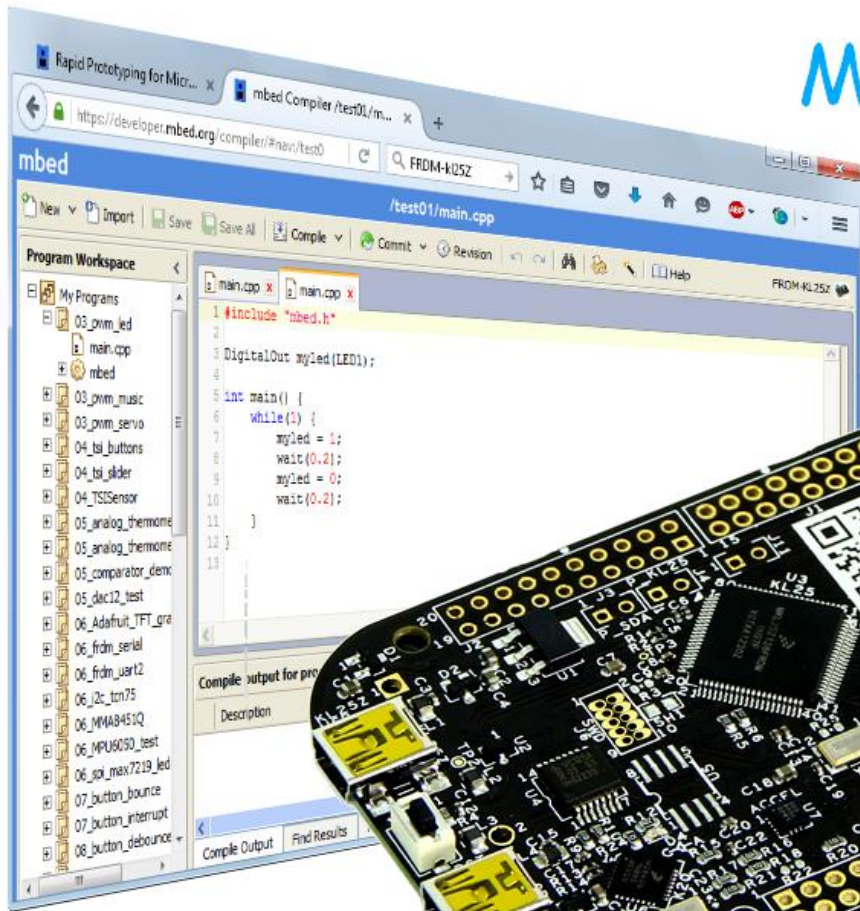


Mikrovezérlő programozás

ARM[®]mbed[™]
környezetben



10. RTOS mutexek, szemaforok

Programszálak szinkronizálása

Az előző fejezet végén bemutatott példaprogram három programszálát futtatott, ám ezek egymástól függetlenül végezték feladatukat, ugyanúgy, mintha három külön mikrovezérlő kártyán futottak volna.

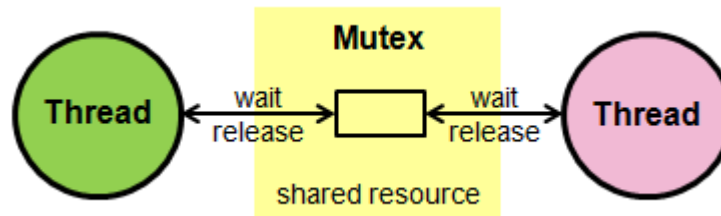
Az **RTOS** alkalmazások többségénél azonban szükség lehet rá, hogy tevékenységeiket egymáshoz szinkronizálják, a közös használatú erőforrásokon megosztozzanak, vagy adatot cseréljenek egymással.

Az **mbed-RTOS** fejlett eszköztárat biztosít ezekhez a feladatokhoz (jelzőbitek, mutex, szemafor, levelesláda és üzenetek formájában), melyek közül most a megosztott erőforrások kezelésére használt **mutex** használatát tárgyaljuk.

