Lab 1.0
MSP432 Microcontroller I/O Programming

Introduction

A microcontroller is an integrated circuit comprised of one or more processors, memories, and programmable interfaces. Microcontrollers are most often used for, well, "control" purposes. When a processor is used for control, we mean that the processor is used to perform fine-grained coordination between different interfaces found in a system (GPIO ports, serial ports, ADCs, DACs, timers, …). Note that a processor cannot "control" an interface if the interface is not configurable (as otherwise the interface has a fixed function). This is why such interfaces are programmable and are hence called *programmable interfaces*. Programmable interfaces are what this lab is all about.

The microcontroller we will be using for this exercise is from Texas Instruments' MSP432 family, more specifically the MSP432P401R shown in Figure 1.

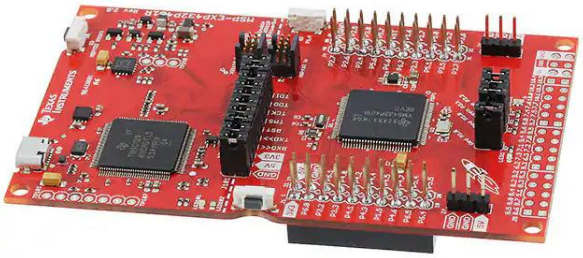

FIGURE 1. MSP432P401R LAUNCHPAD EVALUATION KIT

Goal

The goal of this lab is to program the MSP432P401R so as to perform some basic monitoring and control tasks. The task is to periodically sample analog data from a potentiometer using an ADC and to output a proportionally-sized PWM signal to turn a servomotor accordingly. You will then use an oscilloscope or a logic analyzer to check the generated PWM signal for correctness.

René Beuchat, Sahand Kashani

This lab is broken into 7 exercises to help you incrementally understand how to program the individual programmable interfaces needed to achieve this final goal. Note though that we only ask that the *final task* (exercise 7) of this lab be explained in your report.

Relevant Documentation

TI's processors/microcontrollers are generally described in two documents:

- A family technical reference manual that describes the microcontroller's core and peripherals' functionality in detail.
- A device-specific datasheet that ties the reference manual concepts to the device (it specifies, for example, the device pinout, the peripherals' base addresses, …)

Most of the time, you will need to use both datasheets to program the microcontroller properly. You may also find the Launchpad Development User guide useful to check the pinout of the board, external crystals, …
Note that learning to quickly browse through such documents to identify what you need in order to perform a task is an *essential part of this course*. Many manuals are complex and seem to contain an innumerable amount of information, but you must learn to sift through it if you want to work with such devices!

Accessing Programmable Interfaces

Theory

Programmable interfaces expose their functionality and configurability to the processor through their *register map* (i.e., a table that explains what each of the peripherals' control registers do). One therefore needs to know how to access this register map before we can control the peripheral. Programmable interfaces lie on a microcontroller's I/O bus, so we start from there.

If a processor has a *dedicated* I/O bus (such as AVR-based processors), then *special I/O instructions* must be used to access the I/O bus. In contrast, if a processor has a *memory-mapped* I/O bus (such as ARM-based processors), then *simple load/store instructions* can be used to access the I/O bus.

The MSP432 has a memory-mapped I/O bus and peripherals' register maps are therefore embedded within the address space of the processor. As a programmer you can access a peripheral's registers by reading/writing within the address range that corresponds to the peripheral. You can find typically find the address ranges that correspond to different peripherals within a microcontroller's technical reference manual.

Practice

In general, a microcontroller manufacturer provides users with a software *Hardware Abstraction Level* (HAL) library that allows rapid development by abstracting away the device's low-level hardware considerations through the use of high-level functions. However, as a deep understanding of low-level development and how to access programmable interfaces is required for the following labs (on FPGAs), you are asked *not to use* the HAL libraries in this lab (as otherwise you would learn nothing ☺).

Each programmable interface in a microcontroller is located at a dedicated base offset within the processor's address space. These offsets are described in the technical reference manual, but *please* do not hard-code them in your C code! Doing so would make debugging your code a nightmare. You should instead use named constants whose purpose are easily understandable. Thankfully TI provides programmers with a header file

that defines constants and data structures related to every programmable interface, therefore allowing us to write more readable code. You are strongly recommended to use these headers! To do so, you will need to include the `msp.h` header file as follows.

```
21 #include "msp.h"
```

Underneath the hood, the `msp.h` header actually includes the `msp432p401r.h` header which contains everything needed to program the microcontroller. This header mainly defines C structures for each peripheral and convenient names for each bit of a register associated to a given peripheral. An example pre-defined structure for the MSP432P401R's *Clock System* peripheral is shown below:

```
typedef struct {
  __IO uint32_t KEY;                        /**< Key Register */
  __IO uint32_t CTL0;                       /**< Control 0 Register */
  __IO uint32_t CTL1;                       /**< Control 1 Register */
  __IO uint32_t CTL2;                       /**< Control 2 Register */
  __IO uint32_t CTL3;                       /**< Control 3 Register */
  __IO uint32_t CTL4;                       /**< Control 4 Register */
  __IO uint32_t CTL5;                       /**< Control 5 Register */
  __IO uint32_t CTL6;                       /**< Control 6 Register */
  __IO uint32_t CTL7;                       /**< Control 7 Register */
       uint32_t RESERVED0[3];
  __IO uint32_t CLKEN;                      /**< Clock Enable Register */
  __I  uint32_t STAT;                       /**< Status Register */
       uint32_t RESERVED1[2];
  __IO uint32_t IE;                         /**< Interrupt Enable Register */
       uint32_t RESERVED2;
  __I  uint32_t IFG;                        /**< Interrupt Flag Register */
       uint32_t RESERVED3;
  __O  uint32_t CLRIFG;                     /**< Clear Interrupt Flag Register */
       uint32_t RESERVED4;
  __O  uint32_t SETIFG;                     /**< Set Interrupt Flag Register */
       uint32_t RESERVED5;
  __IO uint32_t DCOERCAL0;                  /**< DCO External Resistor Cailbration 0 Register */
  __IO uint32_t DCOERCAL1;                  /**< DCO External Resistor Calibration 1 Register */
} CS_Type;


#define PERIPH_BASE  ((uint32_t)0x40000000)    /**< Peripherals start address */
#define CS_BASE      (PERIPH_BASE +0x00010400) /**< Base address of module CS registers */
#define CS           ((CS_Type *) CS_BASE)

/* Pre-defined bitfield values */
#define CS_KEY_VAL   ((uint32_t)0x0000695A)    /** CS control key value */
```

