Nicolas Cavedon & Augustin Mohr
Lausanne, le 19.10.2020

Embedded Systems – Lab 1: MSP432

Introduction

For this first lab of the course Embedded Systems, we will be controlling a servomotor using a potentiometer-based joystick. To do that we will program an MSP432 board.

High level description

The joystick's analogic output will be connected to P4.0, which corresponds to an input of the analogic to digital converter. The servomotor will be connected to P2.4, the output of Timer_A0. It will be controlled using a PWM with a 20ms period and a varying duty time of 1-2ms depending on the position of the joystick.

We will use the ADC14 module to convert the values outputted by the potentiometer. It will take samples every 50ms, using Timer_A1. An interrupt triggered by it will update the duty time of the PWM.

Low level description

Clock

The clock we will be using is the Auxiliary clock. As the CS registers are protected by a password, we must first modify the key register with the corresponding password. The auxiliary clock can then be enabled, and we have chosen the REFO clock as the source, as it has a 128 kHz base frequency. This facilitates our conversions and gives us an acceptable precision. The key register is then filled with a value different than the password, to properly protect the CS registers.

```
 97 void initClock(void) {
 98     CS->KEY = CS_KEY_VAL;
 99     CS->CLKEN |= CS_CLKEN_ACLK_EN;      // ENABLE DE ACKL
100     CS->CLKEN |= CS_CLKEN_REFO_EN;      // ENABLE REFO OSCILLATOR
101     CS->CLKEN |= CS_CLKEN_REFOFSEL;     // FREQUENCY = 128 kHz
102     CS->CTL1 &= ~CS_CTL1_DIVA_MASK;
103     CS->CTL1 |= CS_CTL1_DIVA_0;
104     CS->KEY = 0x00000000;
105 }
```

Figure 1 : Clock Initialisation

GPIO

For this lab, we are using two of the General-Purpose Inputs/Outputs of the board. One as output for our PWM, and one as input for the ADC. We use port P2.4 as output for the PWM. First, we define P2 as an output, by writing 0xFF in the DIR register. Then we have to set the P2.4's function to TA0 CCR1 compare output using the SEL0 and SEL1 registers.

Then, we use port P4.0 as the input of our ADC by configuring the SEL0 and SEL1 registers so the P4.0 is routed to pin A13, which can be used as input by the ADC.

```
116
117 void initGPIO(void) {
118
119     P2->DIR = 0xFF;
120     P2->SEL0 |= (0b1 << 4);
121     P2->SEL1 &= ~(0b1 << 4);
122
123     P4->SEL0 |= 0b1;
124     P4->SEL1 |= 0b1; // sets p4.0 as A13
125
126
```

Figure 2 : GPIO initialisation function

Timer_A

The Timer_A module will be used to control both the PWM and the ADC14 sampling timer. The former will be configured using Timer_A0, and the latter Timer_A1. For both timers, we start by clearing them and setting the two clock dividers to "/1" (divide by 1) so we keep our 128kHz frequency. We also set them to compare mode. We configure Timer_A0 to count in UP/DOWN mode for our PWM, and Timer_A1 in UP mode, as we only need it to generate a periodic interrupt to start the ADC's conversion.

```
107 void initTimerA(void) {
108     TIMER_A0->CTL |= TIMER_A_CTL_CLR;
109     TIMER_A0->CTL &= ~TIMER_A_CTL_MC_MASK;
110     TIMER_A0->CTL &= ~TIMER_A_CTL_ID_MASK;
111     TIMER_A0->CTL |= TIMER_A_CTL_MC__UPDOWN | TIMER_A_CTL_SSEL__ACLK;       // CONFIG DE TIMER A0; SOURCE = ACKL, MODE = UP/DOWN
112     TIMER_A0->CTL |= TIMER_A_CTL_ID_0;
113     TIMER_A0->CCTL[0] &= ~TIMER_A_CCTLN_CAP;                                // compare mode
114     TIMER_A0->EX0 &= ~TIMER_A_EX0_IDEX_MASK;
115     TIMER_A0->EX0 |= TIMER_A_EX0_TAIDEX_0;
116
117     TIMER_A1->CTL |= TIMER_A_CTL_CLR;
118     TIMER_A1->CTL &= ~TIMER_A_CTL_MC_MASK;
119     TIMER_A1->CTL &= ~TIMER_A_CTL_ID_MASK;
120     TIMER_A1->CTL |= TIMER_A_CTL_MC__UP | TIMER_A_CTL_SSEL__ACLK;           // CONFIG DE TIMER A1; SOURCE = ACKL, MODE = UP
121     TIMER_A1->CTL |= TIMER_A_CTL_ID_0;
122     TIMER_A1->CCTL[0] &= ~TIMER_A_CCTLN_CAP;                                // compare mode
123     TIMER_A1->EX0 &= ~TIMER_A_EX0_IDEX_MASK;
124     TIMER_A1->EX0 |= TIMER_A_EX0_TAIDEX_0;
125
126
127     TIMER_A1->CCR[0] = ADC_PER;                                             // 50ms period
128 }
```

