# Instalación de FreeRTOS en Kinetis KL25Z

Sistemas de Tiempo Real

Ing. en Computación

Año 2019

Aguilar Sergio 765/7

Ailigo Oriana 1202/8

# Resumen

1. **Objetivos e Introducción**

Este trabajo tiene como objetivo la instalación y prueba del sistema operativo de tiempo real FreeRTOS en la placa de desarrollo Kinetis KL25Z.

Para esto se harán n proyectos, entre los cuales demostraremos el funcionamiento de los siguientes aspectos de FreeRTOS:

* Creación de tareas
* Estado de las tareas
* Uso de las prioridades de las tareas
* Manejo de semáforos
* Funciones de FreeRTOS desde ISRs

Se utilizarán los siguientes periféricos del KL25Z

* Led RGB
* Capacitive touch slider

***Palabras clave:* KL25Z,FreeRtos.**

# 

# La realización del trabajo se dividió en 3 etapas:

# Investigación: investigación sobre FreeRTOS y Kinetis KL25Z para estar informados a la hora de llevar a cabo la etapa de desarrollo.

# Desarrollo: se llevaron a cabo los objetivos específicos de cada proyecto.

# Conclusión: se analizaron los resultados obtenidos para poder sacar conclusiones del trabajo

# 

**2. Investigación**

**2.1 Conceptos Básicos**

**Sistema de tiempo real:** Un sistema de tiempo real es un sistema que tiene que reaccionar a estímulos del ambiente (incluso el avance del tiempo físico) dentro de intervalos marcados por el ambiente. Debe ser lo suficientemente rápido para cumplir los requerimientos temporales, pero que un sistema sea rápido no implica que sea de tiempo real. Es decir, para que el funcionamiento del sistema sea correcto no basta con que las acciones sean correctas, sino que tienen que ejecutarse dentro del intervalo de tiempo especificado (corrección lógica y temporal)

**Sistema operativo de tiempo real (RTOS):** Un RTOS es un sistema operativo que ha sido desarrollado para sistemas de tiempo real. Esto significa que hace lo mismo que un sistema operativo común, pero además brinda herramientas para que los programas de aplicación puedan cumplir compromisos temporales definidos por el programador. Entonces, podemos afirmar que el objetivo de un RTOS no es el mismo que el de un SO convencional.

Los RTOS se emplean generalmente cuando hay que administrar varias tareas simultáneas con plazos de tiempo estrictos.

**Sistemas embebidos**: es un sistema de computación diseñado para realizar una o algunas pocas funciones dedicadas, frecuentemente en un sistema de computación en tiempo real. Al contrario de lo que ocurre con los ordenadores de propósito general (como por ejemplo una computadora personal o PC) que están diseñados para cubrir un amplio rango de necesidades, los sistemas embebidos se diseñan para cubrir necesidades específicas. Debido a los limitados recursos, generalmente no tienen un sistema operativo completo, sino sólo el subconjunto de estos que pueden manejar ventajosamente.En algunos casos el OS no es un programa en sí mismo sino que es un conjunto de funciones que se ejecutan solo en momentos determinados del programa.

**2.1.2 Ventajas de utilizar un RTOS**

● **Para cumplir con compromisos temporales estrictos:** el RTOS ofrece funcionalidad para asegurar que una vez ocurrido un evento, la respuesta ocurra dentro de un tiempo acotado. Es importante aclarar que esto no lo hace por sí solo sino que brinda al programador herramientas para hacerlo de manera más sencilla que si no hubiera un RTOS. Esto implica que una aplicación mal diseñada puede fallar en la atención de eventos aún cuando se use un RTOS.

● **Para no tener que manejar el tiempo “a mano”**: el RTOS absorbe el manejo de temporizadores y esperas, de modo que hace más fácil al programador el manejo del tiempo.

● **Tarea Idle**: cuando ninguna de las tareas requiere del procesador, el sistema ejecuta una tarea llamada idle u ociosa. Esto me permite fácilmente contabilizar el nivel de ocupación del CPU, poner al mismo en modo de bajo consumo o correr cualquier tarea que pudiera ser de utilidad para el sistema cuando no debe atender ninguno de sus eventos.

● **Multitarea**: simplifica sobremanera la programación de sistemas con varias tareas.

