Tecnológico de Monterrey

Microcontrollers

Professor: Dr. Gilberto Ochoa Ruiz

Topic 7: Analog to Digital Converter

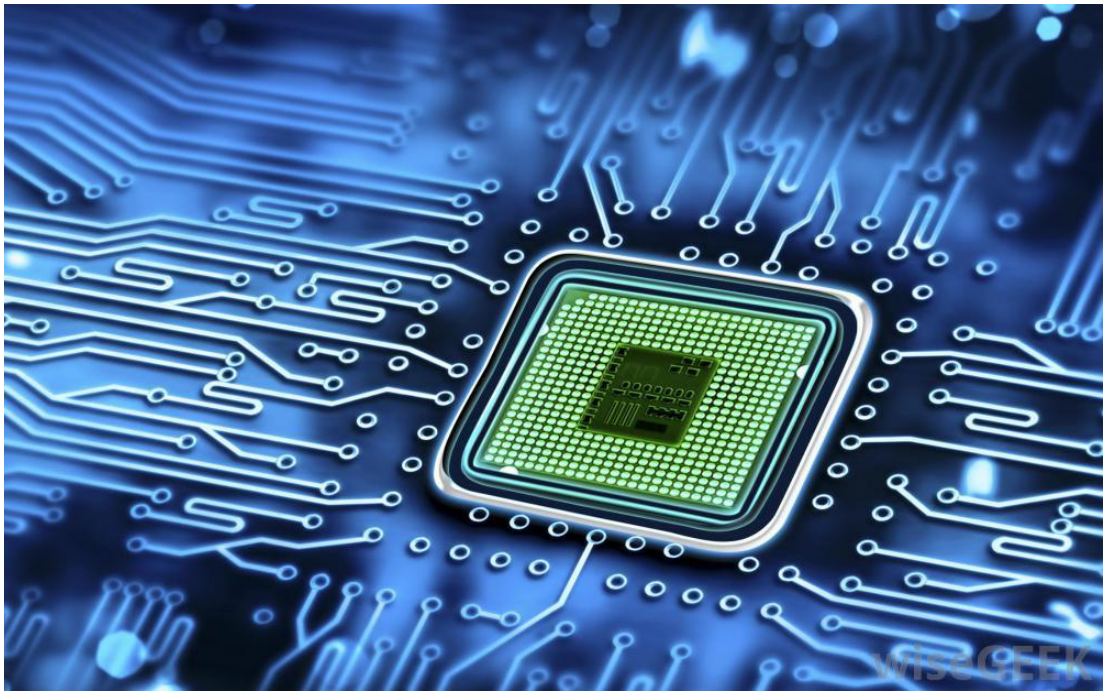
EIAD-204

Wednesday

May 27th 2020

6h30 – 9h30 PM

Guadalajara, Mexico



Basic Programming Techniques – ADC

ADC devices

Analog-to-digital converters are among the most widely used devices for data acquisition.

Digital computers use binary (discrete) values, but in the physical world everything is analog (continuous).

Temperature, pressure (wind or liquid), humidity, and velocity are a few examples of physical quantities that we deal with every day.

A physical quantity is converted to electrical (voltage, current) signals using a device called a *transducer*.

ADC devices

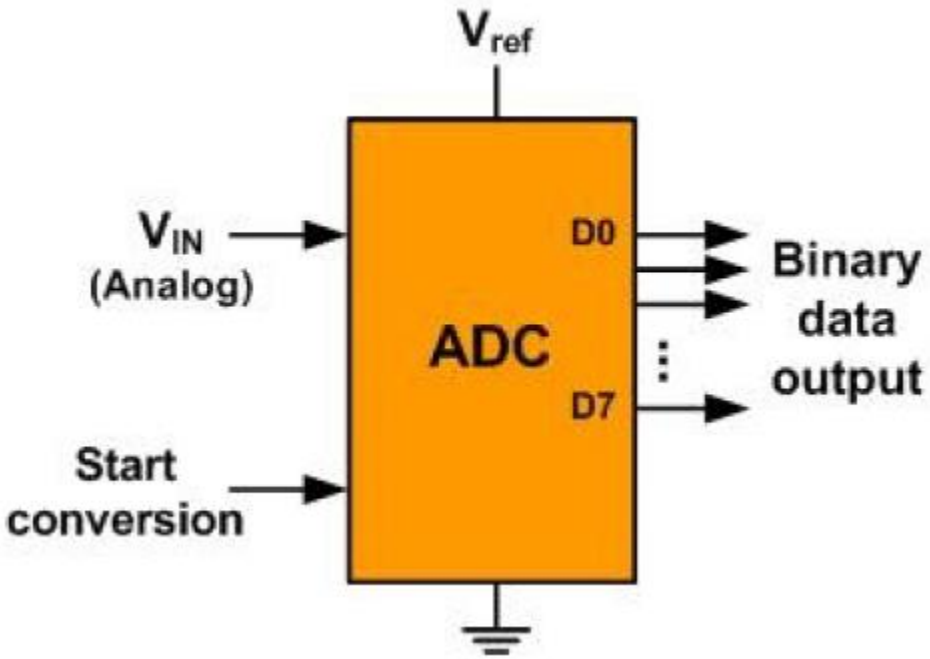
Transducers used to generate electrical outputs are also referred to as *sensors*.

Sensors for temperature, velocity, pressure, light, and many other natural physical quantities produce an output that is voltage (or current).

Therefore, we need an analog-to-digital converter to translate the analog signals to digital numbers so that the microcontroller can read and process the numbers.

ADC devices


Physical Signal
Sensor
Analog Signal
Signal Coditioning
Proper Voltage
ADC
Digital signal
CPU


V_{ref}
V_{IN} (Analog)
Start conversion
ADC
D0
D7
Binary data output

Major characteristics of ADCs - Resolution

The ADC has *n*-bit resolution, where *n* can be 8, 10, 12, 16, or even 24 bits. Higher-resolution ADCs provide a smaller step size, where *step size* is the smallest change that can be discerned by an ADC.

n-bit	Number of steps	Step size
8	256	5V /256 = 19.53 mV
10	1024	5V /1024 = 4.88 mV
12	4096	5V /4096 = 1.2 mV
16	65,536	5V /65,536 = 0.076 mV

Note: Vref = 5V

Major characteristics of ADCs - Resolution

Vref

Vref is an input voltage used for the reference voltage.

The voltage connected to this pin, along with the resolution of the ADC chip, determine the step size.

For an 8-bit ADC, the step size is Vref / 256 because it is an 8-bit ADC, and 2 to the power of 8 gives us 256 steps.

For example, if the analog input range needs to be 0 to 4 volts, Vref is connected to 4 volts. That gives 4 V / 256 = 15.62 mV for the step size of an 8-bit ADC.

Major characteristics of ADCs - Resolution

In another case, if we need a step size of 10 mV for an 8-bit ADC, then Vref = 2.56 V, because 2.56 V/ 256 = 10 mV.

For the 10-bit ADC, if the Vref = 5V, then the step size is 4.88 mV as shown the previous table.

The next tables show the relationship between the Vref and step size for the 8- and 10-bit ADCs, respectively.

In some applications, we need the differential reference voltage where Vref = Vref (+) – Vref (–). Often the Vref (–) pin is connected to ground and the Vref (+) pin is used as the Vref

Major characteristics of ADCs - Resolution

V_{ref} (V)	V_{in} in Range (V)	Step Size (mV)
5.00	0 to 5	5 / 256 = 19.53
4.00	0 to 4	4 / 256 = 15.62
3.00	0 to 3	3 / 256 = 11.71
2.56	0 to 2.56	2.56 / 256 = 10
2.00	0 to 2	2 / 256 = 7.81
1.28	0 to 1.28	1.28 / 256 = 5
1.00	0 to 1	1 / 256 = 3.90
Note: In an 8-bit ADC, step size is V_{ref}/256		

Tecnológico de Monterrey

Major characteristics of ADCs - Resolution

Vref (V)	V_{in} Range (V)	Step Size (mV)
5.00	0 to 5	5 / 1024 = 4.88
4.96	0 to 4.096	4.096 / 1024 = 4
3.00	0 to 3	3 / 1024 = 2.93
2.56	0 to 2.56	2.56 / 1024 = 2.5
2.00	0 to 2	2 / 1024 = 2
1.28	0 to 1.28	1.28 / 1024 = 1.25
1.024	0 to 1.024	1.024 / 1024 = 1
Note: In a 10-bit ADC, step size is $V_{ref}/1024$		

Major characteristics of ADCs – Conversion Time

In addition to resolution, conversion time is another major factor in selecting an ADC.

Conversion time is defined as the time it takes the ADC to convert the analog input to a digital number.

The conversion time is dictated by

the clock source connected to the ADC

in addition to the method used for data conversion and technology used in the fabrication of the ADC.

Major characteristics of ADCs – Digital Output

In an 8-bit ADC we have an 8-bit digital data output of D0–D7, while in the 10-bit ADC the data output is D0–D9.

To calculate the output voltage, we use the following formula:

DOUT = VIN / StepSize

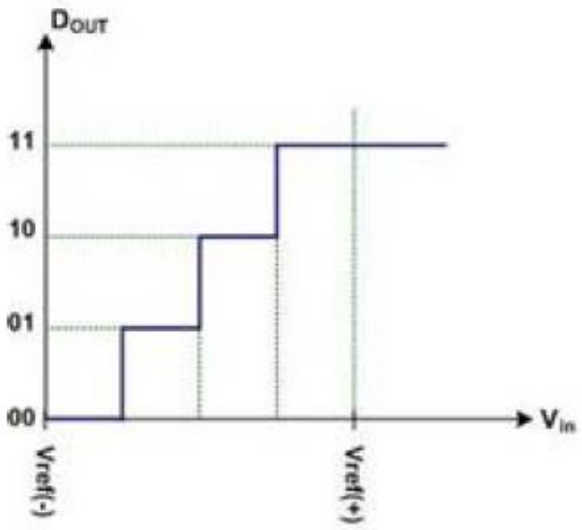
where

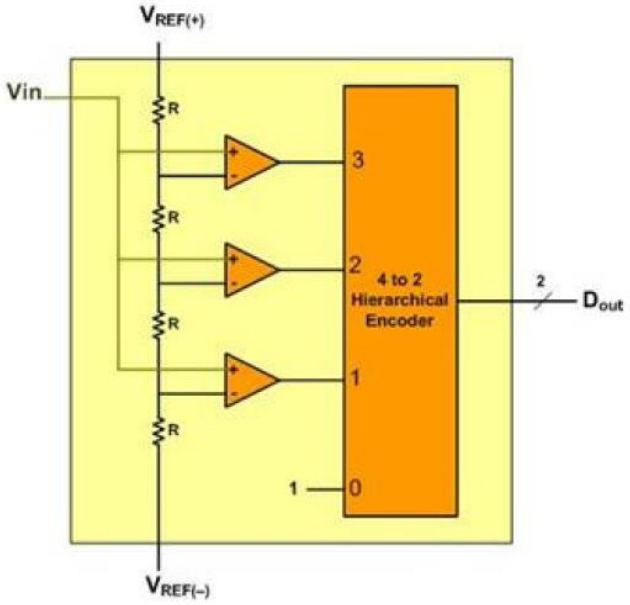
Dout = digital data output (in decimal)
Vin = analog input voltage,
and step size (resolution) is the smallest change, which is Vref/256 for an 8-bit ADC.

