmodligen redan sett en liknande saker i och med tentauppgiften om lejonburen, eller den om tågen som ska åka över en bit järnväg med signaler.

En maskin av detta slag beskrivs med två slags element: Tillstånd och övergångar (states & events). Bilden intill är ett tydligt exempel på en tillståndsgraf för en låsbar dörr, stulen från Wikipedia ;)

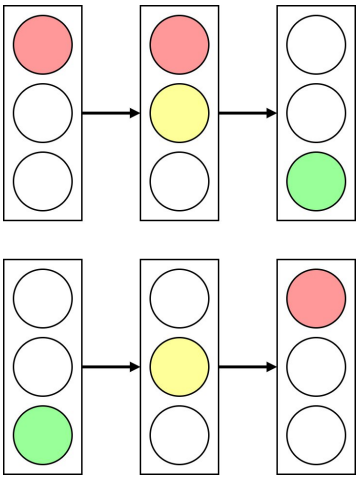


I grafen för den låsbara dörren skulle man kunna säga att tillståndet och output är samma sak, men i det program som du ska bygga i denna laboration så är de lite mer separerade.

Först ska du rita en graf med output från ett övergångsställe och sedan konvertera det till en tillståndsgraf. Efter det koda grafen som ett program för din STM32. När detta fungerar felfritt så kan du gå vidare till sista delen. Där ska du konfigurera interrupts (du kommer alltså använda CubeMX för att ställa in NVIC) så att programmet drar nytta av vad din hårdvara är kapabel till.

2. Problembeskrivning

Denna laborationsuppgift går ut på att bygga ett trafikljus för ett övergångsställe med tryckknapp. För att hålla saker enkla så ska lösningen hålla sig till dessa punkter:



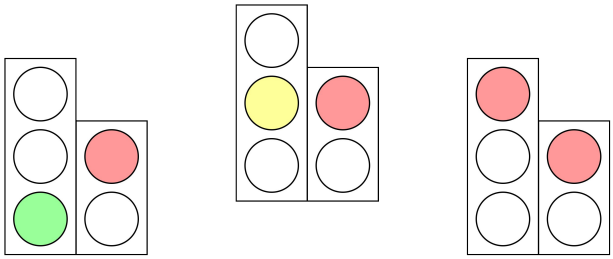
- Knapptryckningen är endast relevant när det är grönt ljus för bilarna.
- Knapptryckningar vid alla andra tillstånd ska ignoreras.
- Grönt ljus för bilar ska inte vara tidsbegränsat. Det enda sättet att lämna det tillståndet ska vara om knappen trycks.
- När en gångare tryckt på knappen så ska det fortsätta vara grönt för bilarna i ett fåtal sekunder.
- Alla andra tillstånd ska vara tidsbestämda.
- Systemet har inga händelser utöver knappen och klockan (riktiga trafikljus har sensorer under vägen mm.)
- Vid uppstart av programmet så ska det starta i ett reset/init-tillstånd som sedan ska leda till ett tillstånd där det är rött för både gångare och bilar.
- När bilarna börjar och slutar köra så ska lamporna vara tända och släckta enligt bilden till höger.

3. Tillståndsmaskin

3.1. Planering – Rita output-diagram

Rita nu output-sekvensen för ett övergångsställe. Fyll alltså på med pilar och de tillstånd som saknas.
Om du är osäker så finns ett övergångsställe väldigt nära: Går ut och ner för trappan mellan JIBS och JTH så hittar du det.

Fokusera på att få output (dvs. vilka lampor som lyser och ej) rätt. Exakt hur du kommer mellan tillstånden löses senare.