● **Escalabilidad**: al tener ejecución concurrente de tareas se pueden agregar las que haga falta, teniendo el único cuidado de insertarlas correctamente en el esquema de ejecución del sistema.

● **Mayor reutilizabilidad del código**: si las tareas se diseñan bien (con pocas o ninguna dependencia) es más fácil incorporarlas a otras aplicaciones.

**2.1.3 FreeRTOS**

FreeRTOS es un sistema operativo de tiempo real para sistemas embebidos, implementado para 34 microcontroladores. Se distribuye bajo la licencia GPL. Está diseñado para ser pequeño y simple; su kernel está formado por 3 o 4 archivos en C (con algunas funciones en Assembler), lo que lo hace portable a diferentes arquitecturas, fácil de leer y mantener.

**Ventajas:**

● **Es de código abierto:** está bajo contínuo desarrollo, y no hay costo de implementación.

● **Pensado para microcontroladores:** es liviano en tamaño de código y en uso de RAM.

● **Amplia comunidad de usuarios:** ayuda a resolver problemas que surgen.

● **Simple**: el kernel está formado por 3 o 4 archivos C.

**2.1.4 Tareas en FreeRTOS**

Las tareas en FreeRTOS deben ser de la siguiente manera:

void **vNombreTarea**( void \***pvParametros** ){

**for**( ;; ){

-- Código de la tarea. --

}

}

Todas las funciones que implementan una tarea deben cumplir con la implementación previa. El parámetro puede ser utilizado para pasar datos a la tarea. ***Las tareas no deben retornar, por este motivo suelen implementarse dentro de un loop infinito***. Si una tarea deja de ser necesaria, puede borrarse explícitamente.

● Creación de tareas: **xTaskCreate**()

● Borrado de tareas: **vTaskDelete**()

Las tareas tienen una prioridad de ejecución, donde 0 es la menor prioridad (se recomienda usar referencias a **tskIDLE\_PRIORITY** +1,+2, etc).

Se pueden crear múltiples instancias de una misma función de tarea, las cuales pueden recibir un parámetro que las caracterice.

Cuando una tarea deja de estar en ejecución, las referencias a variables alojadas en su pila dejan de ser válidas. Al ser funciones de C, se aplican las mismas reglas de visibilidad de variables, entonces se puede compartir memoria entre las tareas usando variables globales. En este caso se debe cuidar el acceso a este recurso compartido.

**La tarea IDLE**: es una tarea que se crea automáticamente con el primer llamado a **xTaskCreate** (). Es el encargado de liberar memoria alocada por el RTOS a tareas que ya fueron borradas. **La tarea Idle NO DEBE BLOQUEAR NI SUSPENDER**, ya que si lo hiciera, no quedaría ninguna tarea en estado Running, y siempre debe haber una.

**Tareas IDLE HOOK:** ciertas funcionalidades de la aplicación pueden ser necesarias implementarlas cuando el sistema está ocioso, por ejemplo:

● Poner al CPU en estado de bajo consumo

● Medir el tiempo disponible del sistema para agregarle más tareas

● Funcionalidad específica de la aplicación

Las tareas IDLE HOOK son funciones que pueden ser llamadas en cada ciclo de la tarea IDLE. Si se quiere ejecutar alguna tarea en la misma prioridad que la tarea IDLE existen dos opciones:

● Implementar una tarea IDLE HOOK:

● Crear una tarea con la misma prioridad que la tarea IDLE.

**2.1.5 Planificación (Scheduling)**

**Planificador**: es la parte del kernel responsable de decidir qué tarea debe ejecutarse en cada instante de tiempo. El planificador puede suspender y luego reanudar la ejecución de las tareas tantas veces como crea conveniente a lo largo de la ejecución del programa.

**Política de planificación:** es el algoritmo utilizado por el scheduler para decidir qué tarea se ejecuta en cada momento.

FreeRTOS usa un algoritmo llamado **Fixed priority preemptive scheduling.** Este algoritmo NO MODIFICA las prioridades de las tareas (salvo en ocasiones particulares). Esto simplifica el análisis de ejecución del sistema, a la vez que transfiere al programador la total responsabilidad del cumplimiento de los plazos.

SIEMPRE ejecuta la tarea de mayor prioridad que está en condiciones de ejecutarse. Esto puede hacer que tareas de menor prioridad no reciban ningún tiempo de ejecución. A esto se le denomina “starvation” de la tarea.