Major characteristics of ADCs – Digital Output

The figure shows a simple 2-bit ADC. In the circuit, the voltage between Vref(+) and Vref(-) is divided into 4 since resistors have the same values.

As a result, the step size is (Vref(+) - Vref(-)) / 4.





Major characteristics of ADCs – Digital Output

If Vin is below step size all the comparators send out zeros.

When Vin is between step size and step size × 2, the lowest comparator sends out 1 and the encoder gives 01.

If Vin is between step size × 2 and step size × 3, the second comparator and the first comparator sends out 1. Since the encoder is hierarchical priority, it sends out the highest value in cases that more than 1 input is high.

As a result, 2 (10 in binary) will be sent out.When Vin is bigger than step size × 3, the third comparator becomes high and 3 will be sent out.

Parallel versus serial ADC

The ADC chips are either parallel or serial.

In parallel ADC, we have 8 or more pins dedicated to bringing out the binary data, but in serial ADC we have only one pin for data out.

The D0–D7 data pins of the 8-bit ADC provide an 8-bit parallel data path between the ADC chip and the CPU. In the case of the 16-bit parallel ADC chip, we need 16 pins for the data path.

In order to save pins, many12- and 16-bit ADCs use pins D0–D7 to send out the upper and lower bytes of the binary data.

Parallel versus serial ADC

In recent years, for many applications where space is a critical issue, using such a large number of pins for data is not feasible.

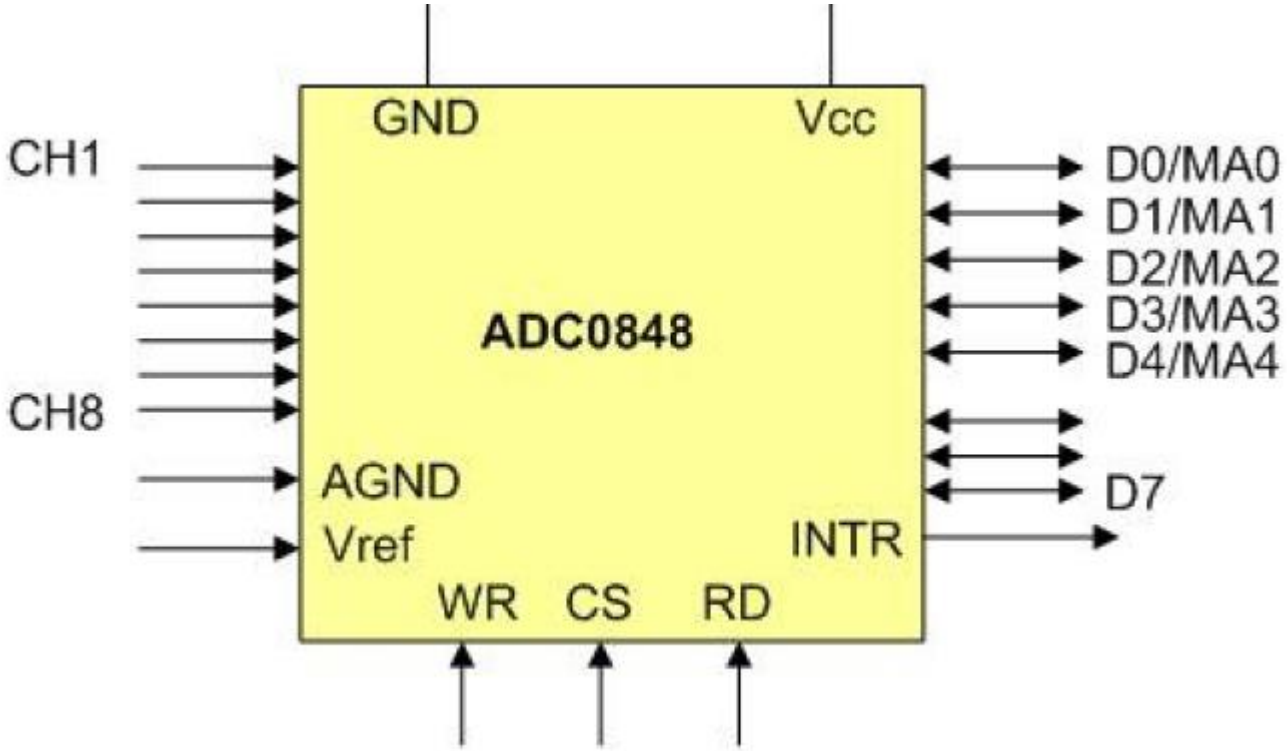
For this reason, serial devices such as the serial ADC are becoming widely used.

While the serial ADCs use fewer pins and their smaller packages take much less space on the printed circuit board

More CPU time is needed to get the converted data from the ADC because the CPU must get data one bit at a time, instead of in one single read operation as with the parallel ADC.

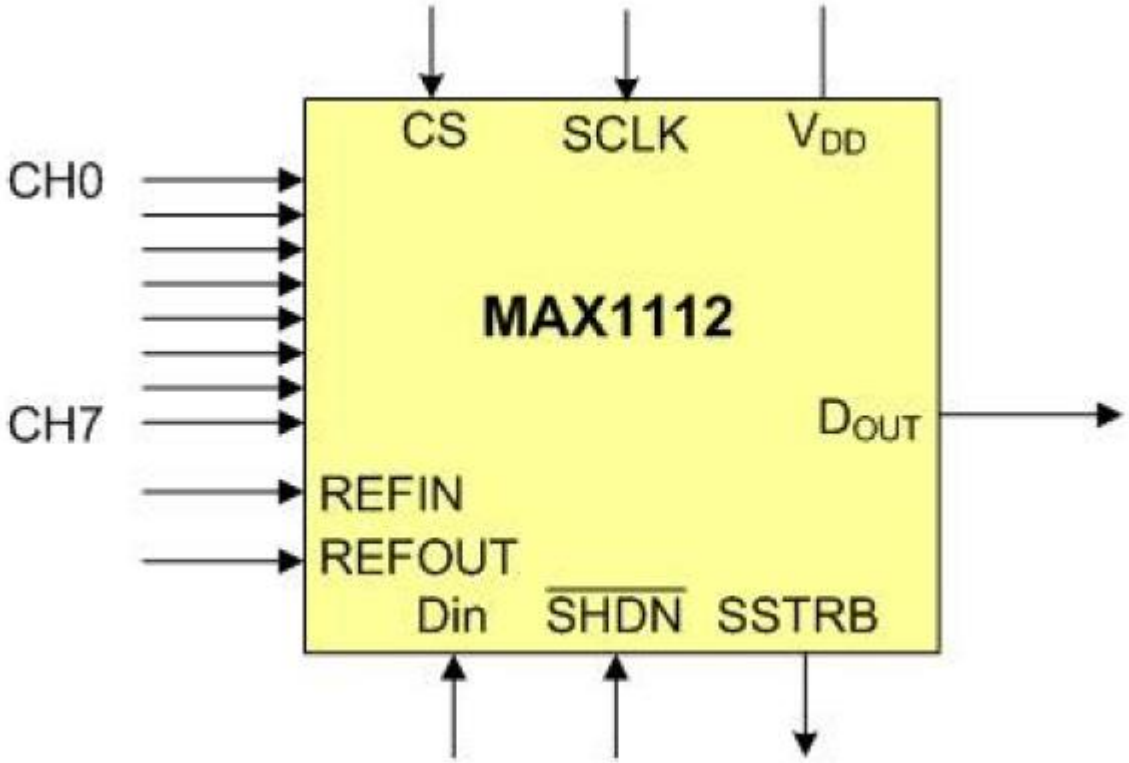
Parallel versus serial ADC

The ADC0848 is an example of a parallel ADC with 8 pins for the data output



Parallel versus serial ADC

The MAX1112 is an example of a serial ADC witha single pin for Dout



Tecnológico de Monterrey

Major characteristics of ADCs – Analog Input Channels

Many data acquisition applications need more than one analog input for ADC.

For this reason, we see ADC chips with 2, 4, 8, or even 16 channels on a single chip.

Multiplexing of analog inputs is widely used as shown in the ADC848 and MAX1112.

In these chips, we have 8 channels of analog inputs, allowing us to monitor multiple quantities such as temperature, pressure, flow, and so on.

Start conversion and end-of-conversion signals

For the conversion to be controlled by the CPU, there are needs for start conversion (SC) and end-of-conversion (EOC) signals.

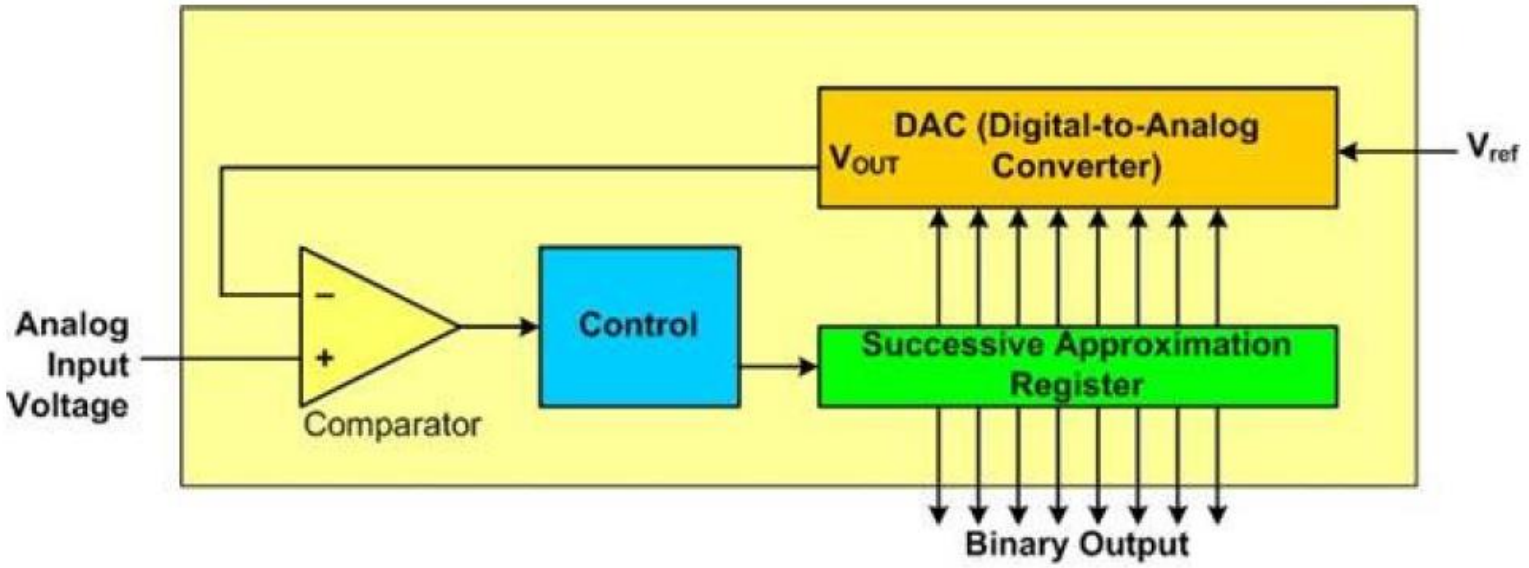
When SC is activated, the ADC starts converting the analog input value of Vin to a digital number.