FIGURE 2. CLOCK SYSTEM STRUCTURE DECLARATION AND BASE OFFSET

Given the definitions above, a user can write to the KEY register of the clock system interface using the following C code statement:

```
CS->KEY  =  CS_KEY_VAL;
```

At first glance it can be difficult to understand how this statement actually writes to a register in the clock system peripheral. A lot is going on under the hood in this simple statement, so let's break it down to better understand what is happening.

1. First, what is the address of the **KEY** register in the clock system interface? We need to know this address if we want to write to it. This information can be obtained from table 6-2 in the technical reference manual and is shown below.

Offset	Acronym	Register Name	Type	Access	Reset	Section
00h	CSKEY	Key Register	Read/write	Word	0000_A596h	Section 6.3.1
04h	CSCTL0	Control 0 Register	Read/write	Word	0001_0000h	Section 6.3.2
08h	CSCTL1	Control 1 Register	Read/write	Word	0000_0033h	Section 6.3.3
0Ch	CSCTL2	Control 2 Register	Read/write	Word	0001_0003h	Section 6.3.4
10h	CSCTL3	Control 3 Register	Read/write	Word	0000_00BBh	Section 6.3.5
30h	CSCLKEN	Clock Enable Register	Read/write	Word	0000_000Fh	Section 6.3.6
34h	CSSTAT	Status Register	Read	Word	0000_0003h	Section 6.3.7
40h	CSIE	Interrupt Enable Register	Read/write	Word	0000_0000h	Section 6.3.8
48h	CSIFG	Interrupt Flag Register	Read	Word	0000_0001h	Section 6.3.9
50h	CSCLRIFG	Clear Interrupt Flag Register	Write	Word	0000_0000h	Section 6.3.10
58h	CSSETIFG	Set Interrupt Flag Register	Write	Word	0000_0000h	Section 6.3.11
60h	CSDCOERCAL0	DCO External Resistor Calibration 0 Register	Read/Write	Word	0100_0000h	Section 6.3.12
64h	CSDCOERCAL1	DCO External Resistor Calibration 1 Register	Read/Write	Word	0000_0100h	Section 6.3.13

FIGURE 3. CLOCK SYSTEM REGISTERS

Notice that **CSKEY** is at offset 00h from the base address of the clock system's register map and that the **CS** macro defines a pointer of type **CS_Type** initialized to the value **CS_BASE**. Any loads/stores performed through the **CS** macro therefore read/write to registers of the clock system interface.

2. Now that we know where the **KEY** register is located, we need to write to it. Remember that the MSP432P401R has a memory-mapped I/O bus and that peripherals are simply found at different offsets within the processor's address space which must be accessed through load/store instructions. If you remember your pointer arithmetic lecture when learning C, you'll remember that the arrow operator is equivalent to a pointer dereference followed by a field access. The figure below shows two equivalent statements.

```
CS->KEY    = CS_KEY_VAL;
(*CS).KEY  = CS_KEY_VAL;
```

Dereferencing the **CS** macro yields **CS_BASE**, and the field access to **KEY** provides an offset (00h) from **CS_BASE** due to how a C compiler stores structures in memory[1]. Finally, the assignment operator (=) causes a store instruction to be generated for address (**CS_BASE + 00h**).

This is the magic behind how such a clean simple statement can result in a write to a programmable interface's register. With this knowledge you should be able to understand how to access any programmable interface's register in the MSP432P401R used for this lab.

[1] Fields within a structure are all aligned, so accessing different fields within the structure results in different addresses being generated.

Exercises (intertwined with theory)

Clock System

The clock subsystem is responsible for providing the clocks for the device. In the case of the MSP432P401R, it is referred to as the Clock System (**CS**) by the reference manual and is shown in Figure 4.