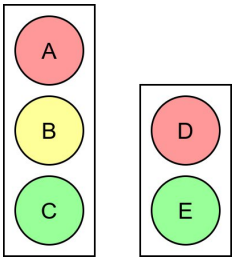


Andreas Axelsson
Alexander Göransson
Viktor Mann

Enchipsdatorer – Laboration 2
Trafikljus

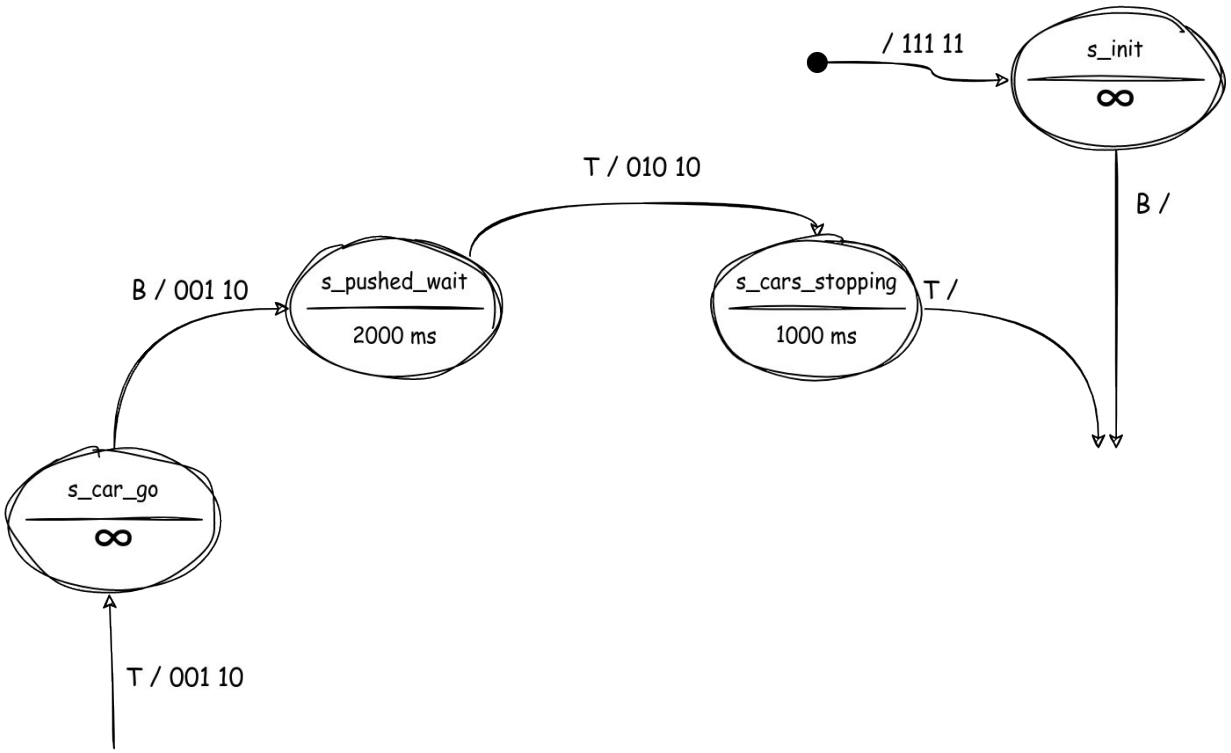
3.2.Planering – Rita state-diagram

Du ska nu rita ett tillståndsdiagram av Mealy-typ (dvs. output sätts vid state-transitioner och inte i state:et) för trafikljusets beteende. Till höger är de lampor du ska ha representerade, bilarnas ljus till vänster, gångarnas till höger. Din notation för output bör vara `ABC DE`. Lampan som tänds när en gångare trycker på knappen är *ej* del av denna output (den görs separat, senare).



Exempel: Output vid grön gubbe blir `100 01` och grönt ljus för bilarna blir `001 10`.

Rita ditt tillståndsdiagram på denna sida. Fokusera på output: Namnge inte tillstånden förrän diagrammet är komplett. `T` och `B` är de två event-typerna: `ev_state_timeout` och `ev_button_push`, se kapitel 3.4 för detaljer. Det bör bli runt 8 st. tillstånd. Pilar som går från ett state till sig själv skall ej ritas.



Det är en god idé att visa ditt tillståndsdiagram för en klasskamrat/handledare/lärare och fråga ”Tror du att detta kommer bli bra?”. Extra tid lagd på planering kommer att skona dig från framtida huvudvärk.