**Si la aplicación está en modo preemptive,** al cabo de un tiempo el scheduler va a retirar la tarea en ejecución solo si hubiera una de igual prioridad en condiciones de ejecutarse.

Si hubiere alguna(s) de menor prioridad, no va(n) a recibir tiempo alguno hasta que todas las de mayor prioridad salgan del estado Ready.

Se define una frecuencia llamada **configTICK\_RATE\_HZ**. La inversa es el lapso que el scheduler asigna a cada tarea para ejecutarse antes de verificar si debe quitarla del estado Running. La definición de esta frecuencia es un punto importante del diseño.

Un valor muy bajo de **TICK\_RATE** hace que el sistema sea lento para responder a los eventos temporales.

Si la tarea en ejecución no cede el control del CPU, la próxima tarea a ejecutarse debe esperar a que termine el lapso actual. Por esto es que para mantener al sistema ágil cada tarea debe usar el CPU lo mínimo e indispensable, o más bien cederlo todas las veces que no lo necesite.

Un valor muy alto de **TICK\_RATE** hace que el scheduler trabaje más seguido, ocupando más tiempo del CPU. Este tiempo debiera ser despreciable respecto del que consume la aplicación. El **TICK\_RATE** del sistema termina siendo un valor de compromiso entre estas dos cotas.

**2.1.6 Estados de las tareas**

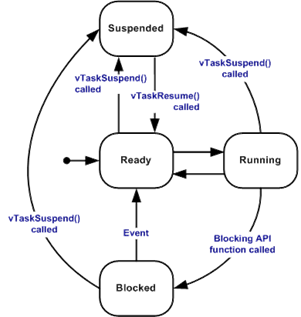
Una tarea puede estar en alguno de los siguientes estados:

● **Running** (Corriendo): cuando una tarea está en ejecución, se dice que está en estado Running. Es la tarea que utiliza el procesador.

● **Ready** (Listo): en este estado están las tareas que están listas para ejecutarse, pero no se están ejecutando porque hay otras tareas de mayor o igual prioridad en el estado Running.

● **Blocked** (Bloqueado): cuando una tarea está esperando por un evento. Puede setearse un “timeout”, para que luego de un determinado tiempo, las tareas se desbloqueen sin que ocurra el evento.

● **Suspended** (Suspendidas): las tareas en este estado no están disponibles para ser elegidas por el scheduler. Para entrar y salir de este estado deben llamarse a las funciones vTaskSuspend() u vTaskResume() respectivamente. No puede especificarse un “timeout”. El kernel no suspende ni reanuda tareas por sí mismo, hay que llamar a las funciones mencionadas previamente.



**2.1.7 Sincronización entre tareas y eventos**

Para sincronizar la ejecución de tareas con eventos, FreeRTOS provee el recurso de los semáforos.

Semáforos: su función es restringir el acceso a una sección particular del programa. El uso de semáforos es una convención a la que deben adherir todas las tareas involucradas.

Los semáforos tienen dos operaciones asociadas:

● Tomar el semáforo (sería equivalente a ponerlo en rojo)

● Dar el semáforo (sería equivalente a ponerlo en verde)

Es responsabilidad del programador tanto tratar de tomar el semáforo antes de acceder a la sección crítica como darlo al salir de la misma.

Entonces, una tarea que debe esperar un evento, lo único que debe hacer es tomar un semáforo que no esté disponible. Luego la tarea o interrupción que descubre o genera el evento tan solo debe liberar el semáforo

Funciones para semáforos en FreeRTOS:

● vSemaphoreCreateBinary(xSemaphoreHandle xSem) → crea semáforo binario, que se manejará con el nombre “xSem”.

● xSemaphoreTake(xSemaphoreHandle **xSem**,portTickType **xBlockTime**) → toma el semáforo “xSem”. xBlockTime indica la cantidad de ticks que la tarea espera a que le liberen el semáforo. Si se quiere esperar un tiempo infinito, utilizar **portMAX\_DELAY**.

● xSemaphoreGive( xSemaphoreHandle xSem ) → libera el semáforo xSem.

**2.1.8 Comunicación entre las tareas**

Como las tareas son funciones de C aplican las mismas reglas de visibilidad para las variables. El uso de variables globales como mecanismo de comunicación presenta dos problemas:

● Cuidar acceso al recurso.

