MSP432 I/O

LaunchPad
MSP432 Laboratory

Goal	Understand the operation of the MSP432 peripherals
Resource	MSP432P401R Microcontroller
Prerequisites	MSP432 Base Course
Theory	
Equipment	➢ MSP432P401R-LaunchPad board ➢ Code composer Studio cross development tools
Duration	~6h

1 Introduction

The objective of this laboratory is to understand how to operate some of the programmable interfaces available on a microcontroller (specifically on the MSP432 family, available on the TI LaunchPad board).

This laboratory is divided into 3 sessions, and the final experiment is to be able to convert an analog signal to a digital one using the Analog to Digital (A/D) converter available on the LaunchPad.

The microcontroller should output a Pulse Width Modulated (PWM) signal with a width that is proportional to the provided analog input. An oscilloscope and/or a logic analyzer will be used to display the PWM output as well as other useful signals.

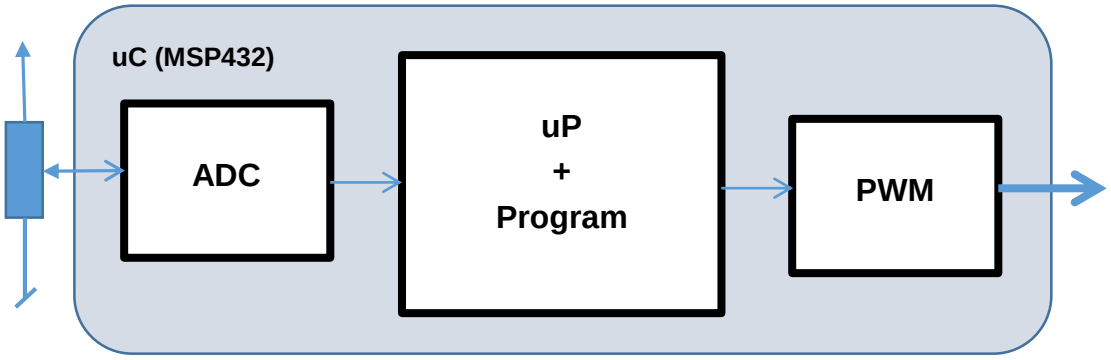

Figure 1. General system block schematic, internal ADC

1.1 Getting Started

TI's processors and microcontrollers documentations are often composed of two datasheets:

- A Family reference manual[1] that describes the µC/µP's core and peripherals functionality in details.
- A Device-specific Datasheet[2] 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 µC properly.
You may also need the Launchpad Development User guide [3]to check the pinout of the board, external crystals, etc…

The low level software development on microcontrollers and microprocessors relies on writing the desired values in the right memory space to communicate with the hardware resources.

Most of the time, in addition with the µC/µP, the manufacturers provide the user with a *Hardware Abstraction Level* library that allows fast development by abstracting the low-level hardware considerations with high level functions.

However, as a deep understanding of low level development is required for the following labs, you are asked NOT TO USE the HAL libraries ☺

TI provides the programmer with a Device Library which defines constants and data structures, allowing to write more readable code. You are strongly recommended to use this library, as such code is way easier to debug!

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

This library mainly defines C structures for each peripheral and convenient names for each bit of a register associated to a given peripheral:

```
466 /**************************************************************************
467 * CS Registers
468 *************************************************************************/
469 /** @addtogroup CS MSP432P401R (CS)
470   @{
471 */
472 typedef struct {
473   __IO uint32_t KEY;                /*!< Key Register */
474   __IO uint32_t CTL0;               /*!< Control 0 Register */
475   __IO uint32_t CTL1;               /*!< Control 1 Register */
476   __IO uint32_t CTL2;               /*!< Control 2 Register */
477   __IO uint32_t CTL3;               /*!< Control 3 Register */
478        uint32_t RESERVED0[7];
479   __IO uint32_t CLKEN;              /*!< Clock Enable Register */
480   __I  uint32_t STAT;               /*!< Status Register */
481        uint32_t RESERVED1[2];
482   __IO uint32_t IE;                 /*!< Interrupt Enable Register */
483        uint32_t RESERVED2;
484   __I  uint32_t IFG;                /*!< Interrupt Flag Register */
485        uint32_t RESERVED3;
486   __O  uint32_t CLRIFG;             /*!< Clear Interrupt Flag Register */
487        uint32_t RESERVED4;
488   __O  uint32_t SETIFG;             /*!< Set Interrupt Flag Register */
489        uint32_t RESERVED5;
490   __IO uint32_t DCOERCAL0;          /*!< DCO External Resistor Cailbration 0 Register */
491   __IO uint32_t DCOERCAL1;          /*!< DCO External Resistor Calibration 1 Register */
492 } CS_Type;
```