4. Datatyper

4.1.Event-typen

Ingen tillståndsmaskin är fullständig utan events. Skriv in event-typens deklaration i ditt program:

```
enum event
{
    ev_none,
    ev_button_push,
    ev_state_timeout
};
```

Notera att deklarationen görs med `enum event` och inte `typedef`. Det senare kan vara användbart men samtidigt så döljer det saker vi inte vill ha dolda här.

En `enum` är ett alias för en int. Varje instans (i detta fall de tre listade) motsvarar en unik siffra. Vill man säkerställa att en viss enum-instans blir en viss siffra så kan detta göras:

```
ev_none = 0,
```

4.2.State-typen

Väldigt likt hur event-typen är kodad ska du skriva in din state-typ. Tag namnen ur diagrammet på föregående sida och skriv in dem

```
enum state
{
    s_init,
    s_car_go,
    /* others go here */
};
```

I C kan man lätt råka ut för namnkonflikter[1]. För att undvika sådant så är det inte ovanligt att man lägger till prefix (t.ex. ”`HAL_`”) på sina enums och funktionsnamn:

Om ett tillstånd heter `cars_go_vroom` så är det lämpligt att döpa enum-värdet till `s_cars_go_vroom` eller `ps_cars_go_vroom`. S är kort för ”state”, PS för ”program state”.

[1] C++ går runt detta med sina namespaces, t.ex. `std::`

Andreas Axelsson
Alexander Göransson
Viktor Mann

Enchipsdatorer – Laboration 2
Trafikljus

5. Output-funktioner

När din state-enum är klar så sätt upp dina dioder i CubeMX och koppla dem på ett kopplingsdäck. Alla oanvända ben utom PA6 går bra.

5.1.De fem lamporna

Dess fullständiga typsignatur ska vara

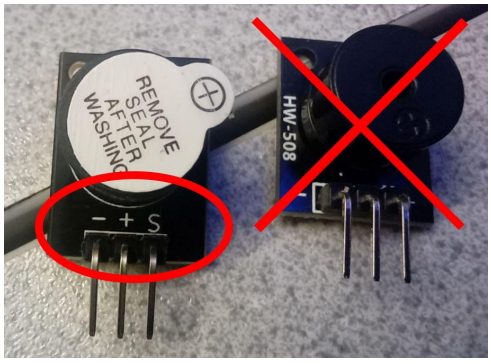
```
void set_traffic_lights(enum state s)
```

Den ska sätta output efter argumentet `s`. Om argumentet är `s_init` så ska alla lampor vara tända. Görs enklast som en stor switch-sats. Argumentet `s` ska entydigt bestämma lampornas output, dvs. använd inga globaler.

5.2.Tryckindikator *[kan skippas tills vidare]*

Lägg till en diod för att indikera om någon tryckt på knappen vid övergångsstället. Skapa en eller två (`push_button_light_on()`/`push_button_light_off()`) funktioner som tänder och släcker denna diod.

5.3.Aktiv Summer *[kan skippas tills vidare]*

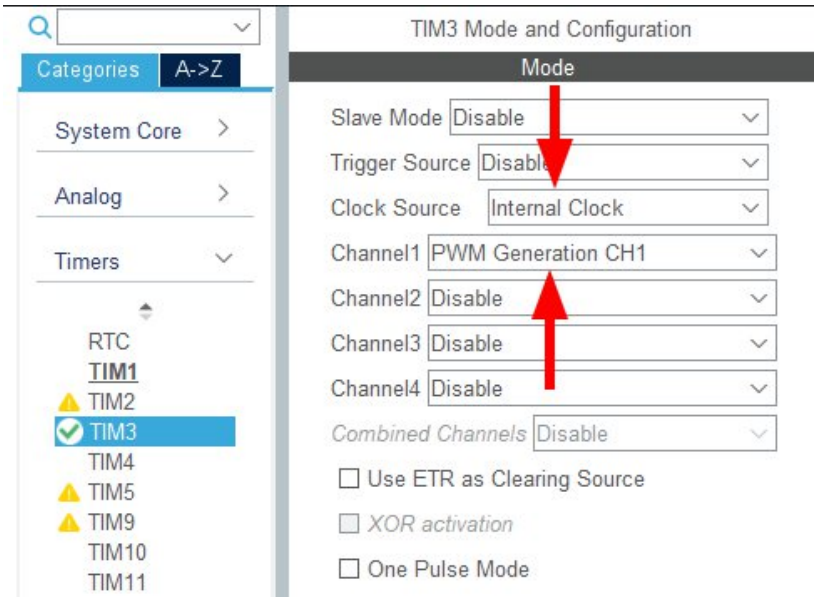


För att driva handledare och andra oskyldiga människor till vansinne finns en modul, `abuzz.h` som du kan använda. Vänligen ta **EJ** bort klisterlappen över högtalarmembranet.