● No permite sincronización entre tareas que esperan lecturas/escrituras de un dato para procesarlo.

A partir de estos problemas surgen las colas, como mecanismo de comunicación entre tareas sincronizadas

**Colas**: Son visibles por todas las tareas (deben ser creadas en forma global). Incorporan el mecanismo de sincronización. De la misma manera que un semáforo, se puede bloquear al leer / escribir datos de una cola.

**Características:**

● Se crean mediante la función **xQueueCreate**()

● Trabajan con **cualquier tipo de datos**, no solo los nativos.

● Los mensajes se pasan en la cola POR COPIA

● Se debe **dimensionar la cantidad de elementos de la cola y el tamaño de los   
 mismos cuando se la crea**. Estos parámetros no pueden cambiarse luego.

● Se la debe crear antes de iniciar el scheduler.

● Se la identifica mediante un **xQueueHandle** devuelto por la llamada a   
 **xQueueCreate**.

**2.1.9 Funciones para colas en FreeRTOS:**

● **xQueueReceive**(): lee y quita el dato de la cola.

● **xQueuePeek**(): lee pero no quita el dato de la cola.

● **xQueueSendToBack**(): escritura normal a una cola.

● **xQueueSendToFront**(): pone el mensaje en la posición de salida. Sería como apilar el dato.

● **uxQueMessagesWaiting**(): consulta los mensajes disponibles en las colas.

**Operaciones con colas que bloquean:**

● Intentar leer de una cola vacía

● Intentar escribir a una cola llena

Por esto, se les puede asignar un **blockTime** igual que a un semáforo. En este caso también debe chequearse el valor de retorno de la operación.

**2.1.10 Interrupciones**

Las rutinas de servicio de interrupciones (ISR) deben ejecutarse por completo, por lo tanto, no debe forzarse un cambio de contexto dentro de una ISR.

Para esto, todas las funciones de FreeRTOS que podrían generar un cambio de contexto tienen una variante ISR-safe, que es el mismo nombre de la función con el sufijo FromISR:

**● xSemaphoreGiveFromISR**

**● xQueueSendFromISR**

**● xQueueReceiveFromISR**

Las operaciones cuyo objetivo es justamente bloquear a la tarea no deben usarse en una ISR y no tienen una variante FromISR.

**● No hay xSemaphoreTakeFromISR**

**● No hay xDelayFromISR**

**● No hay taskYIELDFromISR**

**Fuentes**

<http://www.freertos.org/>  
<http://es.wikipedia.org/wiki/Sistema_embebido>  
<http://es.wikipedia.org/wiki/Sistema_operativo_de_tiempo_real>  
<http://ieee.eie.fceia.unr.edu.ar/PDF_RTOS.pdf>  
<http://www.sase.com.ar/2011/files/2010/11/SASE2011-Introduccion_RTOS.pdf>  
 <http://en.wikipedia.org/wiki/FreeRTOS>

**2.2 Investigación - KL25Z**

Introducción

La familia Kinetis K70 está basada en microcontroladores de 32-bits de bajo consumo, destinado para aplicaciones de control industrial, sistemas de navegación y monitoreo de sistemas médicos, principalmente.

Están basados en un ARM Cortex-M4 core, y además incluyen una unidad de punto flotante y DSP.

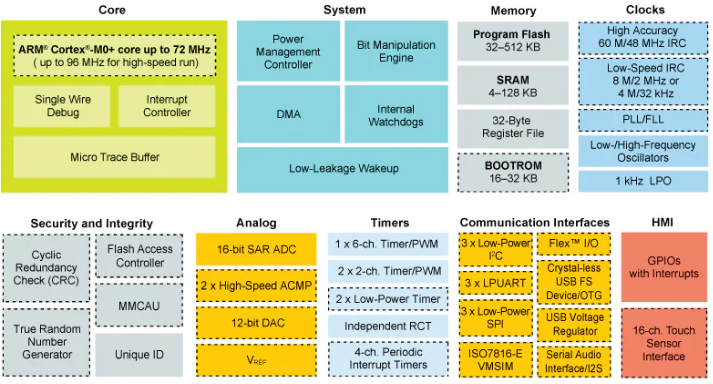
Incluyen un conjunto de periféricos para trabajar con señales analógicas, comunicaciones y control. Se destacan por incluir un controlador de gráficos LCD integrado, USB 2.0 y IEEE® 1588 Ethernet MAC.