Figure 2. Structure declaration in the library "msp432p401r.h"

For example, if you want to write the password to the Clock System peripheral, you could use:

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

[1] http://www.ti.com/lit/ug/slau356h/slau356h.pdf
[2] http://www.ti.com/lit/ds/symlink/msp432p401r.pdf
[3] http://www.ti.com/lit/ug/slau597f/slau597f.pdf

CS is a CS_Type object also defined in "*msp432p401r.h"*, and CS_KEY_VAL is the value of the password.

1.2 Clock

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 Fig.2.

It features seven physical clock sources, and 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 DCO (https://en.wikipedia.org/wiki/Digitally_controlled_oscillator); and
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 binding 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.

To save energy, the clock signals are only sent to the peripherals when one of them needs it. This is done by the Module Clock Request System: When a sub-unit is routed to a clock signal, it sends a *Conditional Clock request* that is then used by the Clock System to send the signal.

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. It is also recommended to write any value in the KEY register when the Clock system is properly configured to protect the system.

The clock selection logic is outlined in the schematic on the next page, and is detailed in the **MSP432P401R** User guide. Make sure you feel comfortable with the control registers of the clock system, i.e. the **CTL0** and **CTL1** registers.

The value of the key is *0x0000695A.*

The control signals in the figure 2 can be modified by the user by writing to the right registers. For exemple, the configuration of the DCO can be done using the **CTL0** register:

Bit	31-19	18-16	15-10	0-9
Signal	…	DCORSEL	Reserved	DCOTUNE

The full description of the Clock system registers can be found in the reference manual.

