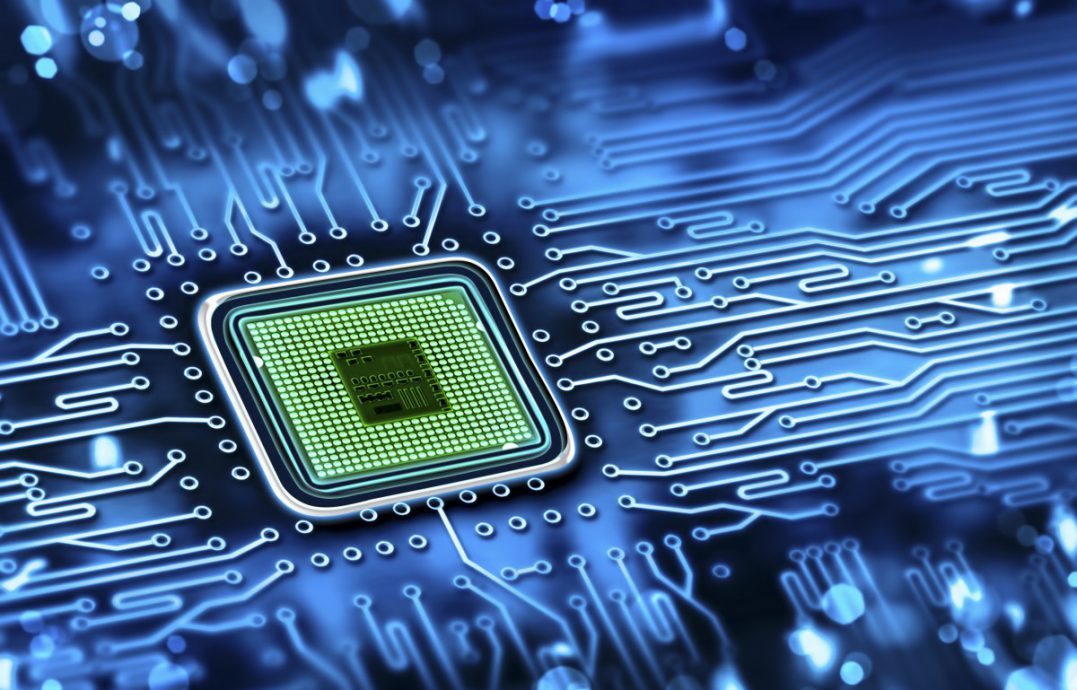
**Lab Report**

****

**Embedded Systems**

**Himanshu Rai**

**111601032 25th October, 2019**

**Objective**

Using RTX (*Real-Time eXecutive*) CMSIS-RTOS (*Real-Time Operating System*) for various tasks.

**Theory**

**Operating System**

An operating system (OS) is system software that manages computer hardware, software resources, and provides common services for computer programs. Operating systems are found on many devices that contain a computer – from cellular phones and video game consoles to web servers and supercomputers.

There are many types of OSs in the world. *Embedded operating systems* (EOS) are designed to be used in embedded computer systems. They are designed to operate on small machines like PDAs with less autonomy. They are able to operate with a limited number of resources. They are very compact and extremely efficient by design. A *real-time operating system* (RTOS) is an operating system that guarantees to process events or data by a specific moment in time. A real-time operating system may be single- or multi-tasking, but when multitasking, it uses specialized scheduling algorithms so that a deterministic nature of behavior is achieved. An event-driven system switches between tasks based on their priorities or external events while time-sharing operating systems switch tasks based on clock interrupts. RTOS are subset of EOS.

**CMSIS-RTOS**

CMSIS-RTOS is an API specification that enables middleware to be designed that works with multiple RTOS products. The CMSIS-RTOS itself is not a product but companies can build an RTOS that is based on CMSIS-RTOS APIs, or add a wrapper layer on top of their own OS APIs to do the same things.