The amount of time it takes to convert varies depending on the conversion method.

When the data conversion is complete, the end-of-conversion signal notifies the CPU that the converted data is ready to be picked up.

Successive Approximation ADC

Successive Approximation is a widely used method of converting an analog input to digital output. It has three main components: (a) successive approximation register (SAR), (b) comparator, and (c) control unit. See the figure below.



Successive Approximation ADC

The successive approximation register is loaded with only the most significant bit set at the start.

An internal digital-to-analog converter converts the value of SAR to an analog voltage which is used to compare to the input voltage.

If the input voltage is higher, the bit is kept. If the voltage is lower, the bit is cleared.

The next bit is tried and the DAC and compare are exercised. This process is repeated for all bits of the SAR.

Successive Approximation ADC

Assuming a step size of 10 mV, the 8-bit successive approximation ADC will go through the following steps to convert an input of 1 Volt:

(1) It starts with binary number 10000000. Since 128 × 10 mV = 1.28 V is greater than the 1 V input, bit 7 is cleared (dropped).

(2) 01000000 gives us 64 × 10 mV = 640 mV and bit 6 is kept since it is smaller than the 1 V input.

(3) 01100000 gives us 96 × 10 mV = 960 mV and bit 5 is kept since it is smaller than the 1 V input,

Successive Approximation ADC

(4) 01110000 gives us 112 × 10 mV = 1120 mV and bit 4 is dropped since it is greater than the 1 V input.

(5) 01101000 gives us 108 × 10 mV = 1080 mV and bit 3 is dropped since it is greater than the 1 V input.

(6) 01100100 gives us 100 × 10 mV = 1000 mV = 1 V and bit 2 is kept since it is equal to input. Even though the answer is found it does not stop.

(7) 011000110 gives us 102 × 10 mV = 1020 mV and bit 1 is dropped since it is greater than the 1 V input.

Successive Approximation ADC

(8) 01100101 gives us 101 × 10 mV = 1010 mV and bit 0 is dropped since it is greater than the 1 V input.

Notice that the Successive Approximation method goes through all the steps

Even if the answer is found in one of the earlier steps!!!

The advantage of the Successive Approximation method is that the conversion time is fixed since it has to go through all the steps.

ADC Programming with the Freescale KL25Z

Because the ADC is widely used in data acquisition, in recent years an increasing number of microcontrollers have on-chip ADC modules.

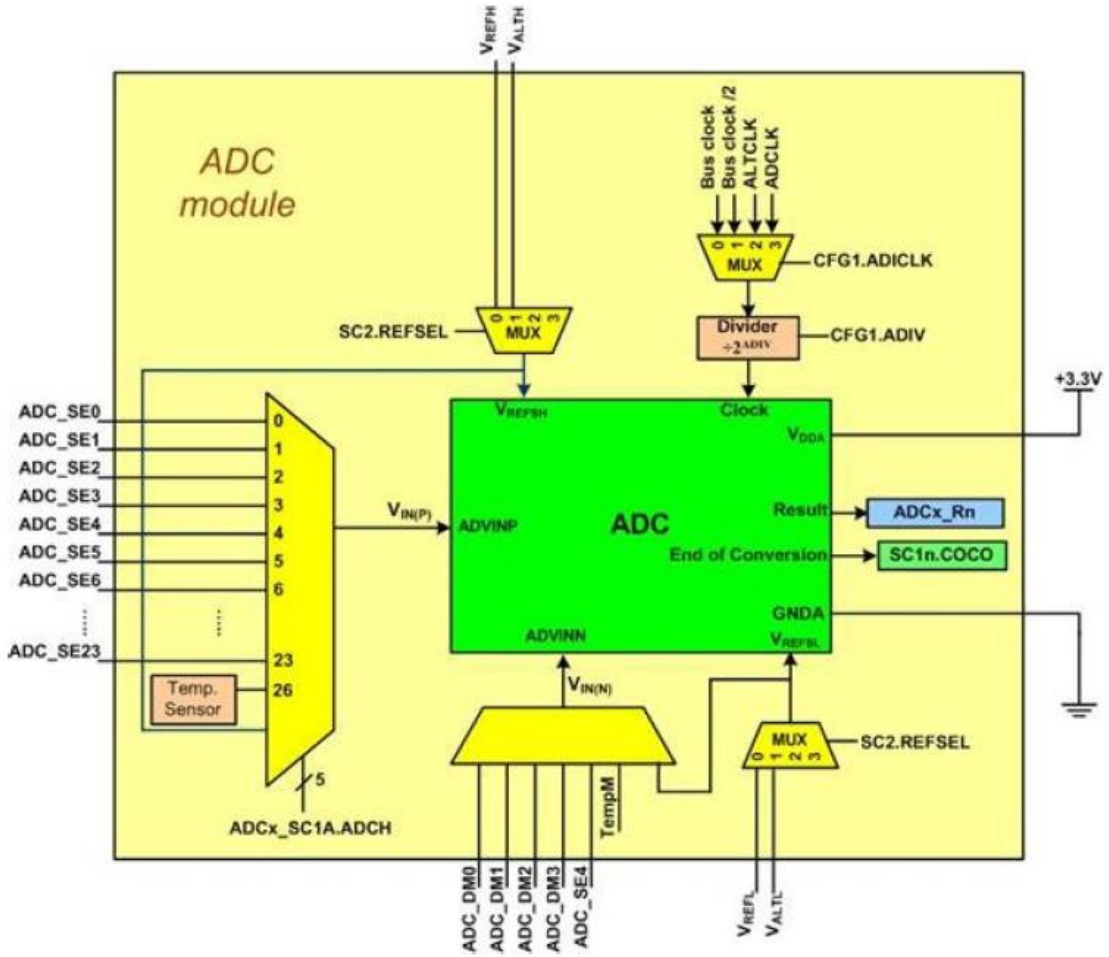
In this section, we discuss the ADC feature of the Freescale KL25Z and show how it is programmed.

The Freescale KL25Z ARM chip has a single ADC module which can support up to 31 ADC channels.

These ADC channels have 16-bit resolution. To program them, we need to understand some of the major registers

Tecnológico de Monterrey

ADC Programming with the Freescale KL25Z



ADC Programming with the Freescale KL25Z

In this section, we examine some of these registers and show how to program the ADC. Below is some major registers of KL25Z ADC from KL25Z reference manual.

Absolute Address	Register
4003 B000	ADC Status and Control Registers 1 (ADC0_SC1A)
4003 B004	ADC Status and Control Registers 1 (ADC0_SC1B)
4003 B008	ADC Configuration Register 1 (ADC0_CFG1)
4003 B00C	ADC Configuration Register 2 (ADC0_CFG2)
4003 B010	ADC Data Result Register (ADC0_RA)
4003 B014	ADC Data Result Register (ADC0_RB)

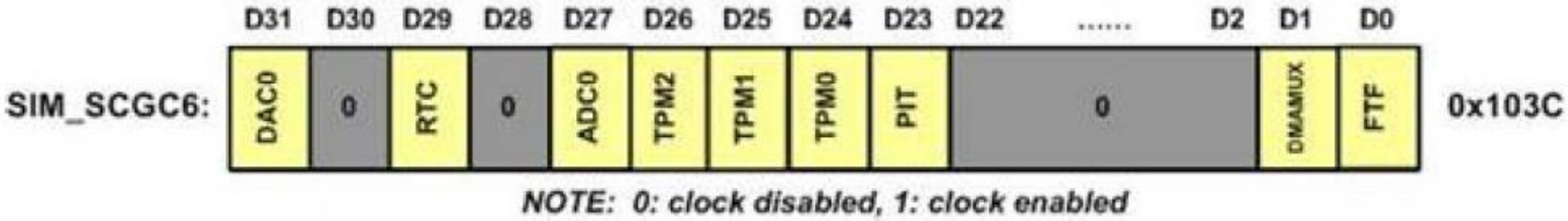
Tecnológico de Monterrey

ADC Programming with the Freescale KL25Z

Enabling Clock to ADC

First thing we need to do is to enable the clock to the ADC0 module. Bit D27 of SIM_SCGC6 register is used to enable the clock to ADC0 module.

The SIM_SCGC6 is part of the System Integration Module and located at physical address 0x4000 803C



ADC Programming with the Freescale KL25Z

Bit	Field	Descriptions
7	ADACT	Conversion active: Indicates that the ADC is converting data (0: Conversion not in progress, 1: Conversion in progress)
6	ADTRG	ADC conversion trigger select (0: software trigger, 1: hardware trigger)
5	ACFE	Compare Function Enable (0: compare function disabled, 1: enabled)
4	ACFGT	Compare Function Greater Than Enable
3	ACREN	Compare range Enable
2	DMAEN	DMA Enable

ADC Programming with the Freescale KL25Z

Bit	Field	Descriptions
1-0	REFSEL	Voltage Reference Select

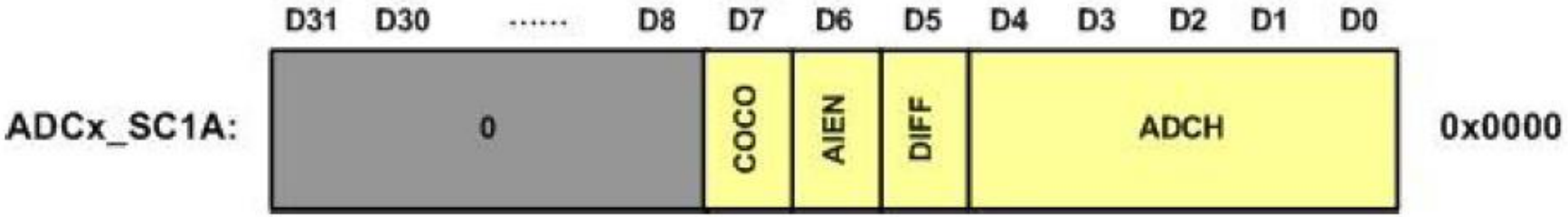
REFSEL	Voltage Reference
00	The V_{REFH} and V_{REFL} pins are used as VREF(+) and VREF(-), respectively.
01	V_{ALTH} and V_{ALTL} pair is used as references.
Others	Reserved

ADC Programming with the Freescale KL25Z

Choosing Vin input channel

The channel selection is done through the ADC0_SC1A (ADC Status and Control 1A) register. (There are more ADC0_SC1n registers but only ADC0_SC1A can be used for software trigger.

The lowest 5 bits of ADC0_SC1A register are used to select one of the 31 single-ended channels to be converted.



ADC Programming with the Freescale KL25Z

Bit	Field	Descriptions
7	COCO	Conversion Complete Flag: (0: Conversion is not completed, 1: Conversion is completed) The COCO is cleared when the ADCx_SC1n register is written or the ADCx_Rn register is read.
6	AIEN	Interrupt Enable: The ADC interrupt is enabled by setting the bit to HIGH. If the interrupt enable is set, an interrupt is triggered when the COCO flag is set.