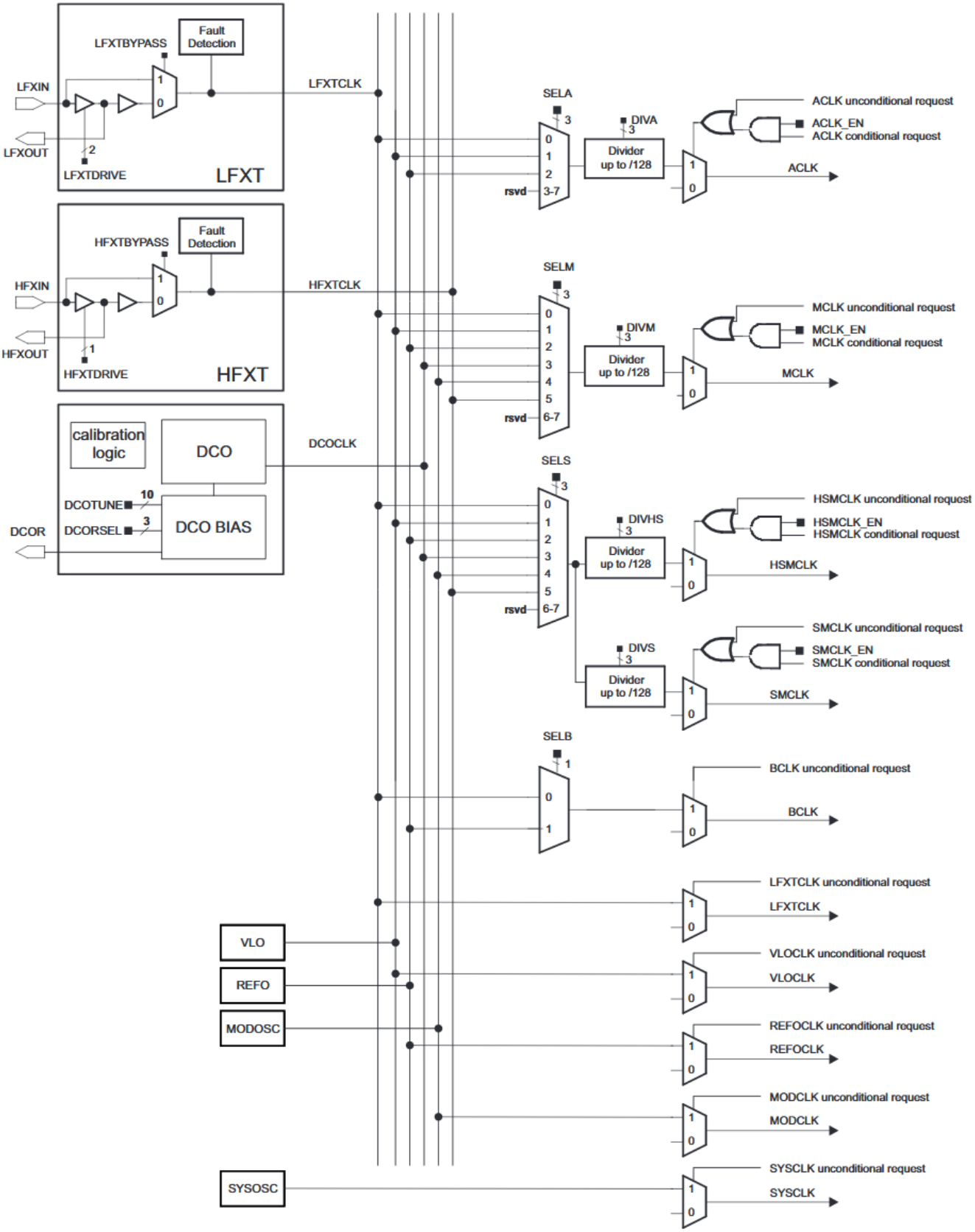

Figure 2. Clock System Block Diagram of the MSP432Px family

1.3 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

Port secondary functions for MSP432P401R

Figure 3 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:

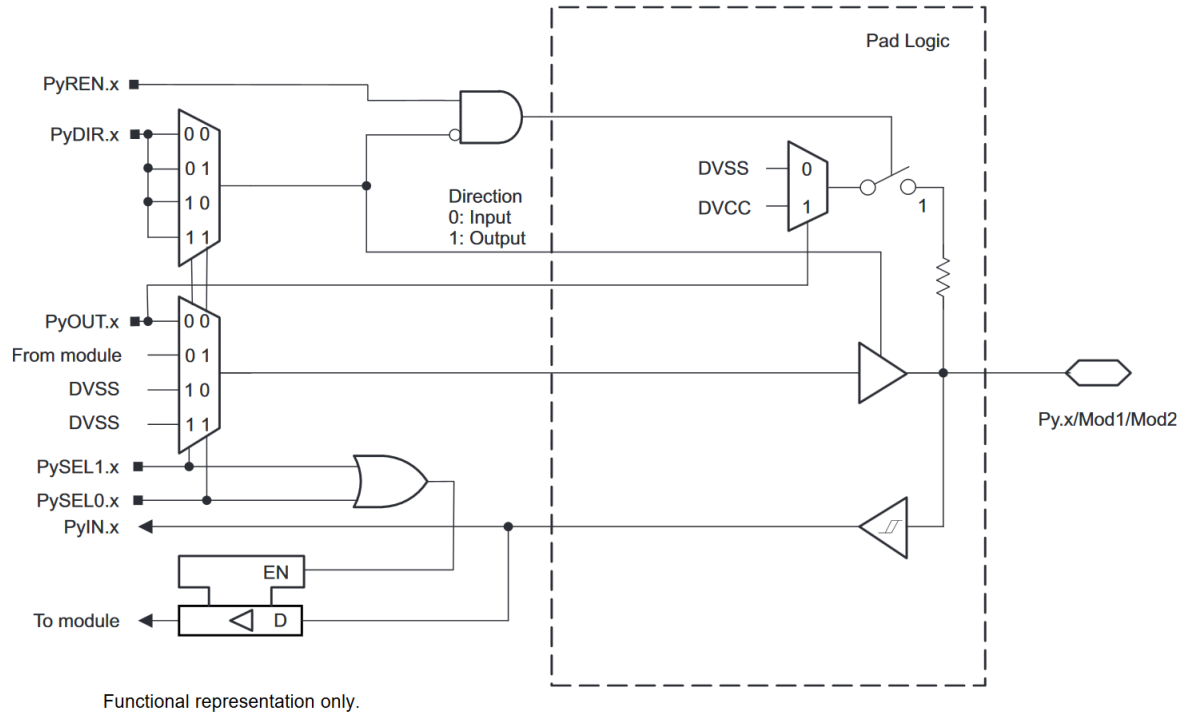

Figure 3. Internal architecture of the Port 2 (MSP432P401R)

Depending on the I/O port, several registers should be configured in order to achieve the desired 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

** In the table above, (y) represents a specific register (for Port 1, P1), and (x) the bit number of the port*

Manipulation 1 ***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 clock cycles 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.