**RTX**

The Keil RTX Real-Time Kernel is a royalty-free RTOS targeted for microcontroller applications. The RTX libraries and source files are included in the CMSIS-PACK package. So when the CMSIS pack for Keil MDK is installed, the RTX is also included. The RTX kernel is supported on all Cortex-M processors.

The Cortex-M version of RTX kernel has the following features:

* SysTick timer support
* No interrupt lock out in Cortex-M versions (the OS do not need to disable the interrupts for any OS operations)

**Signalling**

Signals are a limited form of inter-process communication (IPC), When a signal is sent, the operating system interrupts the target process' normal flow of execution to deliver the signal. Execution can be interrupted during any non-atomic instruction. If the process has previously registered a signal handler, that routine is executed. Otherwise, the default signal handler is executed.

Embedded programs may find signals useful for inter-process communications, as the computational and memory footprint for signals is small. Signals are similar to interrupts, the difference being that interrupts are mediated by the processor and handled by the kernel while signals are mediated by the kernel (possibly via system calls) and handled by processes. The kernel may pass an interrupt as a signal to the process that caused it.

**Semaphores**

In computer science, a semaphore is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent system such as a multitasking operating system. A semaphore is simply a variable. This variable is used to solve critical section problems and to achieve process synchronization in the multi processing environment. A trivial semaphore is a plain variable that is changed (for example, incremented or decremented, or toggled) depending on programmer-defined conditions.

**Source Codes**

**LED\_init.h :** Header file used in other programs

#include<MKL25Z4.h>

**void** LED\_Config() {

SIM->SCGC5 |= 1 << 10; // will activate the port B

SIM->SCGC5 |= 1 << 12; // will activate the port D

// set the pins B18, B19, D1 as gpio

PORTB->PCR[18] &= 0xFFFFF8FF; // set 8th, 9th and 10th bit of PCR[18] to 0

PORTB->PCR[18] |= 1 << 8; // set 8th bit of PCR[18] to 1

PORTB->PCR[19] &= 0xFFFFF8FF; // set 8th, 9th and 10th bit of PCR[19] to 0

PORTB->PCR[19] |= 1 << 8; // set 8th bit of PCR[19] to 1

PORTD->PCR[1] &= 0xFFFFF8FF; // set 8th, 9th and 10th bit of PCR[1] to 0

PORTD->PCR[1] |= 1 << 8; // set 8th bit of PCR[1] to 1

// set the ports as output port

PTB->PDDR |= 1 << 18; // set the 18th bit of PDDR to 1 for output

PTB->PDDR |= 1 << 19; // set the 18th bit of PDDR to 1 for output

PTD->PDDR |= 1 << 1; // set the 1th bit of PDDR to 1 for output

}

**void** LED\_Set() {

PTB->PSOR |= 1 << 18; // red LED

PTB->PSOR |= 1 << 19; // green LED

PTD->PSOR |= 1 << 1; // blue LED

}

**void** LED\_Clear() {

PTB->PCOR |= 1 << 18; // red LED

PTB->PCOR |= 1 << 19; // green LED

PTD->PCOR |= 1 << 1; // blue LED

}

**void** LED\_On(uint32\_t led) {

// function to turn the specified LED on

**switch**(led) {

**case** 0: PTB->PCOR |= 1 << 18; **break**; // red LED

**case** 1: PTB->PCOR |= 1 << 19; **break**; // green LED

**case** 2: PTD->PCOR |= 1 << 1; **break**; // blue LED

}

}

**void** LED\_Off(uint32\_t led) {

// function to turn the specified LED off

**switch**(led) {

**case** 0: PTB->PTOR |= 1 << 18; **break**; // red LED

**case** 1: PTB->PTOR |= 1 << 19; **break**; // green LED

**case** 2: PTD->PTOR |= 1 << 1; **break**; // blue LED

}

}

**PROGRAM 1 :** Blinky.c with RTX - Two threads running in parallel toggling the Red and Green LED on board.

The blinky program has the following two threads:

* main() - starts second thread blinky and toggles the red LED
* blinky() - toggles the green LED

#include <MKL25Z4.H>

#include "cmsis\_os.h" // Include header file for RTX CMSIS-RTOS

#include "LED\_init.h"

// System runs at 48MHz

// LED #0, #1 are port B, LED #2 is port D

**extern** **void** LED\_Config(**void**);

**extern** **void** LED\_Set(**void**);

**extern** **void** LED\_Clear(**void**);

**extern** \_\_INLINE **void** LED\_On(uint32\_t led);

**extern** \_\_INLINE **void** LED\_Off(uint32\_t led);

/\* Thread IDs \*/

osThreadId t\_blinky; // Declare a thread ID for blinky

/\* Function declaration \*/

**void** blinky(**void** **const** \*argument); // Thread

// --------------------------------------------------------

// Blinky

**void** blinky(**void** **const** \*argument) {

**while**(1) {

LED\_On(1); // Green LED on

osDelay(500); // delay 500 msec

LED\_Off(1); // Green LED off

osDelay(500); // delay 500 msec

} // end while

} // end of blinky

// define blinky as thread function

osThreadDef(blinky, osPriorityNormal, 1, 0);

// --------------------------------------------------------

**int** main(**void**)

{

SystemCoreClockUpdate();

// Configure LED outputs

LED\_Config();

// Create a task "blinky" and assign thread ID to t\_blinky

t\_blinky = osThreadCreate(osThread(blinky), NULL);

**while**(1) {

LED\_On(0); // Red LED on

osDelay(200); // delay 200 msec

LED\_Off(0); // Red LED off

osDelay(200); // delay 200 msec

};

}

**PROGRAM 2 :** Simple signal event communication - Two threads turning green and red LEDs on/off alternately by communication through signalling.

A thread enters WAITING state when it executes the function *osSignalWait*. One of the input parameters, a 32-bit value called “signals” defines the signal events required to put the thread back to READY state. Each bit (apart from the MSB) of the “signals” parameter defines the signal events required and if this parameter is set to 0 any signal event can put this thread back to READY state.

#include <MKL25Z4.H>

#include "cmsis\_os.h" // Include header file for RTX CMSIS-RTOS

#include "LED\_init.h"

// System runs at 48MHz

// LED #0, #1 are port B, LED #2 is port D

**extern** **void** LED\_Config(**void**);

**extern** **void** LED\_Set(**void**);

**extern** **void** LED\_Clear(**void**);

**extern** \_\_INLINE **void** LED\_On(uint32\_t led);

**extern** \_\_INLINE **void** LED\_Off(uint32\_t led);

/\* Thread IDs \*/

osThreadId t\_blinky; // Declare a thread ID for blinky

/\* Function declaration \*/

**void** blinky(**void** **const** \*argument); // Thread

// --------------------------------------------------------

// Blinky

**void** blinky(**void** **const** \*argument) {

**while**(1) {

osSignalWait(0x0001, osWaitForever);

LED\_On(1); // Green LED on

osDelay(500); // delay 500 msec

LED\_Off(1); // Green LED off

} // end while

} // end of blinky

// define blinky as thread function

osThreadDef(blinky, osPriorityNormal, 1, 0);

// --------------------------------------------------------

**int** main(**void**) {

SystemCoreClockUpdate();

// Configure LED outputs

LED\_Config();

// Create a task "blinky" and assign thread ID to t\_blinky

t\_blinky = osThreadCreate(osThread(blinky), NULL);

**while**(1){

LED\_On(0); // Red LED on

osDelay(500); // delay 500 msec

LED\_Off(0); // Red LED off

osSignalSet(t\_blinky, 0x0001); // Set Signal

osDelay(500); // delay 500 msec

};

}

**PROGRAM 3 :** Semaphore communication - Three threads turning green, blue and red LEDs on/off through communication using a semaphore with 2 tokens (at a time only 2 of the three LEDs can be on).