Komponenten till vänster i bilden är den aktiva summern. Anslut benet märkt S till PA6. Koppla 3V3 till `+` och GND till `-`.

Dra och släpp filen `abuzz.h` till mappen `Core/Inc` i ditt projekt så att den landar intill `main.h`. När du får upp en dialogruta så ska du välja att filen ska kopieras in i ditt projekt. Detta gäller när du tar in filer i CubeIDE:

- `.h` → `Core/Inc`
- `.c` → `Core/Src`



Öppna CubeMX, klicka på TIM3 och sätt inställningarna enligt bilden här. Spara och generera kod.

Leta reda på anropet till `MX_TIM3_Init()` i din kod. Lägg efter det till raden

```
HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_1);
```

Efter att den raden körts så kommer funktionerna i `abuzz.h` fungera, givet att du importerat filen. Missa inte att du först måste anropa `abuzz_start()`, annars kommer varken `abuzz_p_long()` eller `abuzz_p_short()` fungera. Läs i `abuzz.h` för ytterligare detaljer.

6. Programstruktur

Deklarera två variabler (inne i main – globala variabler ska undvikas när möjligt) för att kunna hålla state och event:

```
enum state st = s_init;
enum event ev = ev_none;
```

Ditt program (main-loopen) ska bestå av två segment:

1. Lägg rätt värde i `ev` för denna loop.
2. Kör relevant kod givet det värde `st` har.

Den första kan vara lite knepig att få rätt men att köra rätt kod för nuvarande värde på `st` är en stor switch-sats med små kodblock för varje övergång till ett nytt state.

7. Event-avläsning

Det finns tre olika events. Varje varv genom main-loopen så måste ett event sättas. Både `ev_button_push` och `ev_state_timeout` är rätt sällsynta att de händer, så nästan varenda loop kommer `ev_none` att användas.

Börja med att ha raden

```
ev = ev_none;
```

längst upp i din main-loop. På så sätt så blir de andra ett ”undantag” från detta standardvärde. Ett sätt att skriva kod på som gör saker lite enklare.

Lösningen du kommer göra i denna del av laborationen kallas för en polling-lösning. Dvs. koden kollar aktivt upp vad som har hänt.

Enchipsdatorer – Laboration 2
Trafikljus

7.1.Generera ev_button_push

Kopiera in funktionen `is_button_pressed()` från tärnings-laborationen. Saker är tyvärr inte så enkla så du nu kan ha

```
ev = ev_none;
pressed = is_button_pressed();
if ( pressed )
{
    ev = ev_button_press;
}
```

Din CPU är så snabb så då skulle du generera hundratals (eller tusentals) `ev_button_press` varje millisekund!

Istället ska `ev_button_press` enbart genereras när knappen går från att inte vara tryckt till att vara tryckt. För att ha koll på en flank så behöver du komma ihåg föregående loops tillstånd. Gör detta genom att ha två variabler, deklarerade innan `while(1)`:

```
int curr_press = is_button_pressed();
int last_press = curr_press;
```

En knapptrycknings-event ska enbart uppkomma när knappen är intryckt denna loop och om knappen ej var intryckt under förra loopen. Det är alltså bara vid en positiv flank som `ev_button_push` ska uppkomma.

7.2.Generera ev_state_timeout

För tid används SysTick, vilket är på millisekund-nivå. Funktionen du bör använda för att läsa av tiden finns med i `JU_STM32_HAL-beskrivning.pdf`.

Om det inte blev någon `ev_button_push` så ska koden kontrollera om det kan vara en `ev_state_timeout`.