Manipulation 2 ***GPIO - Chenillard***

- 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**

1.4 Watchdog Timer

A watchdog timer is initialized during the power-up procedure. The watchdog timer will ***reset the CPU after ~10 ms 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 microcontroller's registers and each of their uses

```
; 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;
```

1.5 Timer

The **MSP432P401R** has four 16-bit timers (4 x TimerA with 6 CCR each):

- **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**)

To use the Timer in output mode, the corresponding bit in the GPIO **SELx** register must be programmed for the specific peripheral mode wanted (and not in GPIO mode). Refer to the device-specific datasheet for the pin-function equivalence.

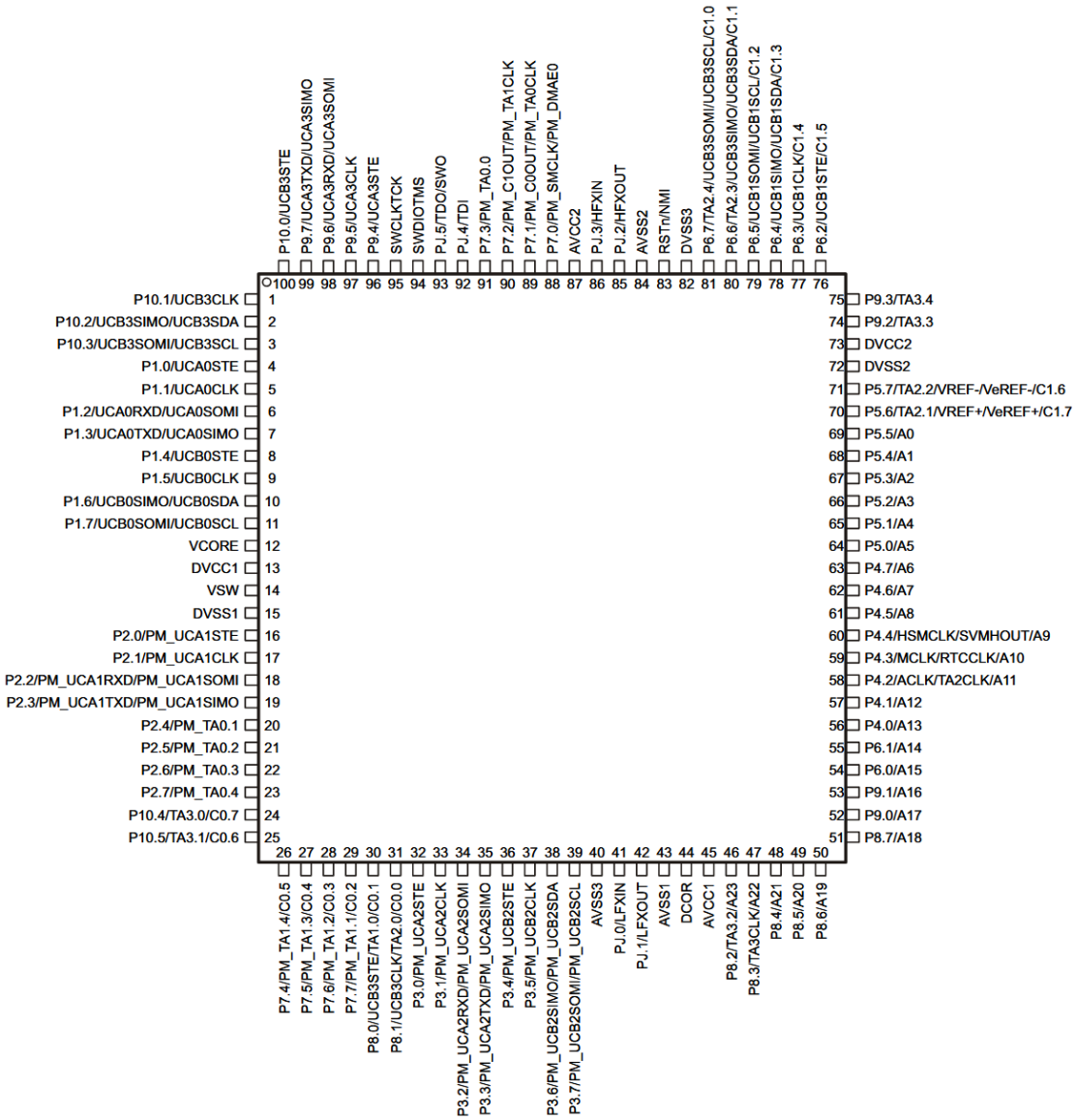

Figure 4. Pinning of MSP432 P401R, 100 pins

1.5.1 TimerA used as a counter

The main block of the Timer Module is a 16-bit free running counter that can be configured to count up or down (**TAxR**). 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 5 below shows the general architecture of the TimerA unit:

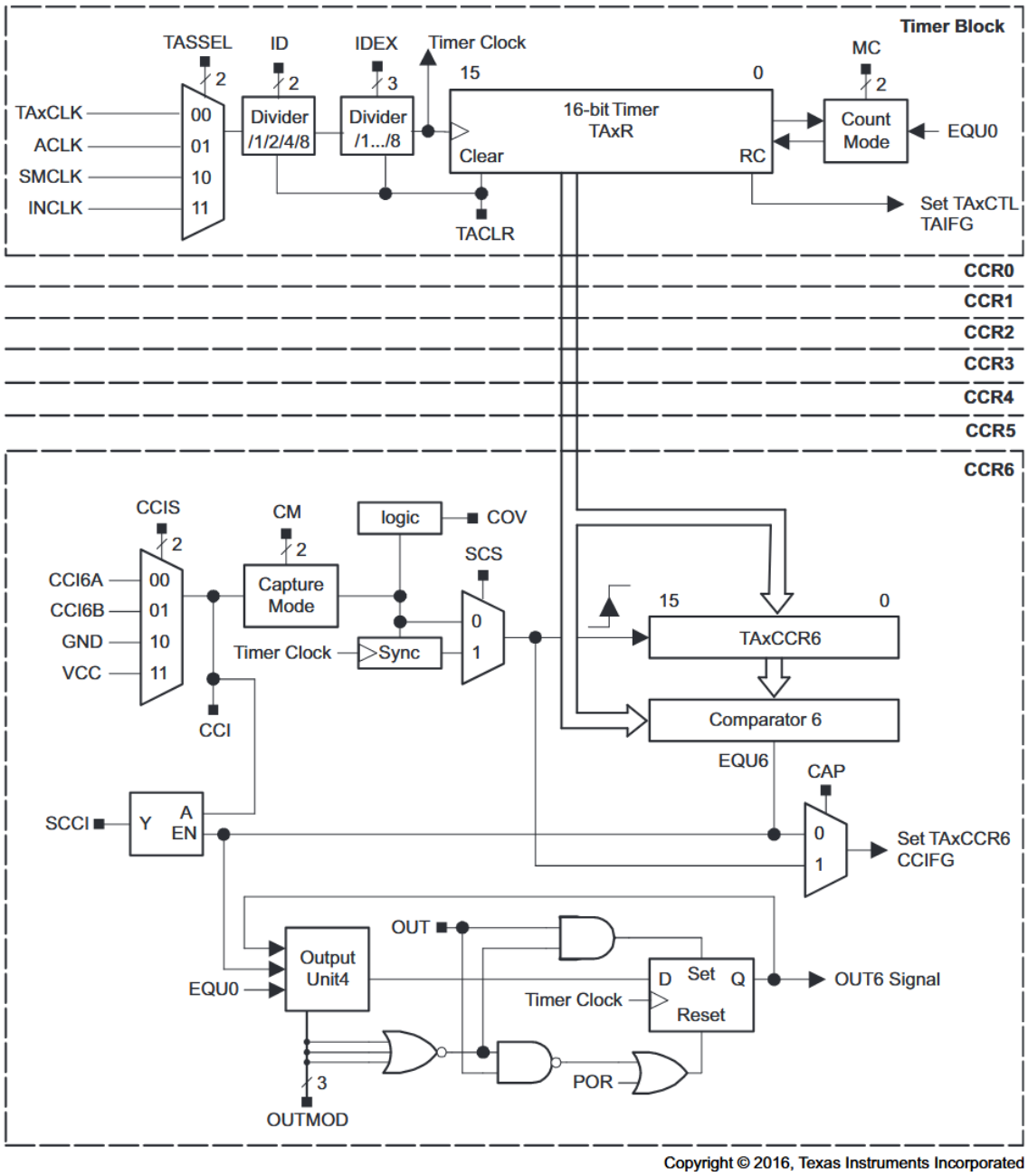

Figure 5. TimerA block schematic (from TI)

You can easily program the timer with a delay by using the Compare function. The clock dividers can be used in order to achieve the desired counting range.