**2.2.1 Módulos Funcionales**

**Introducción:**

La familia Kinetis K25Z está basada en microcontroladores de 32-bits de baja potencia y alto rendimiento, Esta serie combina el rendimiento excepcional de baja potencia y la eficiencia energética del núcleo Arm® Cortex®-M0 + con los conjuntos periféricos, la habilitación y la escalabilidad de la cartera Kinetis MCU, lo que la convierte en una solución ideal para aplicaciones de Internet de las cosas (IoT) .

Incluyen un conjunto de periféricos para trabajar con señales analógicas, comunicaciones y control.



**2.2.2 Características:**

**Ultra-baja-Potencia:**

* Núcleo Cortex-M0 + de 32 bits de la siguiente generación: 2 veces más CoreMark / mA que la arquitectura de 8/16 bits más cercana
* El puerto de acceso de E / S rápido de ciclo único facilita el golpeteo de bits y la emulación de protocolo de software, manteniendo una apariencia de 8 bits.
* Múltiples modos flexibles de baja potencia, además de incluir la nueva opción de reloj de computadora que reduce la potencia dinámica al colocar periféricos en un modo de parada asíncrono.
* LPUART, SPI, I²C, FlexIO, ADC, DAC, temporizador LP y DMA admiten el funcionamiento en modo de baja potencia sin activar el núcleo.

**Memoria:**

* Hasta 512 KB de flash, hasta 128 KB de SRAM
* Circuitos de seguridad para evitar el acceso no autorizado a RAM y contenido flash
* Hasta 32 KB de ROM y el gestor de arranque incorporado simplifica el esfuerzo para programar MCU y permite actualizaciones flash fáciles

**Performance:**

* Cortex-M0 + core, frecuencia central de 72MHz / 96MHz sobre voltaje completo y rango de temperatura (-40ºC a + 105ºC)
* Motor de manipulación de bits para mejorar el manejo de bits de los módulos periféricos.
* El conjunto de instrucciones de pulgar combina una alta densidad de código con un rendimiento de 32 bits
* Hasta 8 ch. DMA para servicio periférico y de memoria con carga de CPU reducida y rendimiento del sistema más rápido
* El COP con reloj independiente protege contra el sesgo del reloj o la fuga de código para aplicaciones a prueba de fallas

**Señales-Mixtas:**

* ADC de hasta 16 bits con resolución configurable, tiempo de muestreo y velocidad / potencia de conversión
* Sensor de temperatura integrado
* Comparador de alta velocidad con DAC interno de 6 bits
* DAC de 12 bits con soporte DMA
* Referencias de voltaje de 1.2 V y 2.1 V (Vref)

**Tiempo y control:**

* Dos de 6 ch. y un módulo PWM de temporizador de baja potencia de 2 canales y 16 bits con soporte DMA
* 2 ch. El temporizador de interrupción periódica de 32 bits proporciona la base de tiempo para la programación de tareas RTOS o la fuente de activación para la conversión de ADC
* El temporizador de baja potencia permite la operación en todos los modos de potencia, excepto VLLS0
* Reloj en tiempo real con calendario.

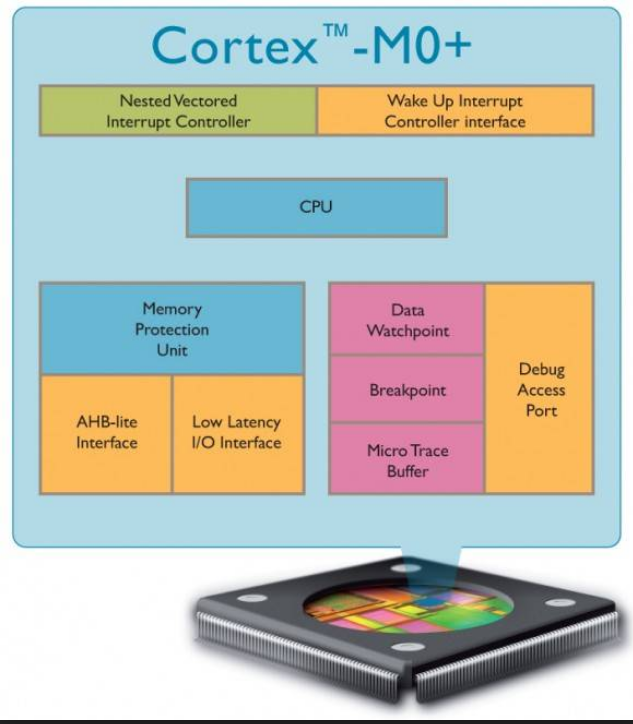
**Interfaz Máquina-Humano**

* La interfaz de detección táctil capacitiva admite hasta 16 electrodos externos y transferencia de datos DMA
* GPIO con soporte de interrupción de pin, capacidad de solicitud de DMA y otras opciones de control de pin