ADC Programming with the Freescale KL25Z

5	DIFF	Differential mode (0: Single-ended mode, 1: Differential mode)
4-0	ADCH	ADC input channel: The field selects the input channel as shown in Figure 7-7. When DIFF = 0 (single-ended mode), values 0 to 23 choose between the 24 input channels (ADC_SE0 to ADC_SE23). When DIFF = 1 (Differential mode), values 0 to 3 select between the 4 differential channels. See the reference manual for more information. When ADCH = 11111, the module is disabled.

ADC Programming with the Freescale KL25Z

Not all the channels are connected to the input pins.

The number of available channels in the Freescale KL25Z varies among the family members.

In the case of KL25Z128VLK4 ARM chip used in FRDM board →

There are 14 channels connected to the input pins and additional 4 channels are connected internally.

Tecnológico de Monterrey

ADC Programming with the Freescale KL25Z

Pin Name	Description	Pin
ADC_SE0	ADC input 0	PTE20
ADC_SE3	ADC input 3	PTE22
ADC_SE4	ADC input 4	PTE21, PTE29
ADC_SE5	ADC input 5	PTD1
ADC_SE6	ADC input 6	PTD5
ADC_SE7	ADC input 7	PTD6, PTE23
ADC_SE8	ADC input 8	PTB0
ADC_SE9	ADC input 9	PTB1
ADC_SE11	ADC input 11	PTC2
ADC_SE12	ADC input 12	PTB2
ADC_SE13	ADC input 13	PTB3
ADC_SE14	ADC input 14	PTC0
ADC_SE15	ADC input 15	PTC1
ADC_SE23	ADC input 23, DAC0 output	PTE30
ADC_SE26	Temperature sensor	
ADC_SE27	Bandgap reference	
ADC_SE29	V_{REFH}	
ADC_SE30	V_{REFL}	
ADC_SE31	Module disabled	

Polling or interrupt

The end-of-conversion is indicated by a flag bit in the ADC0_SC1A (ADC Status Control 1A) register.

Upon the completion of conversion, the D7 bit Conversion Complete (COCO) flag goes high. By polling this flag, we know if the conversion is complete and we can read the value in ADC0_R0 data result register.

We can also use an interrupt to inform us that the conversion is complete but that will require us to set the AIEN (Interrupt Enable) bit (bit 6) high in ADC0_SC1A register.

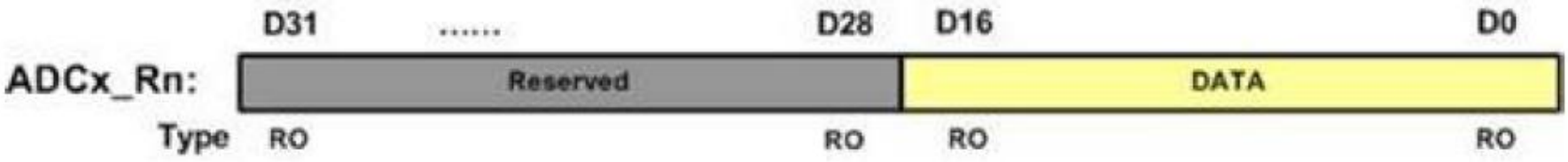
ADC Data result

Upon the completion of conversion, the binary result is placed in the ADC0_RA register.

There are many ADC0_Rn registers corresponding to the ADC0_SC1n registers.

Because we can only use ADC0_SC1A for software trigger, the data will be in ADC0_RA register.

This is a 32-bit register but only the lower 16 bits are used.



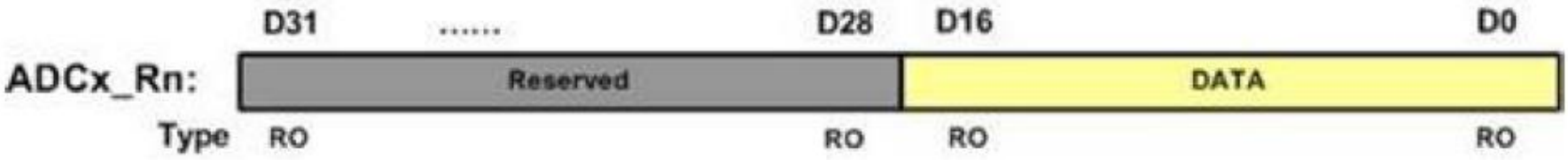
Tecnológico de Monterrey

ADC Data result

For the ADC, we have the options of 8-, 10-, 12-, and 16- bit for single-ended unsigned result.

In any case, always the result is right-justified and the rest of the bits up to bit D15 are unused.

If the result is in 2's complement for differential, then it is signed-extended to bit D15.



Differential versus Single-Ended

In some applications, our interest is in the differences between two analog signal voltages (the differential voltages).

Rather than converting two channels and calculate the differences between them, the KL25Z has the option of converting the differential voltages of two analog channels.

The bit D5 (DIFF) of ADC0_SC1A register allows us to enable the differential option.

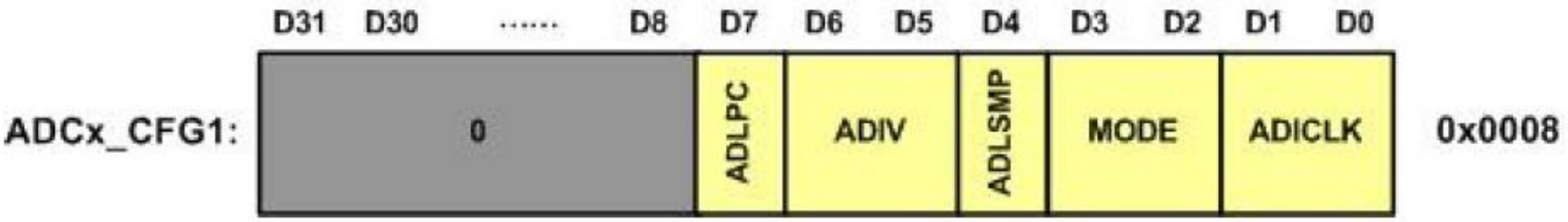
Upon Reset, the default is the single-ended input and we will leave it at that for the discussion here.

Selection Bit Resolution

We use ADCx_CFG1 (ADC configuration 1) register

To select 8, 10, 12, or 16-bit ADC resolution.

This register is also used to select the speed of the clock source to the ADC.



Selection Bit Resolution

Bit	Field	Descriptions
7	ADLPC	Low-Power Configuration
6-5	ADIV	Clock Divide Select: The clock is divided by 2^{ADIV} as shown in Figure 7-7.
4	ADLSMP	Sample time configuration (0: Short sample time, 1: Long sample time)

Notice in ADCx_CFG1, the MODE bits (D3:D2) selects the resolution.

Also notice, if we use the Low Power option with D7 bit, then the conversion speed is limited.

Selection Bit Resolution

Bit	Field	Descriptions
3-2	MODE	Conversion mode selection

MODE	In single-ended mode (DIFF = 0)	In differential mode (DIFF = 1)
00	8-bit conversion	9-bit conversion with 2's complement output
01	12-bit conversion	13-bit conversion with 2's complement output
10	10-bit conversion	11-bit conversion with 2's complement output
11	16-bit conversion	16-bit conversion with 2's complement output

Selection Bit Resolution

Bit	Field	Descriptions
1-0	ADICLK	Input Clock Select

ADICLK	Clock source
00	Bus clock
01	(Bus clock)/2
10	Alternate clock (ALTCLK)
11	Asynchronous clock (ADACK)

ADC Conversion Time

The conversion time for the ADC has 3 parts:

1) In the first phase, a sample amplifier of unity gain samples the analog input for a total of n clock cycles. This buffering of the analog input charges the sample capacitor up to the input potential.

2) In the second phase, the sample buffer is disconnected and connected to the storage node for a certain number of clock cycles. The number of clock cycles can be 4, 6, 10, 16, or 24.

We program this number via the ADLSMP bit in ADCx_CFG1 register and the ADLSTS bits in ADCx_CFG2 register.

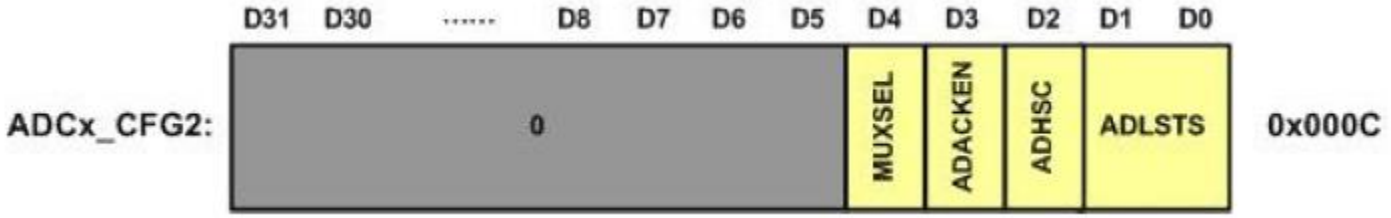
Sampling time ensures that the voltage of the sample capacitor is brought closer to the input voltage. This is important when the input voltage differs significantly from sample to sample. But it prolongs the conversion time of each sample.

3) In the third phase, the analog input is converted to binary numbers using the successive approximation method. In this phase, the number of clocks used depends on how many bits are in the binary output. For each bit we need one clock. That means we need 8 clocks for the 8-bit output, 10 clocks for the 10-bit output, and so on. We choose the *n*-bit resolution option using the MODE bits of ADCx_CFG1 register

The ADCx_CFG1 gives us many options for the clock fed to ADC module.

We can choose the Bus Clock or a fraction of it. Using the ADICLK (AD input Clock) and ADIV (AD divide) bits of ADCx_CFG1 registers we can control the speed of clock source fed to the ADC.

These bits along with the bits in ADCx_CFG2, we can control the conversion time.



Vref in FRDM board

In the Freescale ARM KL25Z chip, the pin for Vref (+) is called VREFH (Vref High) and Vref (-) pin is called VREFL (Vref Low).

In the FRDM board, the VREFH pin is connected to 3.3V, the same supply voltage as the digital part of the chip.

The circuit may be altered to use the AREF pin for an external reference voltage.

Even if we connect the VREFH to an external reference other than the VDD of the chip, it cannot go beyond the VDD voltage. With VREFH=3.3V, we have the step size of 3.3V / 65,536= 0.05 mV since the maximum ADC resolution for KL25Z is 16 bits.

Configuring ADC and reading ADC channel

In using ADC, we must also configure the GPIO pins to allow the connection of an analog signal through the input pin. In this regard, it is the same as all other peripherals. We need to take the following steps to configure the ADC:

1. Enable the clock to I/O pin used by the ADC channel. Table in Slide 35 shows the I/O pins used by various ADC channels.
2. Set the PORTX_PCRn MUX bit for ADC input pin to 0 to use the pin for analog input channel. This is actually the power-on default.
3. Enable the clock to ADC0 modules using SIM_SCGC6 register.

5. Choose clock rate and resolution using ADC0_CFG1 register.

6. Select the ADC input channel using the ADC0_SC1A register. Make sure to use the table to choose the right pin and channel. Also makes sure interrupt is not enabled and single-ended option is used when you select the channel with this register.