As an exercise, write a function that causes the microcontroller to wait for a certain delay in specified in *ms.*

Manipulation 3 ***TimerA0, delay***

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

1.5.2 PWM generation

Use **TimerA0** to generate a periodic pulse through pulse width modulation (PWM mode).

Write a function that generates a pulse with a period of ~20 [ms]. The pulse's duty cycle should be programmed as the function's parameter. Study the different modes available on **TimerA** to generate the PWM pulse.

You can find a block diagram of **TimerA** in Figure 5 above.

Manipulation 4 ***PWM with TimerA***

- Use **TimerA0** to generate a PWM pulse by configuring the CCR comparator to operate in the proper manner. The PWM pulse must have a period of ~20[ms]. Use an oscilloscope to view and validate the results.

1.6 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 handle efficiently the case where a more important interrupt is triggered when another one is being serviced.

Every interruption is associated to a priority determined by the programmer, using a dedicated hardware unit called Nested Vectored Interrupt Controller (NVIC). This unit chooses which interrupt can be triggered on request, and whether an interrupt routine can be stopped by another or not.

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

1.6.1 Interrupt vector table

The interrupt service routines are listed in memory in a fixed format according to their source. (See figure 6) 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.

Address offset	IRQ Source
0x00000000	Reset
0x00000004	NMI
0x00000008	Hard Fault
	...
0x00000060	Timer0 CCR0
0x00000064	Timer0 CCRN
0x00000068	Timer1 CCR0

In Code Composer Studio, the file *"startup_msp432p401r_ccs.c"* defines the vector table and ensures that the table is located at address 0x00000000. (See figure 7)

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 the ISR as:

```
void WDT_A_IRQHandler(void)
```

```
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 7. Interrupt Vector Table declared in "startup_msp432p401r_ccs.c"

1.6.2 NVIC configuration

The NVIC must be configured to enable an interruption and to set its priority.

By default, all the interruptions are set to the priority 0 (highest priority).

The bits of the registers **ISER0** and **ISER1** allow to enable the interruptions 0 to 31 and 31 to 63 respectively, and the registers **IPR0** to **IPR15** may be used to define the priority of each interruption.

Alternatively, you can use the functions:

```
NVIC_EnableIRQ(TA3_0_IRQn);