**Conectividad y comunicación:**

* USB 2.0 On-The-Go (velocidad completa) con regulador USB de bajo voltaje integrado suministra hasta 120 mA fuera del chip a 3.3 voltios para alimentar componentes externos desde una entrada de 5 voltios o un dispositivo USB 2.0 (velocidad completa) con reloj función de recuperación sin la necesidad de cristal externo
* FlexIO para emulación periférica en serie universal o personalizada
* Referencia de reloj interno de alta precisión para soportar comunicaciones de alto rendimiento
* Hasta 3 I2C con soporte DMA, hasta 1Mbps y compatible con funciones SMBus V2
* Hasta 3 LPUART con soporte DMA
* Hasta 3 SPI con soporte DMA
* Módulo I²S para aplicaciones de audio.
* El módulo FlexIO admite una amplia gama de protocolos que incluyen la generación de formas de onda UART, I²C, SPI, I²S, PWM
* Interfaz de tarjeta inteligente VMSIM (ISO7816-E)

**2.2.3 Módulos del Cortex M0+**

****

**1) NVIC El control de interrupción de vector anidado (NVIC)**

(NVIC) proporciona una interfaz entre las fuentes de interrupción externas al núcleo

(periféricos y pines externos) y el núcleo. La prioridad para cada fuente de interrupción es programable (cuatro niveles). Admite hasta 32 interrupciones externas y admite interrupción no enmascarable.

**2) MPU (unidad de protección de memoria)**

Unidad encargada de crear hasta 8 Niveles y subniveles de seguridad para accesos de distintas secciones de memoria solo con los permisos requeridos.

**3) MODOS SLEEP**

3 modos de Sleep : Sleep, deep sleep y deep sleep con SRGB.

**4) La Unidad de punto de observación de datos (DWT)**

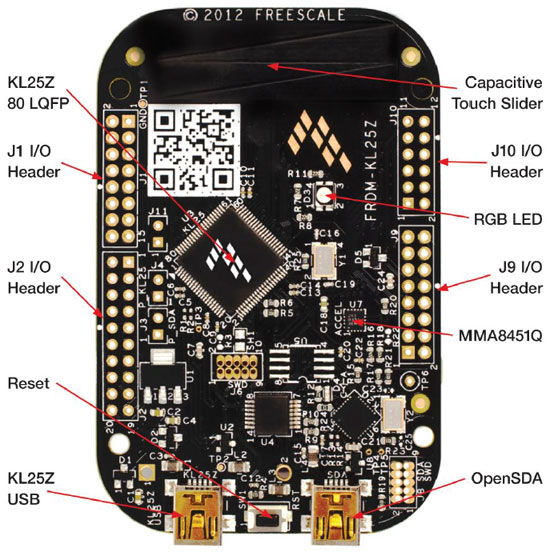
es una unidad de depuración que proporciona hasta 2 puntos de observación para el seguimiento de datos y la creación de perfiles del sistema (estadísticas de CPU, muestra de PC).

**5) La unidad de Breakpoint (BPU)**

permite hasta cuatro puntos de interrupción de hardware para generar eventos de depuración

**6) El Micro Trace Buffer (MTB)**

proporciona un seguimiento de instrucciones simple. Utiliza una pequeña porción de SRAM para el búfer de rastreo.



**Figura x: Vista frontal de la placa KL25Z**

**Fuentes:**

* Freescale\_ARM\_Cortex\_M\_Embedded ( manual reference)
* FRMD-KL25Z.pdf (manual user)
* Kinetis KL25: 48MHz Arm ® Cortex®-M0+ 32-128KB Flash 32-80 pin ( manual reference)
* NXP Sensor Fusion Library for Kinetis MCUs for release 5.0 (datasheet)

**3. Desarrollo**