Lägg till tre variabler i din kod. De skall hålla tick-värden, så de ska vara av samma typ som returytypen hos den funktionen du nyss kollade upp.

```
ticks_left_in_state, curr_tick, last_tick
```

I `ticks_left_in_state` ska de tider du skrivit upp i ditt tillståndsdiagram läggas varje gång du ändrar värdet på `st` (mer om detta senare). Varje gång din kod ser att ett tick har gått så ska `ticks_left_in_state` sänkas. Blir den noll är det dags att generera en `ev_state_timeout`. Var klyftig med dina if-satser så att `ev_state_timeout` endast genereras en gång.

På liknande sätt som du kontrollerar efter flank för knappen, så ska du kontrollera om det är ett nytt tick.

Vissa state-transitioner ska endast hända vid knapptryckning. Om ett state är sådant, t.ex. `s_init`, så kommer `ticks_left_in_state` vara 0 så länge programmet är i detta tillstånd. Då skall inga `ev_state_timeout` genereras. Detta betyder alltså att eventen bara ska genrereras i och med att `ticks_left_in_state` NÅR 0. Aldrig när den redan är på 0.

8. Event-hantering och tillståndsbyte

Exempel: I och med att du går från ett tidsbundet tillstånd till ett annat tidsbundet tillstånd (det andra är 2500 ms) så skulle din kod kunna se ut så här:

```
case s_car_stop:
    if ( ev == ev_state_timeout )
    {
        ev                  = ev_none;         // clear event
        st                  = s_car_starting; // set next state
        ticks_left_in_state = 2500;           // set next timeout
        set_traffic_lights(s_car_starting);   // set output
    }
    break;
```

Var noga med att du konstruerar en Mealy-maskin. Det betyder att output-funktionerna endast anropas i och med att det sker en övergång från ett state till ett annat. I andra state-maskiner anropas output-funktionerna varje loop, oavsett om någon förändring skett eller ej.

9. Kodkrav

Lägg till stöd för tryckindikator och summer i din kod.

- Har korrekta enum-typer för event och state.
- Inga globala variabler.
- Output sätts endast vid state-transitioner.
- Följer det beteende som listats i problembeskrivningen.
- Genererar event enbart vid flanker.

Godkänd polling-lösning:

Enchipsdatorer – Laboration 2
Trafikljus

STOPP! Gå endast vidare när allt tidigare fungerar perfekt! Se kodkraven för att vara säker. **STOPP!**

10. Interrupts och event-kö

Nu fungerar labben – till ytan sett – som önskvärt. Lösningen med att läsa in en event i början av varje loop är vad som kallas för ”polling”. I detta kapitel ska din kod gå över till att använda interrupts[2] istället. Först för knappen och sedan för klockan.

För att detta ska fungera väl så måste du först ha något mer sofistikerat än bara en variabel för din event.

10.1. Event-kö

För att interrupts ska fungera och för att du inte ska råka skriva över en event vid maximal otur så är en kö en bra idé. Lägg först till en ny event sist i listan:

```
ev_error = -99
```

Denna används för att enkelt kunna märka av när något gått snett.

EVQ är en förkortning för ”EVent Queue”. Lägg till följande rader efter definitionen av `enum event { ... }`:

```
#define EVQ_SIZE 10

enum event evq[ EVQ_SIZE ];
int        evq_count       = 0;
int        evq_front_ix    = 0;
int        evq_rear_ix     = 0;
```

Börja med att skriva en funktion `evq_init()` som fyller hela `evq` med `ev_error`. Anropa den någonstans i din initiering, innan main-loopen.

Funktionerna som sedan behövs är

```
void       evq_push_back(enum event e);
enum event evq_pop_front();
```

De ska arbeta med variabeln `evq` som är tänkt att fungera som en cirkulär kö. Nya element ska läggas in på platsen `evq_rear_ix` och element ska hämtas från platsen `evq_front_ix`. Variabeln `evq_count` ska hålla hur många element som finns

[2] Benämns ibland som ISR (Interrupt Service Routine).

lagrade i kön just nu. Finns inga element i kön vid `pop()` ska `ev_none` returneras. Är kön full vid `push()` så ska det nya elementet ignoreras.