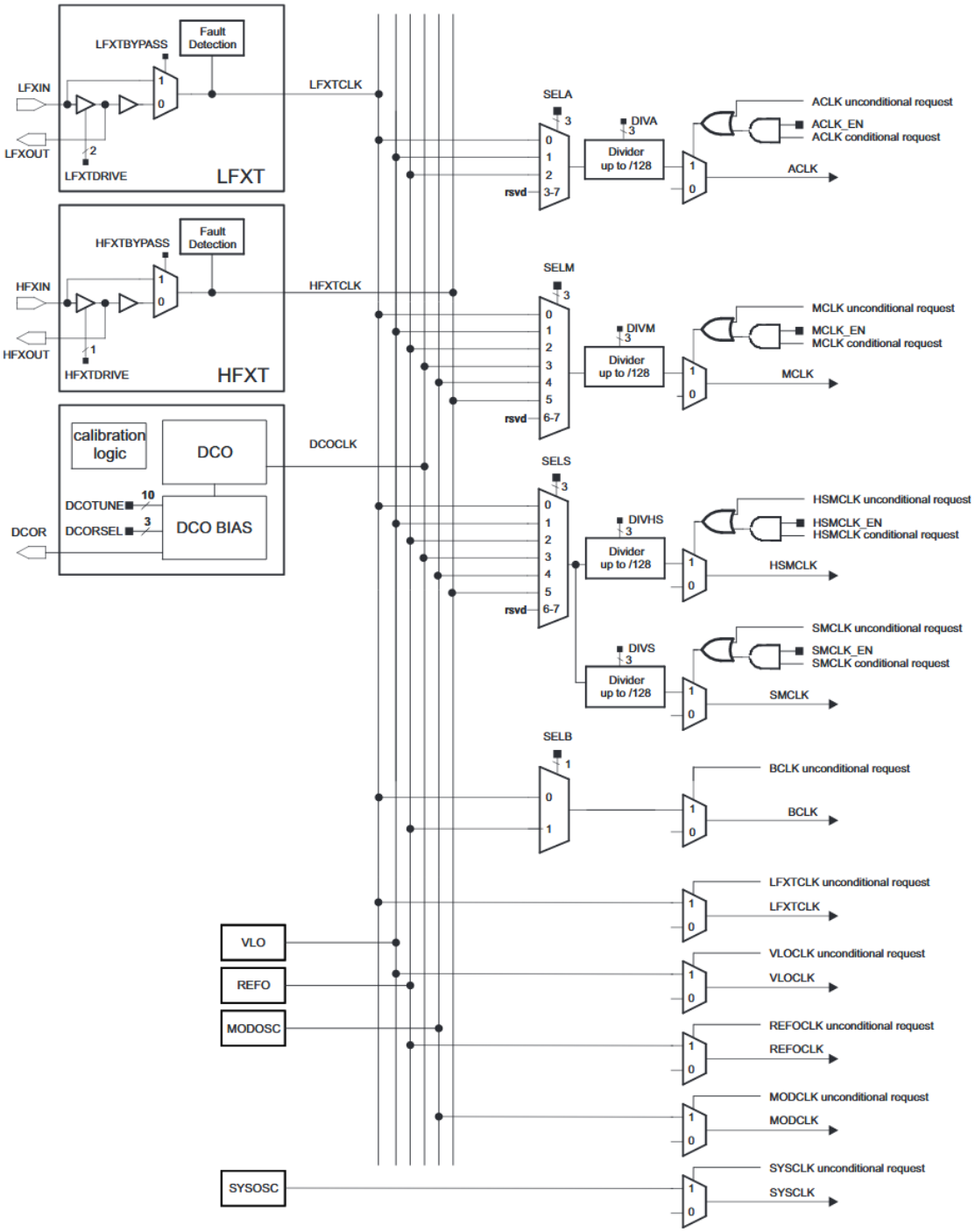

FIGURE 4. CLOCK SYSTEM BLOCK DIAGRAM

The **CS** features 7 physical clock sources. The most important ones for this lab are:

1. **HFXTCLK**: A high-frequency external oscillator that uses external resources (a 48MHz crystal on the Launchpad board).
2. **DCOCLK**: An internal Digitally-Controlled Oscillator (DCO).
3. **LFXTCLK**: A low-frequency oscillator used for LF external crystals (typically 32768 Hz)

Each of these physical clock sources can be used as the source of five clock signals (although, note that all bindings are not possible according to the schematic!):

1. **MCLK**, stands for Main clock, the clock used by the CPU and the system;
2. **SMCLK**, stands for Sub-System Master Clock;
3. **ACLK**, stands for Auxiliary clock;
4. **HSMCLK**, stands for High-frequency Sub-System Master Clock;
5. **BCLK**, stands for Back-up domain Clock;

SMCLK, HSMCLK and the **ACLK** can be selected to be used in certain subsystems, e.g., a timer.

The control signals in the Figure 4 can be modified by writing to the right registers. For example, the DCO can be configured using the **CTL0** register:

IMPORTANT

The **CS** registers are protected by a password to prevent from faulty overwrites. The right bits (see the device-specific datasheet) must be written in the **KEY** register *before* any change to the other **CS** registers. The value of the key is `0x0000695A` and is available in the **CS_KEY_VAL** macro.

GPIO

The LaunchPad board has 6 I/O ports. Each of these I/O ports can be used as a standard GPIO port, or can be configured as functional ports for various peripherals. The peripheral functions available with the MSP432P401R are stated in the following table. The precise description of the functionality of each pin can be found in the device-specific datasheet.

Port	Primary Function	Peripheral Functions
Port 1	I/O (P1.0 to P1.7)	Serial port
Port 2	I/O (P2.0 to P2.7)	Timer, Serial port
Port 3	I/O (P3.0 to P3.7)	Serial port
Port 4	I/O (P4.0 to P4.7)	ADC, external clocks
Port 5	I/O (P5.0 to P5.7)	ADC, ADC Ref, Timer
Port 6	I/O (P6.0 to P6.7)	ADC, Serial port

TABLE 1. ***PRIMARY AND SECONDARY FUNCTIONS FOR MSP432P401R PORTS***

Figure 5 below illustrates how a typical I/O port is organized inside the microcontroller, along with the registers that need to be configured to obtain the intended operation for each pin:

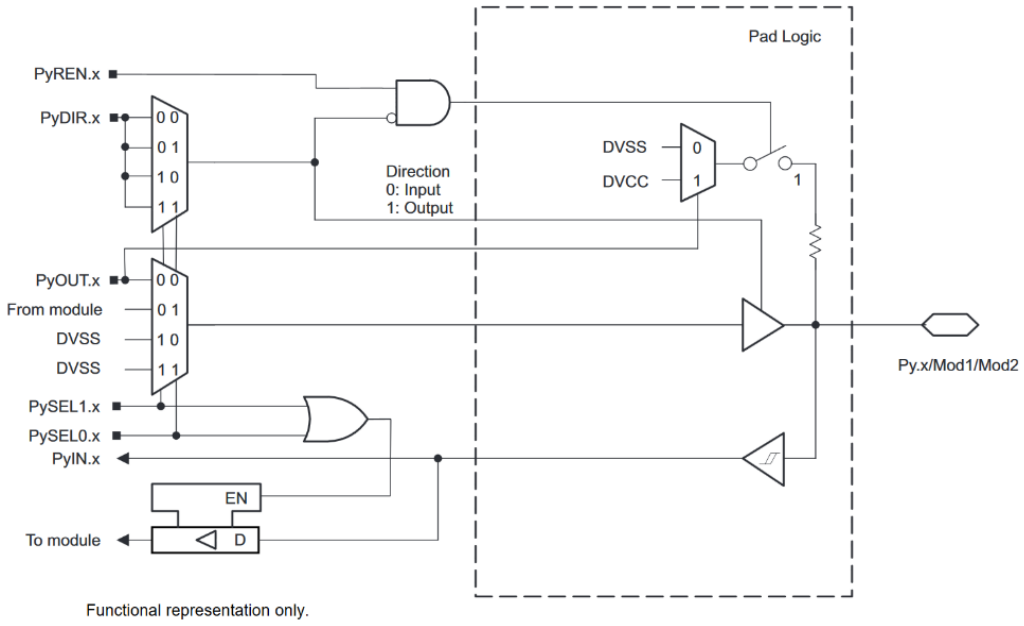

FIGURE 5. INTERNAL ARCHITECTURE OF PORT 2 ON THE MSP432P401R

Depending on the I/O port, several registers should be configured in order to achieve a specific function. The table below summarizes the main registers and their configuration.

Register	Description	Configuration
PyDIR.x	Direction Register – Input/Output	0→ Input, 1 → Output
PyIN.x	Read Value Register	0→ Low, 1 → High
PyOUT.x	Write Value Register	0→ Low, 1 → High
PySELz.x	Function Selection Register	0→ I/O, 1→ Peripheral

TABLE 2. PORT REGISTER CONFIGURATIONS. (Y) REPRESENTS A SPECIFIC PORT (P1, P2, …), AND (X) REPRESENTS THE BIT NUMBER OF THE PORT.

Exercise 1 – PWM on GPIO

- Using the LaunchPad board schematics and the TI MSP432 documentation, write a C program that generates a pulse width modulated (PWM) signal on one of board's available I/O ports.
- Test your solution with a logic analyzer or an oscilloscope.
- Test your solution by performing software measurements directly in your C code (try to count how many clock cycles are used to generate the PWM signal to find out its width).
- Compare the results you obtain through your software measurements with those you see on an oscilloscope/logic analyzer. Is the software measurement precise?

Exercise 2 – Strobe effect on GPIO

- Write a program to generate a rotating strobe effect ("chenillard" effect) on the LaunchPad. This effect should be done by rotating a '1' on Port **P4.0** to **P4.7**, or with the LEDs on Port **P2.0** to **P2.2**

Watchdog Timer

A watchdog timer is a timer that is used to tell if a system is hanging. It is basically a timer that resets the system unless the microcontroller pings the watchdog periodically to inform it that the system is not stuck (i.e., it has not reached an infinite loop). The watchdog timer is initialized during the microcontroller's power-up procedure and by default ***resets the CPU after ~10ms unless it is serviced***. In order to service the watchdog timer, a specific access must be performed before a programmable expiration time. *It is highly recommended to deactivate the watchdog timer for debugging purposes.*

The **WDTCTL** register is a "password-protected" register used to configure the watchdog timer. Any read/write operation to/from the **WDTCTL** register must be done using word instructions. Additionally, write accesses must include the right password 0x5A (***WDTPW***) in the upper byte. Check the MSP432 documentation for a description of the watchdog's registers and each of their uses. We show a few examples below:

```
; Stop the watchdog timer
WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;

; Some other useful selections:
; Periodically clear an active watchdog and specify the delay for next period
WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_IS_0 | WDT_A_CTL_CNTCL;

; Change watchdog clock source
WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_CNTCL | WDT_A_CTL_SSEL_SMCLK;
```

TimerA

The MSP432P401R has four 16-bit timers called TimerA. Each TimerA has 6 "capture-compare" registers (CCR).

- **TimerA0's** signals can be routed as follows:
  - **P2.4** to **P2.7** (respectively **TA0.1** to **TA0.4**)
- **TimerA2**'s signals can be routed as follows:
  - **P4.2** (**TA2CLK**)
  - **P5.6** (**TA2.1**)
  - **P6.6 and P6.7** (respectively **TA2.3** and **TA2.4**)

When configured in "output mode", the timer can directly drive a specific GPIO port. The ports made available to the timer depends on how the microcontroller is wired on a development board. You can find the pinout of the MSP432P401R in Figure 6. Note that a specific bit in the GPIO peripheral's **SELx** register must be programmed so the GPIO port is driven by the timer (and not in standard GPIO mode).