7. Keep monitoring the end-of-conversion COCO flag in ADC0_SC1A register.

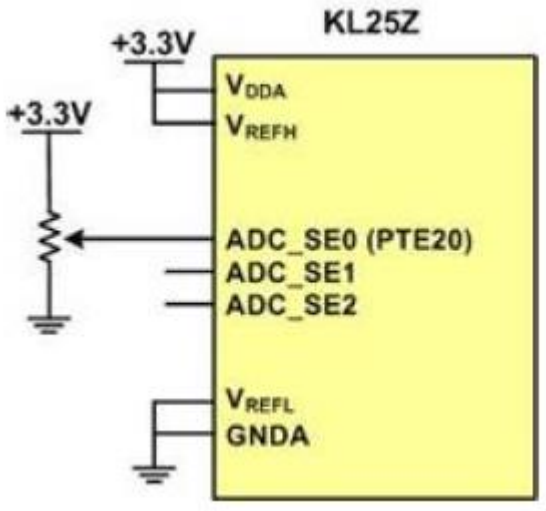
8. When the COCO flag goes HIGH, read the ADC result from the ADC0_RA and save it.

9. Repeat steps 6 through 8 for the next conversion.

Tecnológico de Monterrey

```
/* A to D conversion of channel 0
* This program converts the analog input from channel 0
(PTE20)
* using software trigger continuously.
* Bits 10-8 are used to control the tri-color LEDs. LED code
* is  copied from p2_7. Connect a potentiometer between 3.3V
* and  ground.
* The wiper of the potentiometer is connected to PTE20.
• When the potentiometer is turned, the LEDs should change
  color. */
```

```
#include "MKL25Z4.h"
void ADC0_init(void);
void LED_set(int s);
void LED_init(void);
```



```
int main (void)
{

short int result;
LED_init(); /* Configure LEDs */
ADC0_init(); /* Configure ADC0 */

while (1) {

ADC0->SC1[0] = 0; /* start conversion on channel 0 */

while(!(ADC0->SC1[0] & 0x80)) { } /* wait for
conversion complete */
result = ADC0->R[0]; /* read conversion result and
clear COCO flag */
LED_set(result >> 7); /* display result on LED */
}
}
```

```
void ADC0_init(void)
{

SIM->SCGC5 |= 0x2000; /* clock to PORTE */

PORTE->PCR[20] = 0; /* PTE20 analog input */

SIM->SCGC6 |= 0x8000000; /* clock to ADC0 */

ADC0->SC2 &= ~0x40; /* software trigger */

/* clock div by 4, long sample time, single
ended 12 bit, bus clock */

ADC0->CFG1 = 0x40 | 0x10 | 0x04 | 0x00;
}
```

```
void LED_init(void) {

SIM->SCGC5 |= 0x400; /* enable clock to Port B */

SIM->SCGC5 |= 0x1000; /* enable clock to Port D */

PORTB->PCR[18] = 0x100; /* make PTB18 pin as GPIO */

PTB->PDDR |= 0x40000; /* make PTB18 as output pin */

PORTB->PCR[19] = 0x100; /* make PTB19 pin as GPIO */

PTB->PDDR |= 0x80000; /* make PTB19 as output pin */

PORTD->PCR[1] = 0x100; /* make PTD1 pin as GPIO */

PTD->PDDR |= 0x02; /* make PTD1 as output pin */
}
```

```
void LED_set(int s) {

if (s & 1) /* use bit 0 of s to control red LED */
PTB->PCOR = 0x40000; /* turn on red LED */
else
PTB->PSOR = 0x40000; /* turn off red LED */

if (s & 2) /* use bit 1 of s to control green LED */
PTB->PCOR = 0x80000; /* turn on green LED */
else
PTB->PSOR = 0x80000; /* turn off green LED */

if (s & 4) /* use bit 2 of s to control blue LED */
PTD->PCOR = 0x02; /* turn on blue LED */
else
PTD->PSOR = 0x02; /* turn off blue LED */
}
```

Temperature sensor

There are several internal analog channels as seen in Table in slide 35.

The next program shows how to convert the internal temperature sensor output.

In the ADC initialization, there is no need to initialize the input pin because the channel is connected internally.

```
#include “MKL25Z4.h”
void ADC0_init(void);
void LED_set(int s);
void LED_init(void);
int main (void)
{
short int result;
LED_init(); /* Configure LEDs */
ADC0_init(); /* Configure ADC0 */
while (1) {
ADC0->SC1[0] = 26; /* start conversion on channel 26
temperature */
while(!(ADC0->SC1[0] & 0x80)) { } /* wait for COCO */
result = ADC0->R[0]; /* read conversion result and clear
COCO flag */
LED_set(result); /* display result on LED */
}
```

```
void ADC0_init(void)
{
SIM->SCGC6 |= 0x8000000; /* clock to ADC0 */
ADC0->SC2 &= ~0x40; /* software trigger */
/* clock div by 4, long sample time, single ended 12 bit, bus
clock */
ADC0->CFG1 = 0x40 | 0x10 | 0x04 | 0x00;
}
void LED_init(void) {
SIM->SCGC5 |= 0x400; /* enable clock to Port B */
SIM->SCGC5 |= 0x1000; /* enable clock to Port D */
PORTB->PCR[18] = 0x100; /* make PTB18 pin as GPIO */
PTB->PDDR |= 0x40000; /* make PTB18 as output pin */
PORTB->PCR[19] = 0x100; /* make PTB19 pin as GPIO */
PTB->PDDR |= 0x80000; /* make PTB19 as output pin */
PORTD->PCR[1] = 0x100; /* make PTD1 pin as GPIO */
PTD->PDDR |= 0x02; /* make PTD1 as output pin */
}
```

```
void LED_set(int s) {
if (s & 1) /* use bit 0 of s to control red LED */
PTB->PCOR = 0x40000; /* turn on red LED */
else
PTB->PSOR = 0x40000; /* turn off red LED */
if (s & 2) /* use bit 1 of s to control green LED */
PTB->PCOR = 0x80000; /* turn on green LED */
else
PTB->PSOR = 0x80000; /* turn off green LED */
if (s & 4) /* use bit 2 of s to control blue LED */
PTD->PCOR = 0x02; /* turn on blue LED */
else
PTD->PSOR = 0x02; /* turn off blue LED */
}
```

Sensor Interfacing and Signal Conditioning

This section will show how to interface sensors to the microcontroller.

We examine some popular temperature sensors and then discuss the issue of signal conditioning.

Although we concentrate on temperature sensors, the principles discussed in this section are the same for other types of sensors such

as light and
pressure sensors.

Temperature sensors

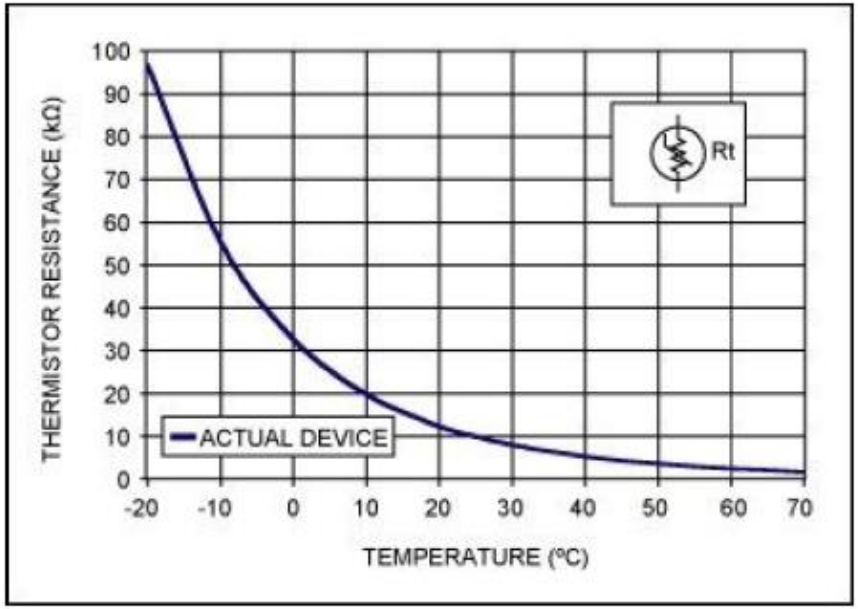
Transducers convert physical data such as temperature, light intensity, flow, and speed to electrical signals.

Depending on the transducer, the output produced is in the form of voltage, current, resistance, or capacitance.

For example, temperature is converted to electrical signals using a transducer called a *thermistor*.

A thermistor responds to temperature change by changing resistance, but its response is not linear

Temperature (‘C)	Tf (K ohms)
0	29.490
25	10.000
50	3.893
75	1.700
100	0.817



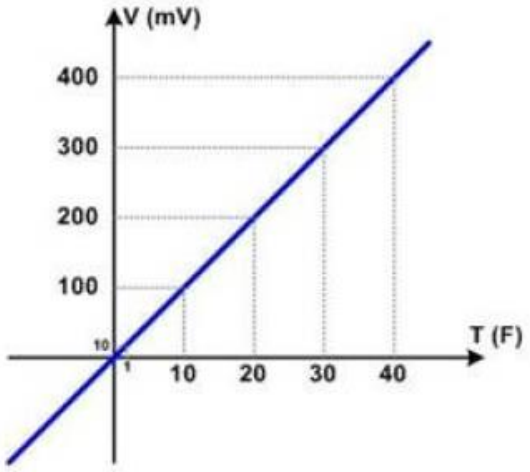
The resistance of a thermistor is typically modeled by Steinhart-Hart equation and requires a logarithmic amplifier to produce a linear output.

Simple and widely used linear temperature sensors include the LM34 and LM35 series from National Semiconductor

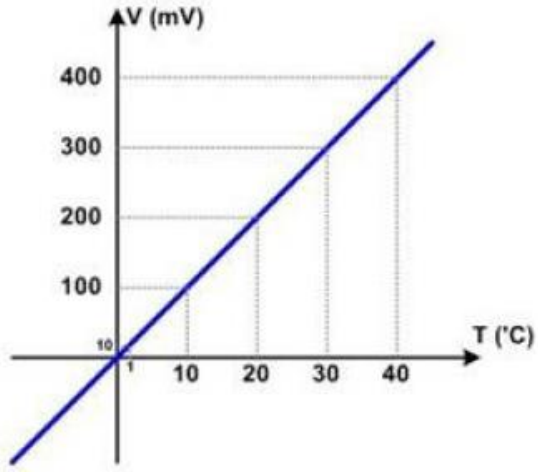
Tecnológico de Monterrey

LM34 and LM35 temperature sensors

The sensors of the LM34 series are precision integrated-circuit temperature sensors whose output voltage is linearly proportional to the Fahrenheit temperature. The LM34 requires no external calibration because it is internally calibrated. It outputs 10 mV for each degree of Fahrenheit (LM35 °C)



(a) LM34



(b) LM35

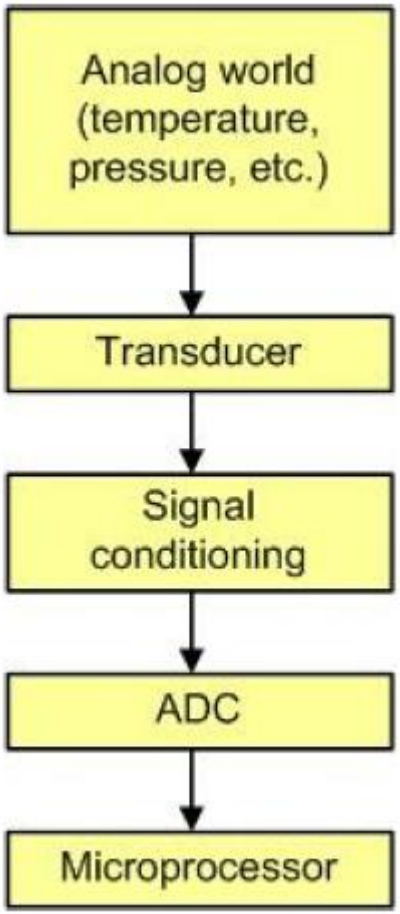
Signal conditioning

The common transducers produce an output in the form of voltage, current, charge, capacitance, or resistance.