Försök gärna att implementera kön själv, som övning. En fungerande implementation finns i Appendix, i slutet av detta dokument.

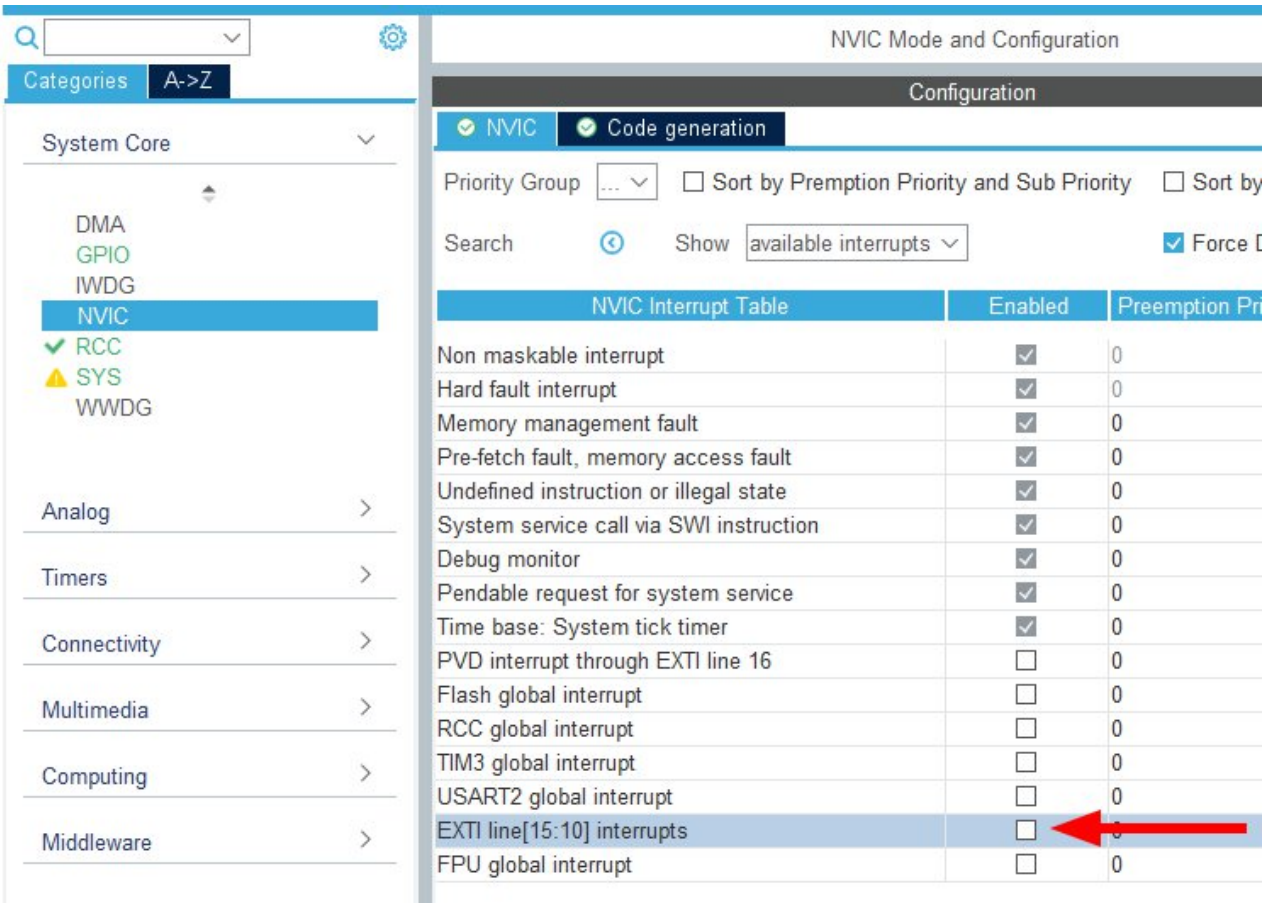
När du har funktionerna inskrivna så ändra event-delen av din main-loop: Kommentera ut din tidigare event-avläsning och använd denna rad (vilket använder sig av kön) används istället:

```
ev = evq_pop_front();
```