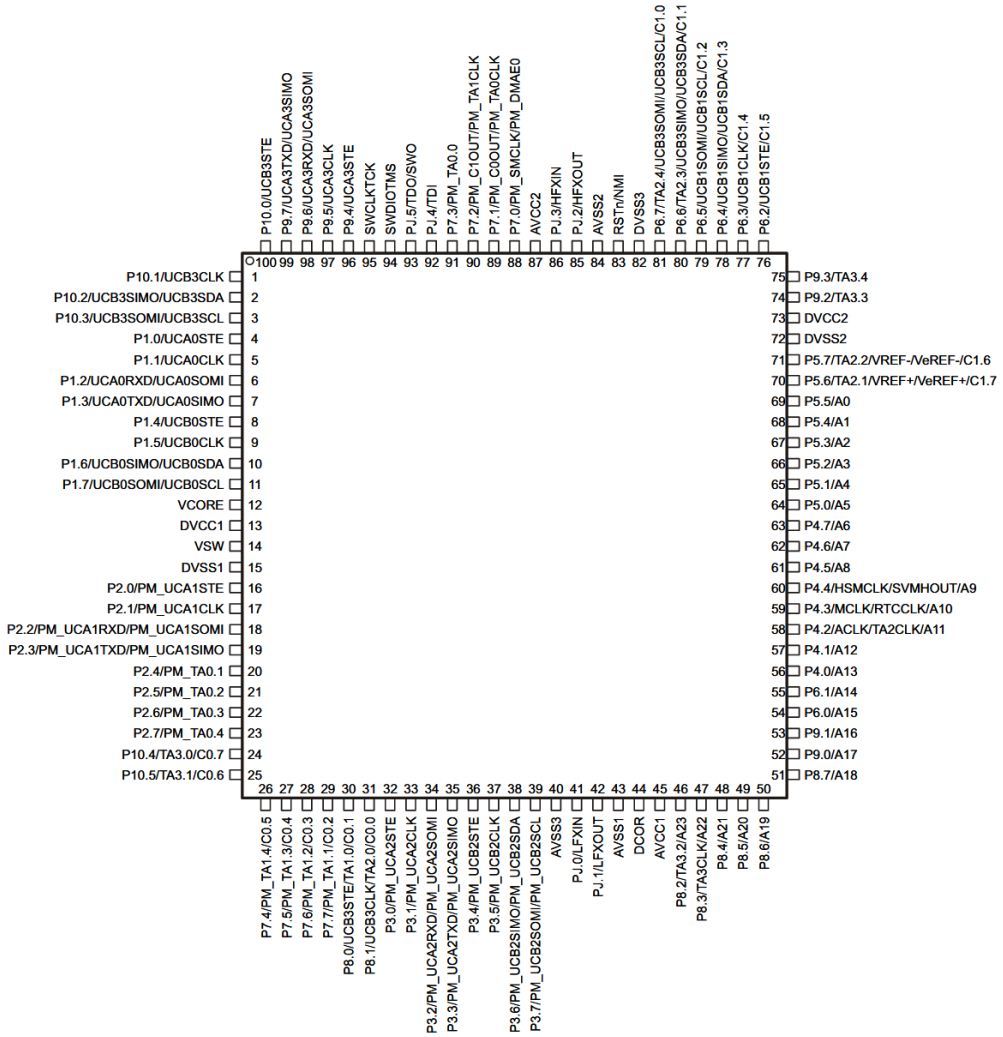

FIGURE 6. PINNING OF THE MSP432P401R

The main block of the Timer Module is a 16-bit free running counter **TAxR** that can be configured to count up or down. The **TAxCCRy** register is used to compare a desired value with the free running counter (0xFFFF is the maximum upper value).

The **TAxCCRy CCIFG** flag is used to indicate when the counter has reached the desired value, and could generate an interruption if properly configured. Figure 7 below shows the general architecture of the TimerA unit:

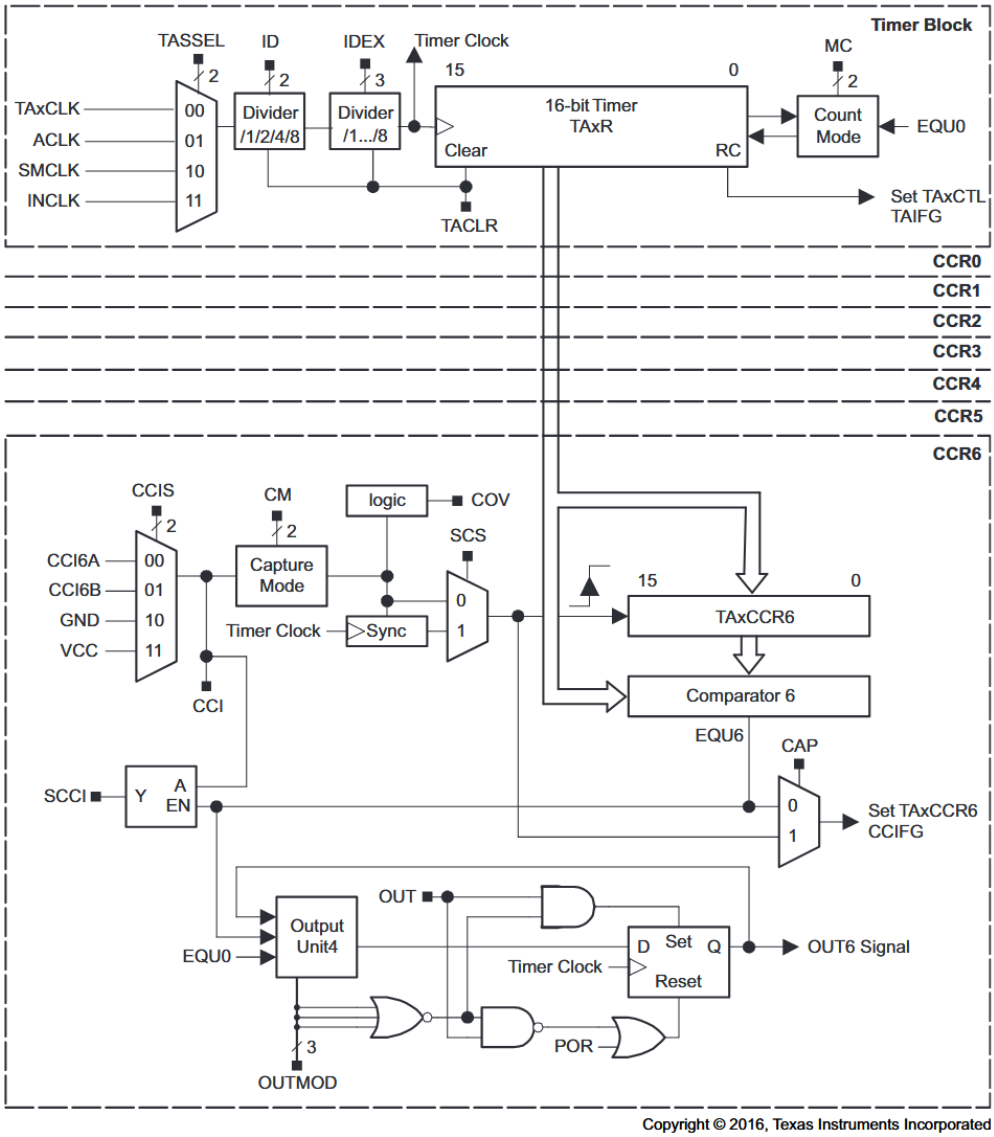

FIGURE 7. TIMERA BLOCK DIAGRAM

You can easily program the timer to implement a delay by using the compare function. Clock dividers can be used to extend/shrink the desired counting range.

Exercise 3 – Use TimerA to create a delay

- Write a function that takes a delay in ms as an input argument, and which causes the microcontroller to wait for the programmed time. Use **TimerA0**'s **Compare** functionality and don't forget to correctly program the **TAxCCR** register and actively poll the **CCIFG** flag!

Exercise 4 – Use TimerA to generate a PWM signal

- Use **TimerA0** to generate a PWM pulse on a GPIO port using the CCR comparator. The PWM pulse must have a period of ~20ms. Use an oscilloscope to view and validate your result.

Interrupt Management in the Cortex M4 Architecture

The Cortex M4F ARM core embeds hardware resources to handle nested interrupts, which means that it is able to efficiently handle the case where a more important interrupt is triggered while a lower priority is already being serviced.

Every interrupt is associated with a programmer-determined priority using a dedicated hardware unit called the *Nested Vectored Interrupt Controller* (NVIC). This unit chooses which interrupt to trigger and whether an interrupt routine can be stopped by incoming interrupt or not.

When an exception is triggered, the processor must push its current register values on the stack and jump to the right interrupt service routine. The NVIC manages the transition from an interrupt to another in a more efficient way, as this is a costly and critical operation for real-time systems.

Interrupt vector table

The interrupt service routines are listed in memory in a fixed format. When an interrupt is triggered and the priorities are checked by the NVIC, the processor jumps to the routine specified at the address offset corresponding to interrupt source. The `startup_msp432p401r_ccs.c` file defines the vector table and ensures that the table is located at address 0x00000000.

```
110 #pragma RETAIN(interruptVectors)
111 #pragma DATA_SECTION(interruptVectors, ".intvecs")
112 void (* const interruptVectors[])(void) =
113 {
114     (void (*)(void))((uint32_t)&__STACK_END),
115                                             /* The initial stack pointer */
116     Reset_Handler,                          /* The reset handler         */
117     NMI_Handler,                            /* The NMI handler           */
118     HardFault_Handler,                      /* The hard fault handler    */
119     MemManage_Handler,                      /* The MPU fault handler     */
120     BusFault_Handler,                       /* The bus fault handler     */
121     UsageFault_Handler,                     /* The usage fault handler   */
122     0,                                      /* Reserved                  */
123     0,                                      /* Reserved                  */
124     0,                                      /* Reserved                  */
125     0,                                      /* Reserved                  */
126     SVC_Handler,                            /* SVCall handler            */
127     DebugMon_Handler,                       /* Debug monitor handler     */
128     0,                                      /* Reserved                  */
129     PendSV_Handler,                         /* The PendSV handler        */
130     SysTick_Handler,                        /* The SysTick handler       */
131     PSS_IRQHandler,                         /* PSS Interrupt             */
132     CS_IRQHandler,                          /* CS Interrupt              */
133     PCM_IRQHandler,                         /* PCM Interrupt             */
134     WDT_A_IRQHandler,                       /* WDT_A Interrupt           */
135     FPU_IRQHandler,                         /* FPU Interrupt             */
136     FLCTL_IRQHandler,                       /* Flash Controller Interrupt*/
137     COMP_E0_IRQHandler,                     /* COMP_E0 Interrupt         */
138     COMP_E1_IRQHandler,                     /* COMP_E1 Interrupt         */
139     TA0_0_IRQHandler,                       /* TA0_0 Interrupt           */
140     TA0_N_IRQHandler,                       /* TA0_N Interrupt           */
```

FIGURE 8. INTERRUPT VECTOR TABLE DECLARED IN STARTUP_MSP432P401R_CCS.C

To set up an Interrupt Service Routine, it is possible either to replace the corresponding ISR in the *interruptVectors[ ]* array in this file with the name of your custom ISR, or to define an ISR with the same name as the pre-defined ISR. For example, to handle the interrupt corresponding to the watchdog timer, you should declare an ISR as:

```
void WDT_A_IRQHandler(void)
```

NVIC configuration

The NVIC must be configured to enable an interrupt and to set its priority. By default, all the interrupts are set to priority 0 (the highest priority). The bits of registers **ISER0** and **ISER1** allow enabling interrupts 0 to 31 and 31 to 63 respectively. Registers **IPR0** to **IPR15** can, in turn, be used to define the priority of each interrupt.