NVIC_SetPriority(TA3_0_IRQn,4);
```

Where *TA3_0_IRQn* is the IRQ ID defined in "*msp432p401r.h"*

1.7 TimerA0 interrupt-generation

It is possible to use **TimerA0** in Output compare mode to generate a periodic interrupt. A vector table contains the address of every interrupt routine that needs to be called for a specific Interrupt Request.

Manipulation 5	***Interruption on TimerA0***
➢ Use **TimerA0** to generate periodic interrupts every ~50ms. Toggle a GPIO pin on each interrupt. Use logic analyzer to view and validate the results.	

1.8 ADC

The **MSP432P401R** supports 14-bit analog-to-digital conversion. The programmable module responsible for this is referred to as the **ADC14** peripheral. Its block diagram is depicted below.

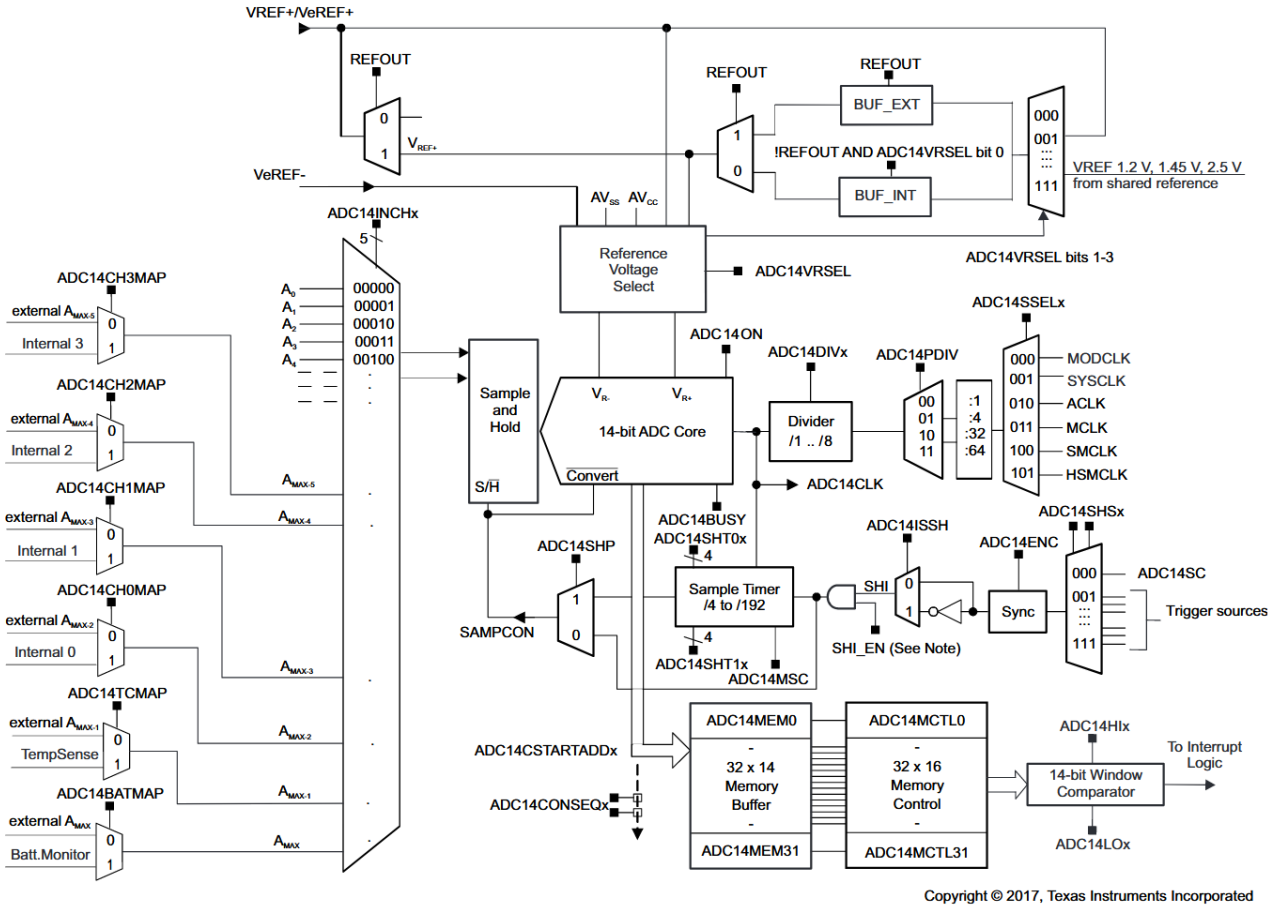

Figure 8. ADC14 bloc diagram (fromTI)

It basically works as follows:

- Pins can be configured as analog inputs to the **ADC14**. Using the **PxSEL0** register, as specified by the datasheet, one can for example map **P4.0** to **A13**.
- The inputs can be selected using the **INCHx** bits for a given **MEM** register with **ADC14MEMx** register (again, make sure to check the *ADC14 Registers* section of the user manual).
- 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 (i.e. you), 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.

Now it is your turn to configure all these registers, and don't forget to read the documentation ;) !

Manipulation 6 ***ADC, Analog to Digital Converter***

➢ 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.

1.8.1 ADC to control a servo-motor using PWM

The goal of this section is to use the sampled value obtained from the **ADC14** to control the duty cycle of your PWM. The A/D converter should be read every ~50ms. Use interrupts to meet this timing requirement.

The figure below shows an example configuration of the system for the 7th manipulation:

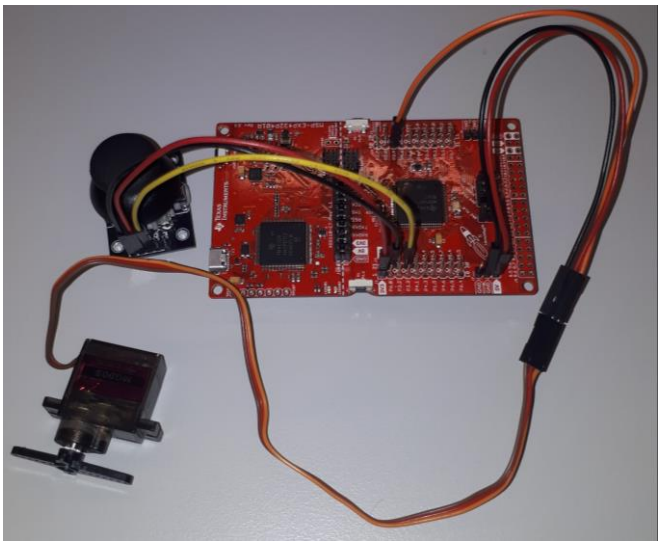

Figure 9. Kit connection with potentiometer and servomotor

A Joystick is plugged such that its VCC pin matches one of the +3.3V power pin of the LaunchPad board, its output is tied to **P4.0** (which can be configured to be the analog input of the **ADC14**), as shown in figure 10.