Figure 3 : Timer_A initialisation

Generating a PWM

To achieve the right frequency needed for the PWM, we need to correctly set the CCR0 register. Since the timer has a base frequency of 128kHz and is in up/down mode, we must count to 16 to have a 1ms period. The CCR0 register is therefore initialized with the value 20 * 16 to have a 20ms period. To have the proper duty cycle, we must again multiply the value of our required duty time in ms by 16. In the initialization function, we initialize the register with a value that will be overwritten when the first ADC conversion is done. We then need to choose the correct output mode for our PWM. We want our signal to be high for 1-2ms, then low for the remaining 18-19ms. The timer used is set to be in Up/Down mode. Therefore, the correct output mode according to the technical reference manual is output mode 6, toggle/set. However, when we use that output mode, we get the behaviour described as output mode 2 in the documentation. When we use output mode 2, it reverts to the behaviour we want, which is why we ended up choosing this setting over the output mode 6.

```
88 void initPWM(float duty) {
89
90     TIMER_A0->CCR[0] = PWM_PER;                          // 20 ms period
91     TIMER_A0->CCR[1] = duty * 16;                        // duty cycle
92     TIMER_A0->CCTL[1] &= ~TIMER_A_CCTLN_OUTMOD_MASK;
93     TIMER_A0->CCTL[1] |= TIMER_A_CCTLN_OUTMOD_2;         // Toggle/reset, acts here as Toggle/set for some reason
94
95 }
```

Figure 4 : PWM generation

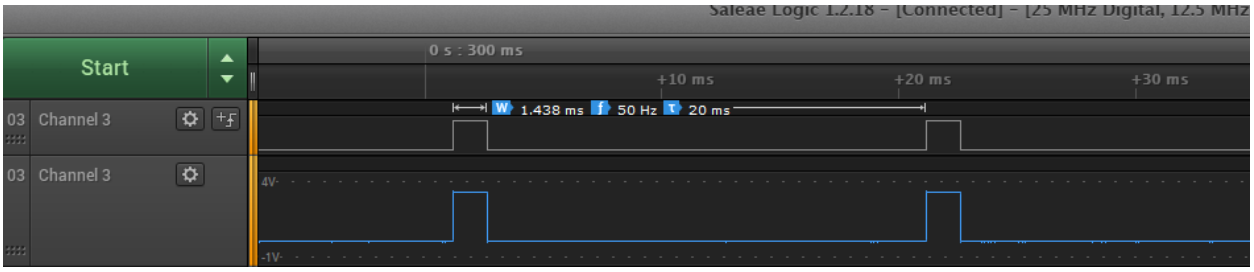

Figure 5 : PWM measured with the Logic Analyser

ADC

To configure the ADC for this lab, we start by selecting the clock we want to use, in our case it's the auxiliary clock ACLK. Then we configure the ADC to "sample and hold" mode, and we select the software trigger source as our trigger source. We also specify the resolution we want (14bits) and the source of the analogic signal we want to convert (here it's A13). Then we just turn on the ADC and enable the conversion.

```
140 void initADC(void){
141
142
143     ADC14->CTL0 &= ~ADC14_CTL0_SSEL_MASK;
144     ADC14->CTL0 |= ADC14_CTL0_SSEL_2;        // select ACLK
145     ADC14->CTL0 |= ADC14_CTL0_SHP;           // sample and hold mode
146     ADC14->CTL0 &= ~ADC14_CTL0_SHS_MASK;
147     ADC14->CTL0 |= ADC14_CTL0_SHS_0;         // SC mode
148     ADC14->CTL1 &= ~ADC14_CTL1_RES_MASK;
149     ADC14->CTL1 |= ADC14_CTL1_RES__14BIT;    // 14bit resolution, 16 clock ticks sampling time
150     ADC14->CTL0 |= ADC14_CTL0_ON;            // enables the ADC
151     ADC14->MCTL[0] |= ADC14_MCTLN_INCH_13;
152     ADC14->CTL0 |= ADC14_CTL0_ENC;
153 }
```

Figure 6 : ADC Initialisation

Interrupts

We have two interrupts in our program; the first one, generated by timer_A1, resets the interrupt flag and starts the ADC conversion by setting the "start conversion flag" to 1. The second interrupt is generated once the conversion is finished, and the values stored in the ADC14 memory register have been updated. In this interrupt, we update the Timer_A0 CCR1 register to reflect the values read by the converter. To do this, we first add 16 which corresponds to 1ms. Then, the value read by the converter is divided by 2048, which can be done more efficiently by shifting the bits to the right by 10. This is done because we again need to multiply by 16, and because the converter is set to store values in 14 bits of precision, we need to divide it by 2^14 to have a normalized value. 2^14/16 = 2048, which is 2^10.

```
130 void initInterrupts(void){
131
132     NVIC_EnableIRQ(TA1_N_IRQn);          // IRQ handler
133     NVIC_SetPriority(TA1_N_IRQn,0);      // priority
134     TIMER_A1->CTL |= TIMER_A_CTL_IE;     // Interrupt enable
135     NVIC_EnableIRQ(ADC14_IRQn);          // IRQ handler
136     NVIC_SetPriority(ADC14_IRQn,0);      // priority
137     ADC14->IER0 |= ADC14_IER0_IE0;       // ADC register 0 Interrupt
138 }
```

Figure 7 : Interrupts initialisation

```
157 void TA1_N_IRQHandler(void) {
158
159     TIMER_A1->CTL &= ~TIMER_A_CTL_IFG;                  // reset interrupt flag
160
161     ADC14->CTL0 |= ADC14_CTL0_SC;                       // start conversion
162
163
164 }
165
166 void ADC14_IRQHandler(void) {
167
168
169     resultat = (uint16_t)ADC14->MEM[0];
170     TIMER_A0->CCR[1] = 16 + (resultat >> 10); // 1ms + 0-1 ms depending on the adc
171
172
173 }
```

Figure 8 : Interrupt routines

Conclusion

We have achieved the goal to control the servomotor with a potentiometer. We can see that even for such a simple task as this, a lot of configuration must be done.