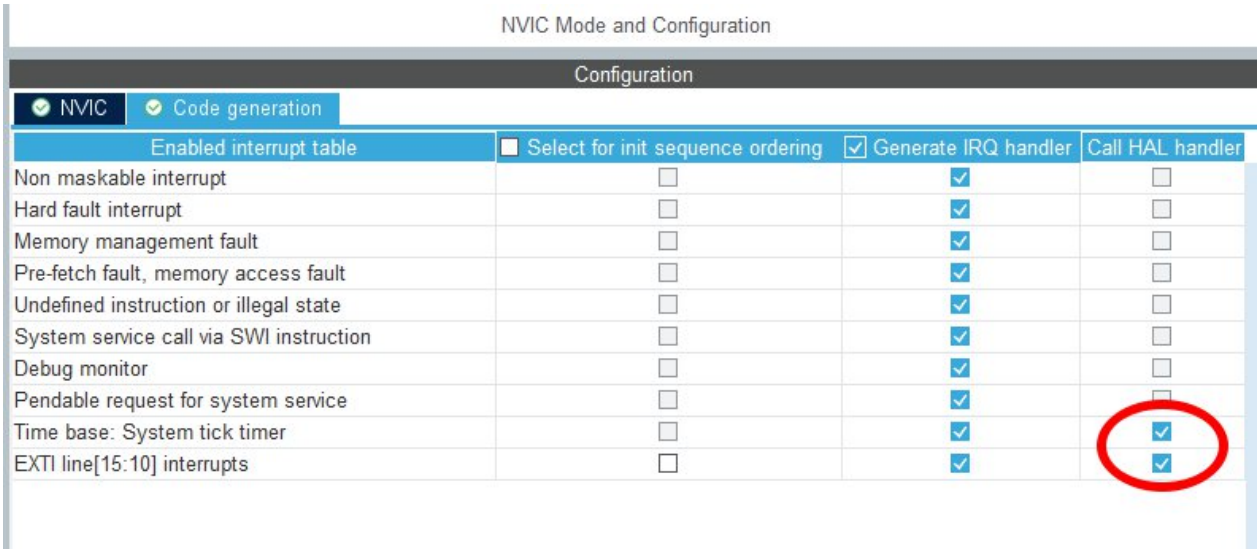
Din kod ska nu fungera likadant som den gjorde innan den använde en kö.

10.2.Interrupt för knappen

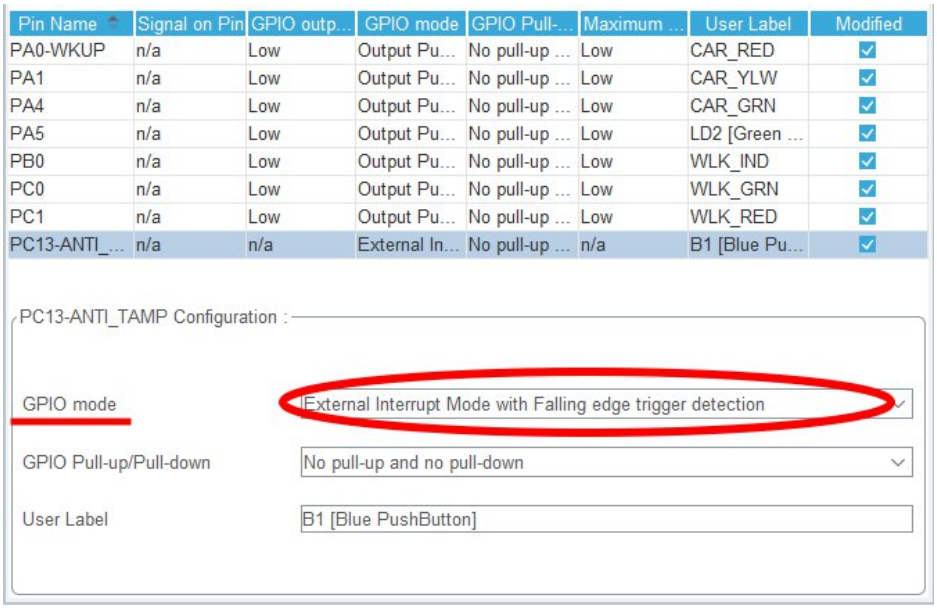
Istället för att varje loop kontrollerar om knappen är tryckt så ska ditt program sättas upp så att en ISR körs varje gång knappen trycks. Denna ISR ska vara vad som registrerar en `ev_button_push`. Börja med att öppna CubeMX och bocka i rutan utpekad i bilden:



Öppna nu fliken ”Code generation”. Det borde se ut som i bilden här. Märk att det är inte bara en HAL-funktion för knappen som kommer att anropas utan även en funktion relaterad till SysTick. Den senare kommer att tas upp i nästa kapitel.