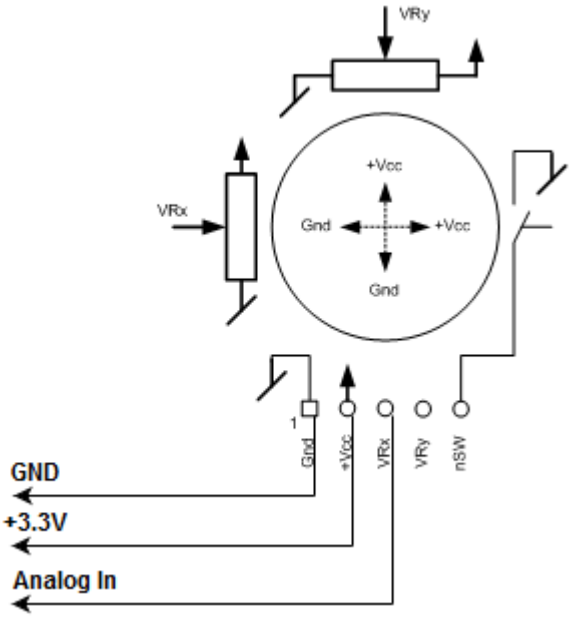

Figure 10. Connection of the Joystick to the board
Figure 11.

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

The 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 and should be configured as shown in the figure below.

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.

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:

1ms corresponds to 0° and 2ms correspond to α_{max} (maximum angle of the motor)

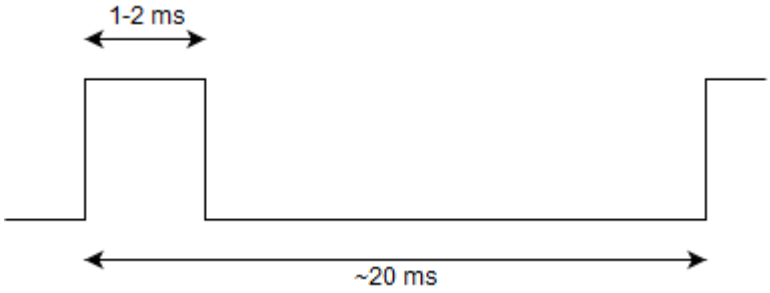

Figure 10.– PWM Pulse Width Modulation timing

Manipulation 7 ***Timer, ADC, PWM, GPIO and interrupts***

- Use a timer interrupt to periodically enable the ADC converter in **software** and to start a conversion of the potentiometer value.
- Use another interrupt from the (**ADC14** module this time) to catch the sampled value and use it to adjust the duty cycle of the pulses your pulse-width modulator generates.
- Make a demo to an assistant where you can visualize the result with an oscilloscope/logic analyzer and the servomotor.
- **Extra**: try to avoid the use of a **software** routine to enable the ADC conversion process, but instead connect the timer directly to the **ADC14** (find out where to perform the connection from the **ADC14** bloc diagram above).

1.8.2 *Optional:* Control the servo-arm using the joysticks

Using the same procedure as the previous section, try to write a program that controls the servo-arm.

This servo-arm has two servo-motors that are the same as the previous experiment, one of them controls the “pan” of the arm and the other controls the “tilt”.

This time, two joysticks can be used to control the two servo-motors.

You can use for example the analog input **A13, A11, A9**, and **A8**, on pins **P4.0, P4.2, P4.4, P4.5** respectively, and the pins **P2.4** and **P4.5** to drive the servo-motors.

Configure the **ADC14** unit properly to sample the 4 analog signals. You may choose to trigger an interrupt when the 4 signals are sampled.

You can ask the assistants to get one of such arms in class, have fun ☺