In order to perform A-to-D conversion on these signals, they need to be converted to voltage unless the transducer output is already voltage.

In addition to the conversion to voltage, the signal may also need gain and offset adjustment to achieve optimal dynamic range.

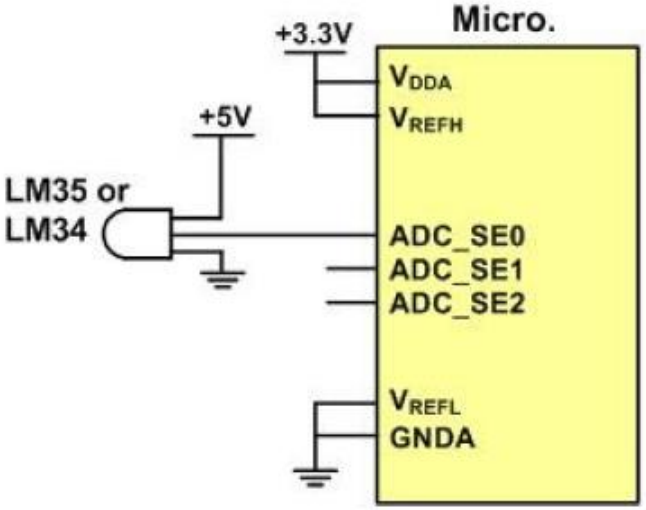
A low-pass analog filter is often incorporated to eliminate the high frequency to avoid aliasing.

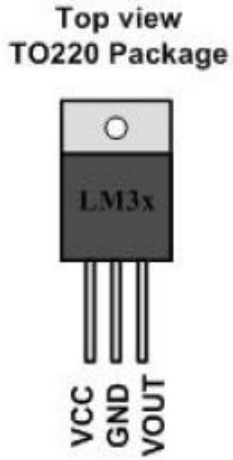

Analog world (temperature, pressure, etc.)
Transducer
Signal conditioning
ADC
Microprocessor


Homework

Interfacing the LM34 to the ARM Microcontroller

The A/D of Freescale ARM Microcontroller has 16-bit resolution with a maximum of 65,536 steps, and the LM34 produces 10 mV for every degree of temperature change. The maximum operating temperature of the LM34 is 300 degrees F, so the highest output will be 3000 mV (3.00 V), which is below 3.3V of Vref.





Tecnológico de Monterrey

```
/* A to D conversion of channel 0
* This program converts the analog input from channel 0
*(PTE20) using software trigger continuously. PTE20 is
*connected to an LM34 Fahrenheit temperature sensor.

* The conversion result is displayed as temperature
*through UART0 virtual serial port.
* 16-bit precision is used for conversion. At higher *
*precision, the noise is more significant so 32 samples
* averaging is used.
* The LM34 output voltage is 10mV/degreeF. The ADC of
FRDM-*KL25Z uses 3.3V as Vref so:
* temperature = result * 330.0 / 65536
* Open a terminal emulator at 115200 Baud rate at the host
* PC and * observe the output.
*/
```

```
#include “MKL25Z4.h”
#include <stdio.h>

void ADC0_init(void);
void delayMs(int n);
void UART0_init(void);
void UART0Tx(char c);
void UART0_puts(char* s);

int main (void) {

int result;
float temperature;
char buffer[16];

ADC0_init(); /* Configure ADC0 */
UART0_init(); /* initialize UART0 for output */
```

```
while (1) {
ADC0->SC1[0] = 0; /* start conversion on channel 0
*/
while(!(ADC0->SC1[0] & 0x80)) { } /* wait for COCO
*/
result = ADC0->R[0];
/* read conversion result and clear COCO flag */

temperature = result * 330.0 / 65536;
/* convert voltage to temperature */

sprintf(buffer, "\r\nTemp = %6.2fF", temperature);
/* convert to string */

UART0_puts(buffer);
/* send the string through UART0 for display */
delayMs(1000);
} }
```

```
void ADC0_init(void)
{
SIM->SCGC5 |= 0x2000; /* clock to PORTE */

PORTE->PCR[20] = 0; /* PTE20 analog input */

SIM->SCGC6 |= 0x8000000; /* clock to ADC0 */

ADC0->SC2 &= ~0x40; /* software trigger */

ADC0->SC3 |= 0x07; /* 32 samples average */

/* clock div by 4, long sample time, single
ended 16 bit, bus clock */

ADC0->CFG1 = 0x40 | 0x10 | 0x0C | 0x00;
}
```

```
/* initialize UART0 to transmit at 115200 Baud */
void UART0_init(void) {
SIM->SCGC4 = 0x0400; /* enable clock for UART0 */
SIM->SOPT2 = 0x04000000;
/* use FLL output for UART Baud rate generator */
UART0->C2 = 0; /* turn off UART0 while changing
configurations */
UART0->BDH = 0x00;
UART0->BDL = 0x17; /* 115200 Baud */
UART0->C4 = 0x0F; /* Over Sampling Ratio 16 */
UART0->C1 = 0x00; /* 8-bit data */
UART0->C2 = 0x08; /* enable transmit */
SIM->SCGC5 = 0x0200; /* enable clock for PORTA */
PORTA->PCR[2] = 0x0200; /* make PTA2 UART0_Tx pin */
}
```

```
void UART0Tx(char c) {
while(!(UART0->S1 & 0x80)) {
} /* wait for transmit buffer empty */
UART0->D = c; /* send a char */
}
void UART0_puts(char* s) {
while (*s != 0) /* if not end of string */
UART0Tx(*s++); /* send the character through UART0 */
}
/* Delay n milliseconds
* The CPU core clock is set to MCGFLLCLK at 41.94 MHz
in SystemInit().
*/
void delayMs(int n) {
int i;
int j;
for(i = 0 ; i < n; i++)
for (j = 0; j < 7000; j++) {}
}
```

Tecnológico de Monterrey

Microcontrollers

Professor: Dr. Gilberto Ochoa Ruiz

Topic 7: Digital to Analog Converter

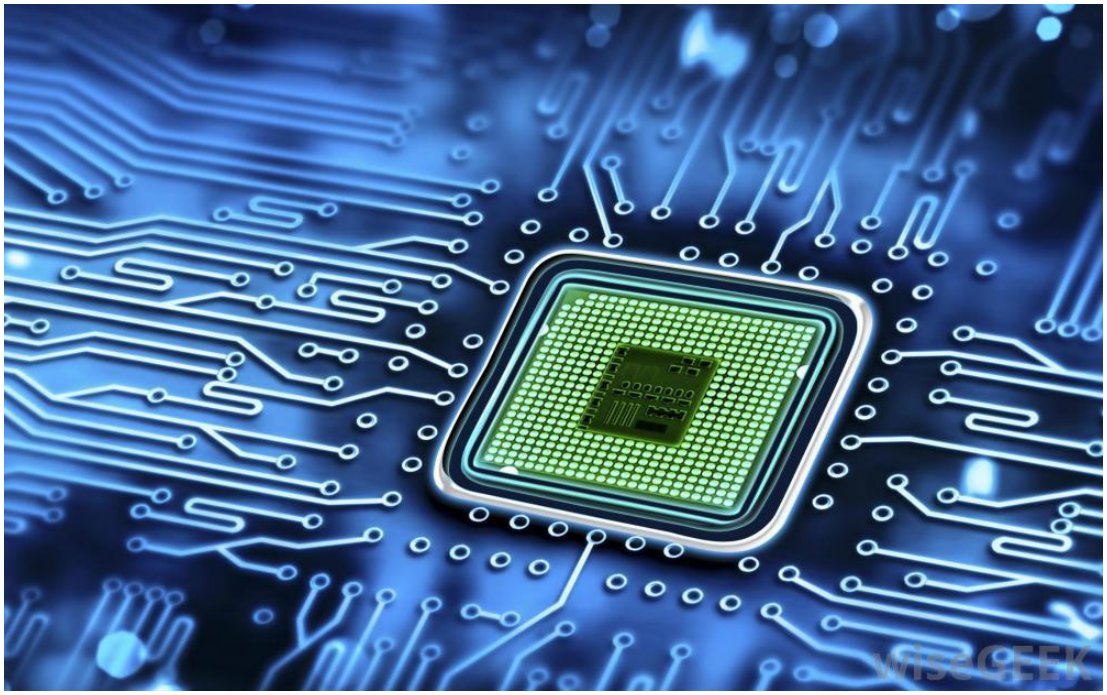
EIAD-204

Wednesday

May 27th 2020

6h30 – 9h30 PM

Guadalajara, Mexico



Basic Programming Techniques – DAC

Digital-to-analog (DAC) converter

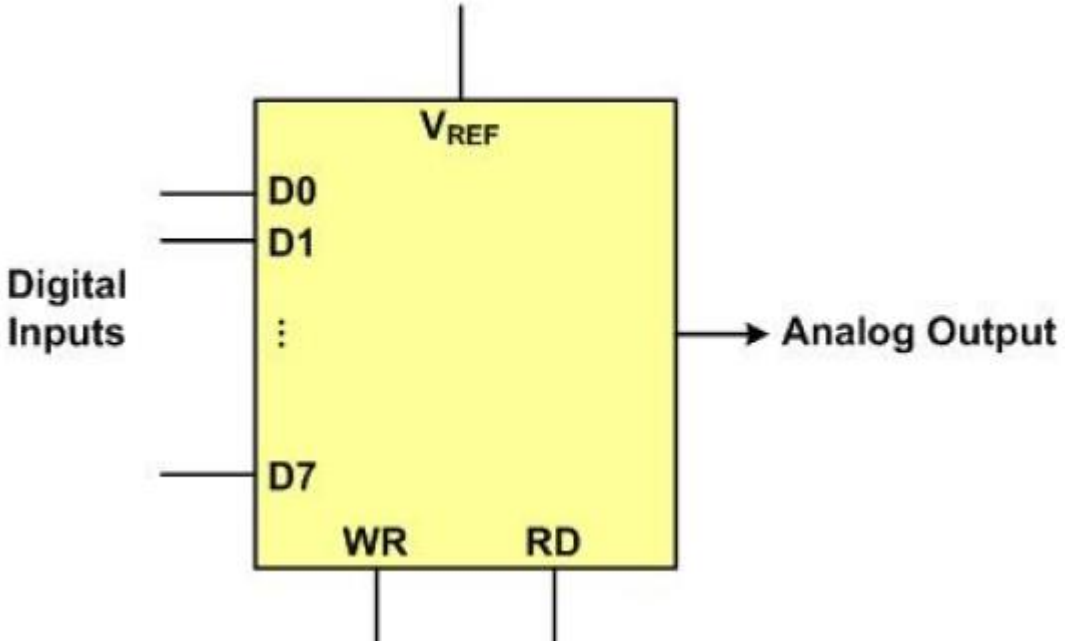
The digital-to-analog converter (DAC) is a device widely used to convert digital signals to analog signals. In this section we discuss the basics of a DAC.