#include <MKL25Z4.H>

#include "cmsis\_os.h" // Include header file for RTX CMSIS-RTOS

#include "LED\_init.h"

// System runs at 48MHz

// LED #0, #2 are port B, LED #1 is port D

**extern** **void** LED\_Config(**void**);

**extern** **void** LED\_Set(**void**);

**extern** **void** LED\_Clear(**void**);

**extern** \_\_INLINE **void** LED\_On(uint32\_t led);

**extern** \_\_INLINE **void** LED\_Off(uint32\_t led);

/\* Thread IDs \*/

osThreadId t\_blinky\_red; // Declare a thread ID for blinky

osThreadId t\_blinky\_green; // Declare a thread ID for blinky

osThreadId t\_blinky\_blue; // Declare a thread ID for blinky

/\* Function declaration \*/

**void** blinky\_red(**void** **const** \*argument); // Thread

**void** blinky\_green(**void** **const** \*argument); // Thread

**void** blinky\_blue(**void** **const** \*argument); // Thread

/\* Declare Semaphore \*/

osSemaphoreDef(two\_LEDs); // Declare a Semaphore for LED control

/\* Semaphore IDs \*/

osSemaphoreId two\_LEDs\_id; // Declare a Semaphore ID for LED control

// --------------------------------------------------------

// Blinky

**void** blinky\_red(**void** **const** \*argument) {

**while**(1) {

osSemaphoreWait(two\_LEDs\_id, osWaitForever);

LED\_On(0); // Red LED on

osDelay(400); // delay 400 msec

LED\_Off(0); // Red LED off

osSemaphoreRelease(two\_LEDs\_id);

osDelay(600); // delay 600 msec

} // end while

} // end of blinky

**void** blinky\_green(**void** **const** \*argument) {

**while**(1) {

osSemaphoreWait(two\_LEDs\_id, osWaitForever);

LED\_On(1); // Green LED on

osDelay(400); // delay 400 msec

LED\_Off(1); // Green LED off

osSemaphoreRelease(two\_LEDs\_id);

osDelay(600); // delay 600 msec

} // end while

} // end of blinky

**void** blinky\_blue(**void** **const** \*argument) {

**while**(1) {

osSemaphoreWait(two\_LEDs\_id, osWaitForever);

LED\_On(2); // Blue LED on

osDelay(400); // delay 400 msec

LED\_Off(2); // Blue LED off

osSemaphoreRelease(two\_LEDs\_id);

osDelay(600); // delay 600 msec

} // end while

} // end of blinky

// define blinky as thread function

osThreadDef(blinky\_red, osPriorityNormal, 1, 0);

osThreadDef(blinky\_green, osPriorityNormal, 1, 0);

osThreadDef(blinky\_blue, osPriorityNormal, 1, 0);

// --------------------------------------------------------

**int** main(**void**)

{

SystemCoreClockUpdate();

// Configure LED outputs

LED\_Config();

// Create Semaphore with 2 tokens

two\_LEDs\_id = osSemaphoreCreate(osSemaphore(two\_LEDs), 2);

// Create threads "blinky\_xxx" and assign thread ID to t\_blinky\_xxx

t\_blinky\_red = osThreadCreate(osThread(blinky\_red), NULL);

t\_blinky\_green = osThreadCreate(osThread(blinky\_green), NULL);

t\_blinky\_blue = osThreadCreate(osThread(blinky\_blue), NULL);

// Terminate main

osThreadTerminate(osThreadGetId());

**while**(1){

osDelay(1000); // delay 1000 msec

};

}

**Results**

The desired outputs were obtained and demonstrated during the lab.

**References**

* ARM University program
* Wikipedia
* Freescale (NXP) user manual
* The Definitive Guide to ARM CortexM0/M0+