Det finns olika inställningar för *när* din ISR kommer att anropas. Öppna GPIO-inställningarna i CubeMX. Markera raden för knappen och observera att inställningarna för GPIO-porten ser annorlunda ut än vad den gör för de i output-läget.



Sätt GPIO mode till ”External Interrupt Mode” (ej ”External Event”) så att det matchar den flank som kommer då knappen trycks ned.

Skriv nu in din interruptfunktion i `main.c`. Läs i `JU_STM32_HAL-beskrivning.pdf` för detaljer kring detta. Den ska registrera ett event i kön.

10.3.SysTick

I förra kapitlet märkte vi att det finns kod (HAL handler) som körs vid SysTick. Expandera mappen `Core/Src` i ditt projektträd. Där har du filen `stm32f4xx_it.c`. Öppna den och titta i Outline-listan till höger.

Om du konfigurerat din knapp korrekt så hittar du `EXTI15_10_IRQHandler()`[3] i listan. Intill den har du funktionen `SysTick_Handler()`. Denna är vad som anropas varje gång ett tick har gått på ditt system. För det mesta har inbyggda system en tickrate på 1000 tick per sekund. Notera att det finns plats för din egen kod i `SysTick_Handler()`. Du vill lägga till kod i `SysTick_Handler()` som kan manipulera de variabler som deklarerats i `main.c`.

Filen `stm32f4xx_it.c` inkluderar `”main.h”`. Öppna `main.h` och lägg till denna funktionsdeklaration på lämplig plats:

```
void my_systick_handler();
```

Lägg till ett anrop till `my_systick_handler()` i `SysTick_Handler()`. Gör det i USER CODE-blocket efter anropet till `HAL_IncTick()`. Skriv sedan definitionen för `my_systick_handler()` i `main.c`. Börja med denna definition:

```
int systick_count = 0;

void my_systick_handler()
{
    systick_count++;
    if (systick_count == 1000)
    {
        HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
        systick_count = 0;
    }
}
```

Om detta får lampan på kortet att byta tillstånd en gång i sekunden så fungerar anropen.

Gör variabeln `ticks_left_in_state` global. Funktionen `my_systick_handler()` ska anropa `evq_push_back(ev_state_timeout)` på lämpligt vis. P.g.a. hur du satt

[3] Om du följer dess kod (Ctrl+klick eller högerklick & ”Open Declaration”) så kommer du se att det är via den som knappens callback-funktion anropas.

upp `my_systick_handler()` så vet du att den blir anropad precis efter att tick-siffran[4] stigit. Testet `curr_tick != last_tick` ska alltså inte utföras.

11. Appendix

```
#define EVQ_SIZE 10

enum event evq[ EVQ_SIZE ];
int        evq_count       = 0;
int        evq_front_ix    = 0;
int        evq_rear_ix     = 0;

void evq_push_back(enum event e)
{
    // if queue is full, ignore e
    if ( evq_count < EVQ_SIZE )
    {
        evq[evq_rear_ix] = e;
        evq_rear_ix++;
        evq_rear_ix %= EVQ_SIZE;
        evq_count++;
    }
}

enum event evq_pop_front()
{
    enum event e = ev_none;
    if ( evq_count > 0 )
    {
        e = evq[evq_front_ix];
        evq[evq_front_ix] = ev_error; // detect stupidity
        evq_front_ix++;
        evq_front_ix %= EVQ_SIZE;
        evq_count--;
    }
    return e;
}
```

Godkänd interrupt-lösning:

[4] Siffran som kan hämtas med `HAL_GetTick()`.