Recall from your digital electronics book the two methods of making a DAC: binary weighted and R/2R ladder. The vast majority of integrated circuit DACs, including the DAC0808 discussed in this section use the R/2R method since it can achieve a much higher degree of precision.

The first criterion for selecting a DAC
is its resolution, which is a function of the number of bits of the digital input.

Basic Programming Techniques – DAC

The common ones are 8, 10, and 12 bits. The number of digital input bits decides the resolution of the DAC since the number of analog output levels is equal to 2n, where n is the number of digital input bits. Therefore, the 8-bit DAC such as the DAC0808 provides 256 discrete voltage (or current) levels of output.



Basic Programming Techniques – DAC

DAC0808

In the DAC0808, the digital inputs are converted to current (IOUT). By connecting a resistor to the IOUT pin, we convert the conversion result current to voltage.

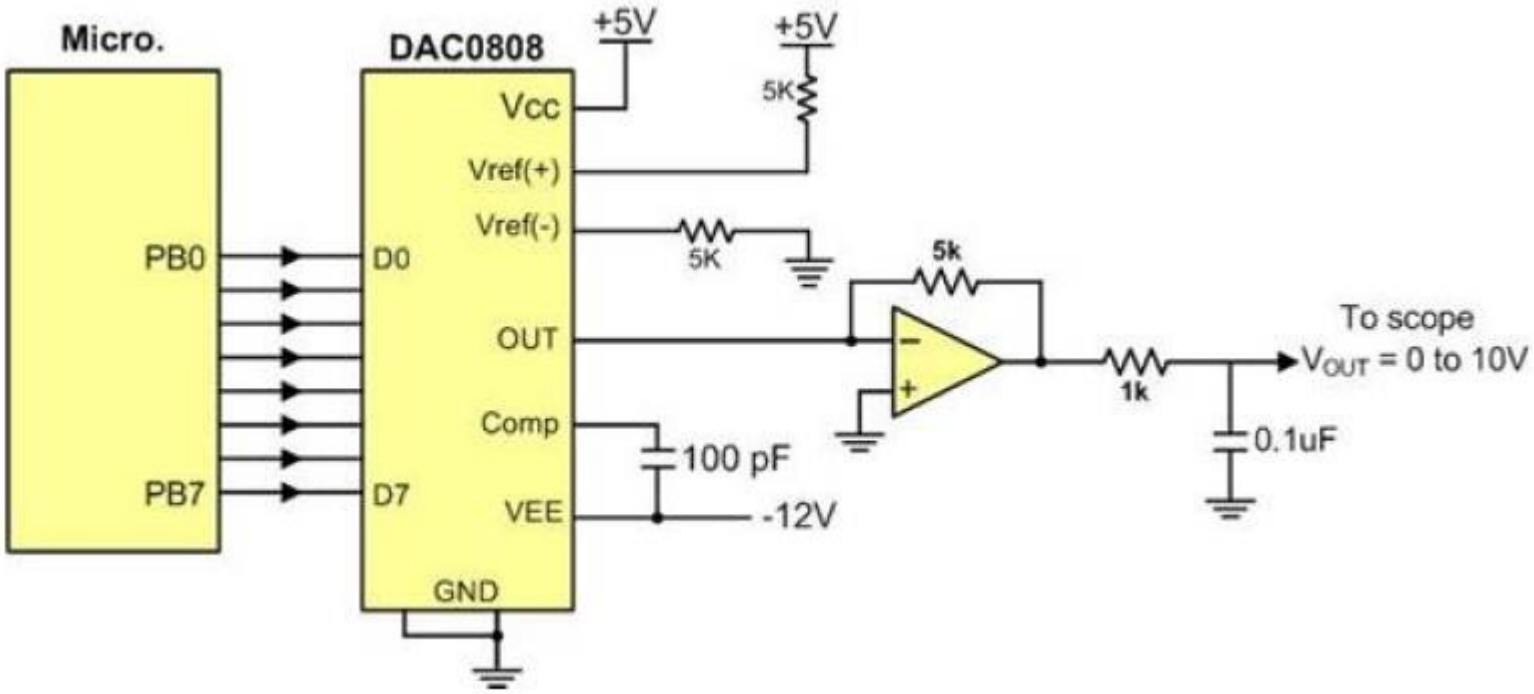
The total current provided by the IOUT is a function of the binary numbers at the D0–D7 inputs of the DAC0808 and the reference current (Iref), and is as follows:

$$IOUT = Iref \times (D7/2 + D6/4 + D5/8 + D4/16 + D3/32 + D2/64 + D1/128 + D0/256) = Iref \times Data / 256$$

where D0 is the LSB, D7 is the MSB for the inputs, and Iref is the reference input current that must be applied to pin 14.

Basic Programming Techniques – DAC

The Iref current is generally set to 2.0 mA. The figure below shows the generation of current reference (setting Iref = 2 mA) by using the standard 5-V power supply and 5K ohm resistors.



Basic Programming Techniques – DAC

Converting Iout to voltage in DAC0808

We connect the output pin IOUT to a resistor, convert this current to voltage, and monitor the output on the scope.

However, in real life this can cause inaccuracy since the input resistance of the load where it is connected will also affect the output voltage.

For this reason, the Iref current output is buffered by connecting it to an op amp such as the 741 with Rf = 5K ohms for the feedback resistor.

Assuming that R = 5K ohms, by changing the binary input, the output voltage changes as shown next

Basic Programming Techniques – DAC

Example

Assuming that R = 5K and Iref = 2 mA, calculate Vout for the following binary inputs:

(a) 10011001 binary (0x99) (b) 11001000 (0xC8)

Solution:

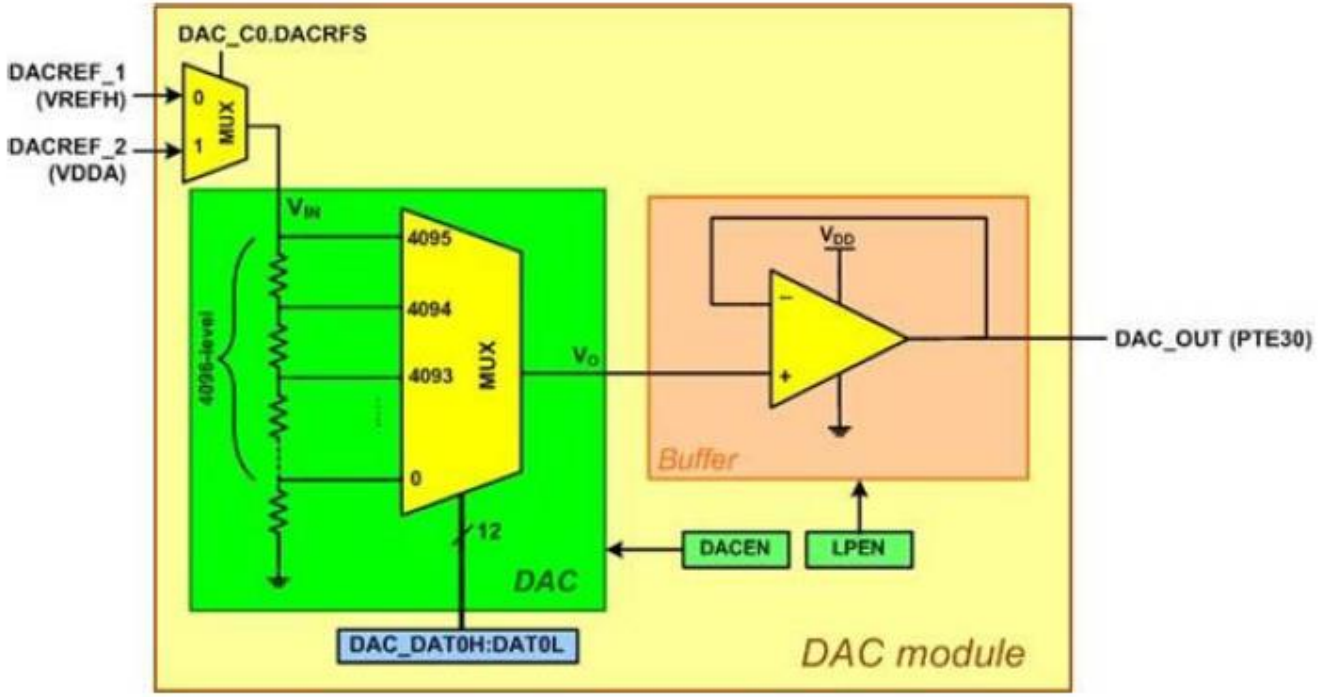
(a) Iout = 2 mA (153/255) = 1.195 mA and Vout = 1.195 mA × 5K = 5.975 V

(b) Iout = 2 mA (200/256) = 1.562 mA and Vout = 1.562 mA × 5K = 7.8125 V

Basic Programming Techniques – DAC

DAC Features of KL25Z

The Freescale KL25Z comes with an on-chip DAC. The on-chip DAC is 12-bit string converter. A string DAC uses a resistor ladder and an analog multiplexer.



Basic Programming Techniques – DAC

DAC Features of KL25Z

Below is a list of the major registers of KL25Z DAC from KL25Z ref. manual

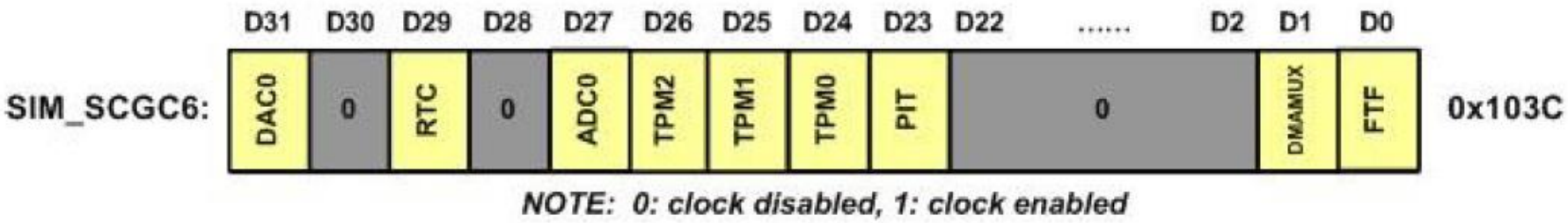
Absolute Address	Register
4003 F000	DAC Data Low Register (DAC0_DAT0L)
4003 F001	DAC Data High Register (DAC0_DAT0H)
4003 F002	DAC Data Low Register (DAC0_DAT1L)
4003 F003	DAC Data High Register (DAC0_DAT1H)
4003 F020	DAC Status Register (DAC0_SR)
4003 F021	DAC Control Register (DAC0_C0)

Basic Programming Techniques – DAC

Clocking the DAC

Next, we will examine some of the registers and use them to create staircase ramp and sine wave signals. First, we must provide clock to the DAC.