The following functions can be helpful for configuring the NVIC for a specific peripheral.

```
NVIC_EnableIRQ(TA3_0_IRQn);
NVIC_SetPriority(TA3_0_IRQn,4);
```

Exercise 5 – TimerA0 interrupts

- Use **TimerA0** to generate periodic interrupts every ~50ms. Toggle a GPIO pin on each interrupt. Use a logic analyzer to view and validate the results.

ADC

The MSP432P401R can perform a 14-bit analog-to-digital conversion. The programmable module responsible for this is referred to as the **ADC14** peripheral. Its block diagram is depicted in Figure 9.

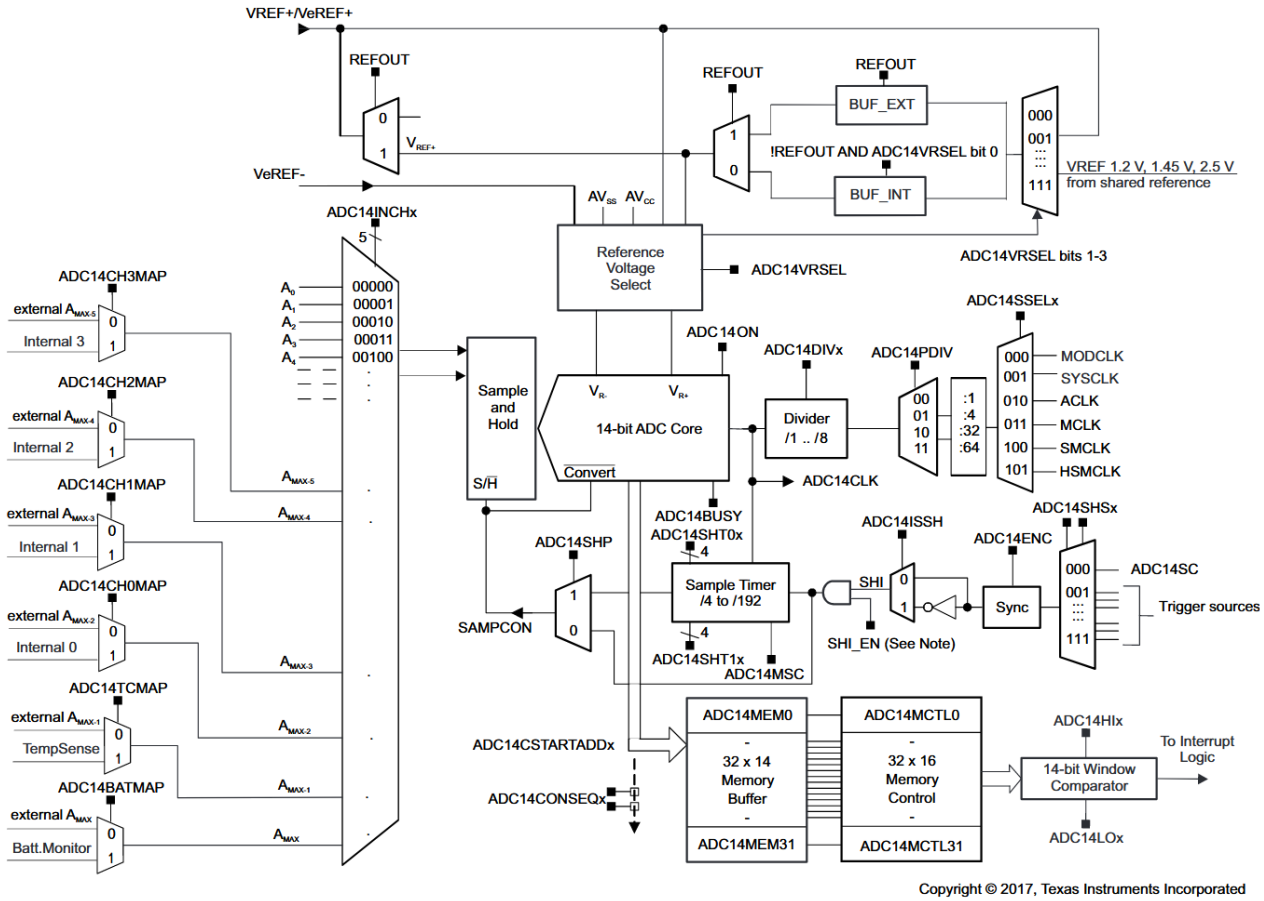

FIGURE 9. ADC14 BLOCK DIAGRAM

The **ADC14** basically functions as follows:

- Configure a pin to be used as an analog input to the ADC. For example, one can map **P4.0** to **A13** using the **PxSEL0** register.
- Select the specific input to sample by setting the **INCHx** bits.
- Select the **MEM** register to which the digital value should be written. You can choose one of the **ADC14MEMx** registers.
- At the rising edge of the **SHI** signal, a sampling stage will be initiated. Then, depending on how the **Sample Timer** is configured by the programmer, the **SAMPCON** signal is held high during a certain period (in function of the period of **SHI**). The **SAMPCON** signal determines how long the analog signal must be sampled.
- As soon as **SAMPCON** goes low, the conversion stage is initiated and will last 16 **ADC14CLK** cycles.
- Finally, the sampled value will be available in the **ADC14MEM** register.

Figure 10 shows the small PS2-style joystick we will use as our analog input.