This is done with SIM_SCGC6 register. Notice bit D31 is used to provide the clock to on-chip DAC0.



Basic Programming Techniques – DAC

DAC Control 0 register

We also need to enable the DAC itself before we can use it. This is done with bit D7 of DAC Control 0 (DAC0_C0) register.

	D7	D6	D5	D4	D3	D2	D1	D0	
DAC_C0:	DACEN	DACRFS	DACTRGSEL	DACSWTRG	LPEN	0	DACBTIEN	DACBBIEN	0x0021

Other important bits in DAC_C0 registers are D5 (DACTRGSEL: DAC Trigger select) and D6 (DAC0_DACRFS: DAC Reference Select). The choices of reference voltages are DACREF_1 for VREFH and DACREF_2 for VDDA. In the FRDM_KL25Z board, VREFH is tied to VDDA so there is no difference is using either reference. We will use software trigger.

Basic Programming Techniques – DAC

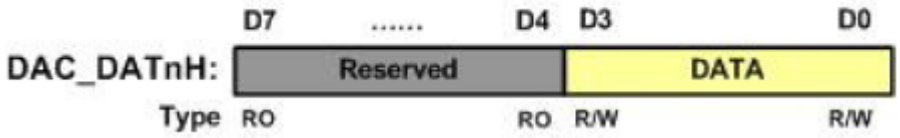
DAC Control 0 register

Bit	Field	Descriptions
7	DACEN	DAC Enable (0: DAC is disabled, 1: DAC is enabled)
6	DACRFS	DAC Reference Select (0: DACREF_1, 1: DACREF_2)
5	DACTRGSEL	DAC Trigger Select (0: hardware trigger, 1: software trigger)
4	DACSWTRG	DAC Software Trigger
3	LPEN	DAC Low Power Control (0: High-Power mode, 1: Low-Power mode)
1	DACBTIEN	DAC Buffer read pointer Top flag Interrupt Enable
0	DACBBIEN	DAC Buffer read pointer Bottom flag Interrupt Enable

Basic Programming Techniques – DAC

DAC Buffer register

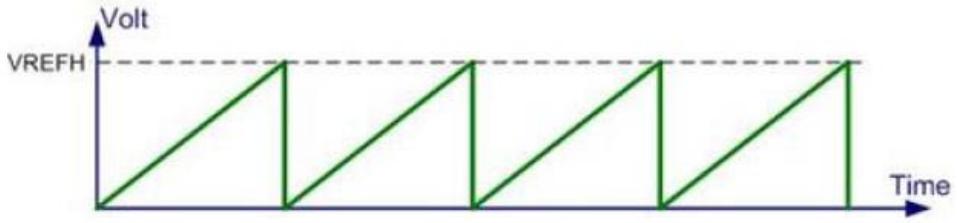
The data registers DAC0_DATnH:DAC0_DATnH hold the 12-bit digital (binary) value needed to be converted to the analog. The DAC0_DATnL register is used for the lower 8 bits and DAC0_DATnH for the upper 4 bits of the 12-bit binary value.



For KL25Z, there are two sets of buffer data registers, DAC0_DAT0 and DAC0_DAT1. The use of buffer register is more meaningful when hardware trigger is used. When buffer is disabled in DAC0_C1, only DAC0_DAT0 is used.

Basic Programming Techniques – DAC

```
/* Use DAC to generate sawtooth waveform
* The DAC is initialized with no buffer and use software
trigger,
* so every write to the DAC data registers will change
the analog output.
* The loop count i is incremented by 0x0010 every loop.
The 12-bit DAC
* has the range of 0-0x0FFF. Divide 0x1000 by 0x0010
yields 0x0100 or 256.
* The sawtooth has 256 steps and each step takes 1 ms.
The period of the
* waveform is 256 ms and the frequency is about 3.9 Hz.
*/
#include “MKL25Z4.h”
void DAC0_init(void);
void delayMs(int n);
```



Basic Programming Techniques – DAC

```
int main (void) {

int i;
DAC0_init(); /* Configure DAC0 */

while (1) {

for (i = 0; i < 0x1000; i += 0x0010) {

/* write value of i to DAC0 */
DAC0->DAT[0].DATL = i & 0xff; /* write low byte */

DAC0->DAT[0].DATH = (i >> 8) & 0x0f;/* write high byte */

delayMs(1); /* delay 1ms */
}
}
}
```

Basic Programming Techniques – DAC

```
void DAC0_init(void) {
SIM->SCGC6 |= 0x80000000; /* clock to DAC module */
DAC0->C1 = 0; /* disable the use of buffer */
DAC0->C0 = 0x80 | 0x20; /* enable DAC and use
software trigger */
}
/* Delay n milliseconds
* The CPU core clock is set to MCGFLLCLK at 41.94
MHz in SystemInit().
*/
void delayMs(int n) {
int i;
int j;
for(i = 0 ; i < n; i++)
for (j = 0; j < 7000; j++) {}
}
```

Basic Programming Techniques – DAC

Generating a sine wave

To generate a sine wave, we first need a table whose values represent the magnitude of the sine of angles between 0 and 360 degrees.

The values for the sine function vary from -1.0 to +1.0 for 0 to 360 degree angles.

Therefore, the table values are positive integer numbers representing the voltage magnitude for the Sine of the angle.

This method ensures that only positive integer numbers are output to the DAC by the processor.

Basic Programming Techniques – DAC

Angle Θ (degrees)	Sin Θ	V_{OUT} (Voltage Magnitude) (1.5V × sin Θ) + 1.5V	Values Sent to DAC (decimal) (Voltage Mag. ÷ 0.000806 V)
0	0.000	1.500	1862
30	0.500	2.250	2793
60	0.866	2.799	3474
90	1.000	3.000	3724
120	0.866	2.799	3474
150	0.500	2.250	2793
180	0.000	1.500	1862

Basic Programming Techniques – DAC

Angle Θ (degrees)	Sin Θ	V_{OUT} (Voltage Magnitude) (1.5V × sin Θ) + 1.5V	Values Sent to DAC (decimal) (Voltage Mag. ÷ 0.000806 V)
210	-0.500	0.750	931
240	-0.866	0.201	249
270	-1.000	0.000	0
300	-0.866	0.201	249
330	-0.500	0.750	931
360	0.000	1.500	1862

Basic Programming Techniques – DAC

To generate the table, we assumed the full-scale voltage of 3.3V for the DAC output.

Full-scale output of the DAC is achieved when all the data input bits of the DAC are high.

We will generate a sine wave with amplitude of 3.0V. Since

DAC only accepts positive integers, the values of sine wave shall be 1.5V±1.5V.

Therefore, to achieve the output amplitude of 3.0V, we use the following equation:

$$VOUT = 1.5V + (1.5V \times \sin \Theta)$$

Basic Programming Techniques – DAC

The DAC is 12 bit and the VREFH is 3.3V, so the step size if 3.3V / 4096 = 0.000806V. To find the values sent to the DAC for various angles, we simply divide the VOUT voltage by 0.000806V.

Example: to verify the values for 30 and 60°

(a) VOUT = 1.5 V + (1.5 V × sin Θ) = 1.5 V + 1.5 V × sin 30 = 1.5 V + 1.5 V × 0.5 = 2.25 V

DAC input values = 2.25 V ÷ 0.000806 V = 2793 (decimal)

(b) VOUT = 1.5 V + (1.5 V × sin Θ) = 1.5 V + 1.5 V × sin 60 = 1.5 V + 1.5 V ×0.866 = 2.799 V

DAC input values = 2.799 V ÷ 0.000806 V = 3474 (decimal)

Basic Programming Techniques – DAC

```
/* Use DAC to generate sine wave with look-up table
* This program uses a pre-calculated lookup table to
generate a  sine wave output through DAC. */

#include "MKL25Z4.h"

void DAC0_init(void);
void delayMs(int n);

int main (void) {

int i;
const static int sineWave[] =

{1862, 2793, 3474, 3724, 3474, 2793,
1862, 931, 249, 0, 249, 931};
```

Basic Programming Techniques – DAC

```
DAC0_init(); /* Configure DAC0 */

while (1) {

for (i = 0; i < 12; i++) {
/* write value to DAC0 */
DAC0->DAT[0].DATL = sineWave[i] & 0xff;
/* write low byte */

DAC0->DAT[0].DATH = (sineWave[i] >> 8) & 0x0f;
/* write high byte */

delayMs(1); /* delay 1ms */
}
}
}
```

Basic Programming Techniques – DAC

```
void DAC0_init(void) {
SIM->SCGC6 |= 0x80000000; /* clock to DAC module */
DAC0->C1 = 0; /* disable the use of buffer */
DAC0->C0 = 0x80 | 0x20; /* enable DAC and use
software trigger */
}
/* Delay n milliseconds
* The CPU core clock is set to MCGFLLCLK at 41.94
MHz in SystemInit().
*/
void delayMs(int n) {
int i;
int j;
for(i = 0 ; i < n; i++)
for (j = 0; j < 7000; j++) {}
}
```

Basic Programming Techniques – DAC

```
/* Use DAC to generate sine wave
* The program calculates the lookup table to
generate
* sine wave.
*/

#include “MKL25Z4.h”
#include <math.h>
void DAC0_init(void);
#define WAVEFORM_LENGTH 256
int sinewave[WAVEFORM_LENGTH];
int main(void) {
void delayMs(int n);
int i;
float fRadians;
const float M_PI = 3.1415926535897;
```

Basic Programming Techniques – DAC

```
/* construct data table for a sine wave */
fRadians = ((2 * M_PI) / WAVEFORM_LENGTH);
for (i = 0; i < WAVEFORM_LENGTH; i++) {
sinewave[i] = 2047 * (sinf(fRadians * i) + 1);
}
DAC0_init(); /* Configure DAC0 */
while (1) {
for (i = 0; i < WAVEFORM_LENGTH; i++) {
/* write value to DAC0 */
DAC0->DAT[0].DATL = sinewave[i] & 0xff; /* write
low byte */
DAC0->DAT[0].DATH = (sinewave[i] >> 8) & 0x0f;/*
write high byte */
delayMs(1); /* delay 1ms */
}
}
}
```

Basic Programming Techniques – DAC

```
void DAC0_init(void) {
SIM->SCGC6 |= 0x80000000; /* clock to DAC module */
DAC0->C1 = 0; /* disable the use of buffer */
DAC0->C0 = 0x80 | 0x20; /* enable DAC and use
software trigger */
}
/* Delay n milliseconds
* The CPU core clock is set to MCGFLLCLK at 41.94
MHz in SystemInit().
*/
void delayMs(int n) {
int i;
int j;
for(i = 0 ; i < n; i++)
for (j = 0; j < 7000; j++) {}
}
```

Basic Programming Techniques – DAC