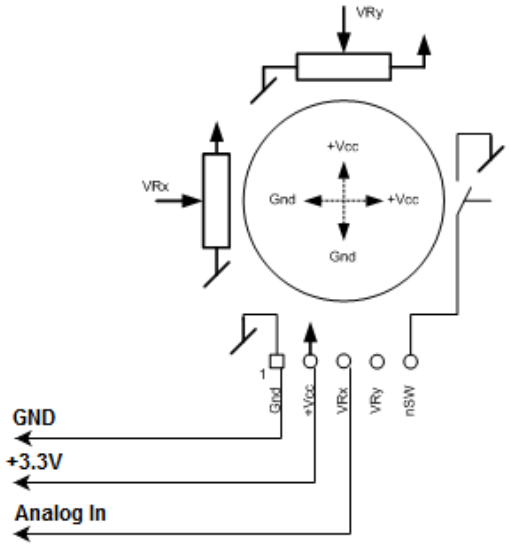

FIGURE 10. JOYSTICK BLOCK DIAGRAM

EVEN IF THE SUPPLY VOLTAGE WRITTEN ON THE PCB OF THE JOYSTICK IS +5V, THIS PIN MUST BE TIED TO +3.3V AS THE SUPPLY VOLTAGE OF THE MICROPROCESSOR IS 3.3V

Exercise 6 – ADC

- Write a function that uses the **ADC14** module to acquire an analog signal obtained from an external potentiometer. To plug in the potentiometer, refer to figure and the explanations provided in the next section.

Lab 1.0 – Final task

We now arrive at the final task of this lab. The goal is to implement the system shown in Figure 11 where an ADC is used to convert the analog value of a joystick in to a digital signal that is then used to generate a

proportionally sized PWM signal that turns a servomotor's head. The PWM output should satisfy the timings depicted in Figure 12 for the servomotor to function correctly.

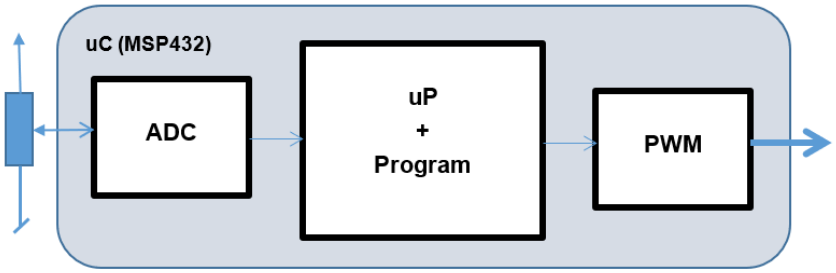

FIGURE 11. END-TO-END ACQUISITION AND PWM GENERATION

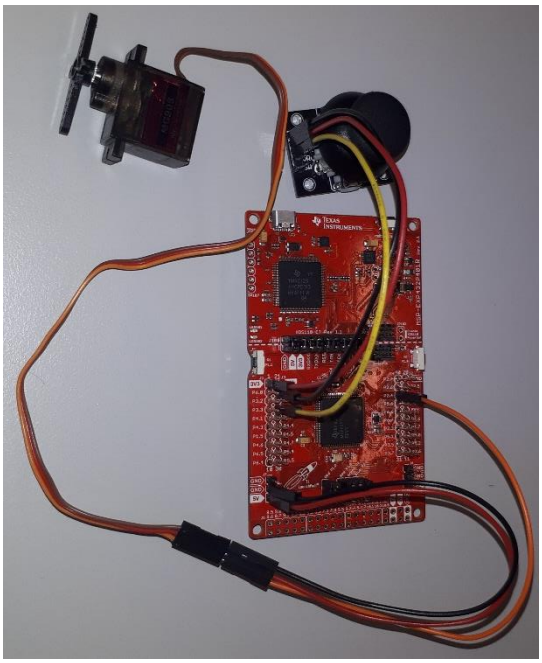

FIGURE 12. FINAL ASSEMBLY WITH JOYSTICK AND SERVOMOTOR

Figure 12 above shows an example of how such a system would look.

- A Joystick is connected such that its VCC pin matches one of the +3.3V power pins of the LaunchPad board and its output is tied to **P4.0** (which can be configured to be the analog input of the **ADC14**).
- A servomotor is connected to the GND and +5V pins of the Launchpad at the bottom right of the board. The servomotor is controlled via PWM to satisfy the timings shown in Figure 13.
- The orange wire is the input of the servomotor and should be tied to a pin that outputs the PWM generated by your timer (for example **P2.4**). The width of the pulse controls the angle of the motor.
- **WARNING: THE BLACK WIRE OF THE SERVOMOTOR MUST BE CONNECTED TO GND AND THE RED ONE TO VCC! THE SERVOMOTOR CAN BE DAMAGED IF NOT PLUGGED IN CORRECTLY.**

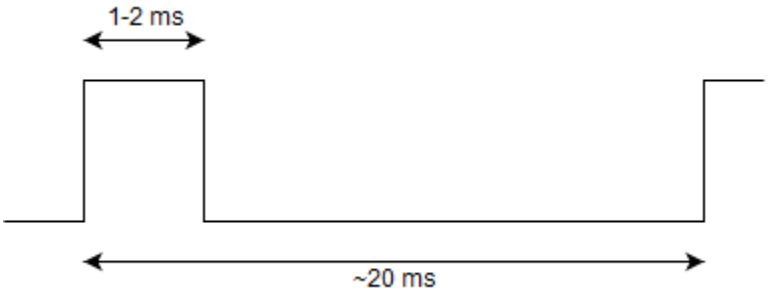

FIGURE 13. PWM TIMING

Exercise 7 – Timer, ADC, PWM, GPIO and interrupts

- Use a timer interrupt to periodically enable the ADC converter in **software** (through an ISR) and to start a conversion of the joystick value.
- Use another interrupt from the **ADC14** to catch the sampled value and use it to adjust the duty cycle of your PWM signal. If your system is working correctly, the servomotor should move proportionally to the movement of the joystick.
- Use an oscilloscope/logic analyzer to confirm that your PWM signal is generated correctly.
- Give a demo of your working system to an assistant.