Mutex – kölcsönös kizárás

A **mutex** név az angol **mutual exclusion** rövidítése, kölcsönös kizárást jelent. Többfeladatos programozásnál előfordul, hogy két vagy több programszál egyidejűleg ugyanazt az erőforrást akarja használni, ami nem engedhető meg, mert pl. összekeverednének a soros porton vagy az USB porton kiküldött adatok.

Ilyenkor **versengés** alakul ki, s az erőforráshoz hamarabb forduló programszál egy **mutex** változó segítségével lefoglalja az eszközt, majd használat után megszünteti a lefoglalást.



A kritikus szakaszhoz később érkező programszál pedig várakozásra kényszerül, amíg az erőforrás fel nem szabadul. A programszálak tehát kölcsönösen kizárják egymást a kritikus szakaszon, nem fordulhat elő, hogy egyidejűleg vezérlik a közös használatú perifériát.

A Mutex objektumosztály tagfüggvényei

A kölcsönös kizárást biztosító mutexek létrehozása és kezelése a **Mutex** objektumosztály segítségével végezhető, tagfüggvényeit az alábbi táblázatban foglaltuk össze.

Függvéynév	Funkció
Mutex név	Létrehoz és inicializál egy "név" nevű mutex objektumot
lock(time)	Várakozik, amíg a mutex elérhetővé válik. Ha a mutex foglalt, a várakozás alapértelmezetten korlátlan ideig tarthat, vagy a <i>time</i> paraméterben ezredmásodpercekben megadott idő leteltékor időtúllépéssel kilép a várakozásból.
trylock()	Megpróbálja lefoglalni a mutex példányt, de nem várakozik, ha nem sikerül.
unlock()	Felszabadítja a korábban lefoglalt mutex példányt.

Megjegyzések:

- ❖ A **Cortex-M0** mikrovezérlők esetében használt **ARM microlib C programkönyvtár** esetében nincsenek beépített **stdio** mutexek, a megosztott hozzáférés védelméről tehát nekünk kell gondoskodnunk (pl. **printf()** használata esetén).
- ❖ Megszakítás szinten nem hívhatjuk meg a Mutex osztály tagfüggvényeit!

Mintapélda a mutex használatára

Az alábbi példában három programszál verseng a sdtio kimenet (esetünkben UART0) használatáért. A megosztott erőforrás hozzáféréseinek védelmét a **stdio_mutex** látja el.

```
#include "mbed.h"
#include "rtos.h"

Mutex stdio_mutex;

void notify(const char* name, int state) {
    stdio_mutex.lock();
    printf("%s: %d\n\r", name, state);
    stdio_mutex.unlock();
}

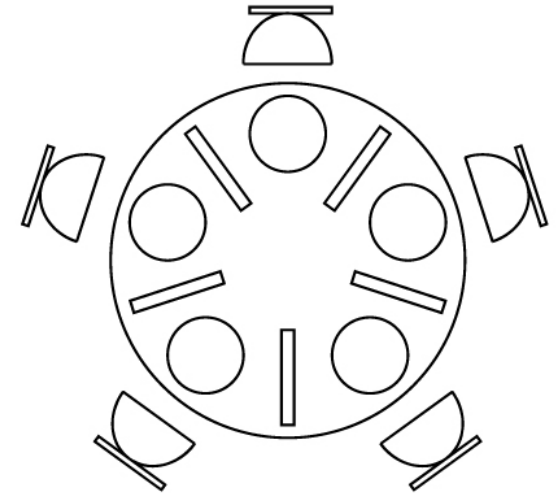
void test_thread(void const *args) {
    while (true) {
        notify((const char*)args, 0); Thread::wait(1000);
        notify((const char*)args, 1); Thread::wait(1000);
    }
}

int main() {
    Thread t2(test_thread, (void *) "Th 2");
    Thread t3(test_thread, (void *) "Th 3");
    test_thread((void *) "Th 1");
}
```

Az étkező filozófusok problémája

Az étkező filozófusok esete (az angol szakirodalomban: **Dining Philosopher Problem** néven ismert) egy olyan probléma, ami könnyen és gyorsan megérthető, de felmerülő problémái valós informatikai problémákká nőnek ki magukat.

Egy tibeti kolostorban öt filozófus él. Minden idejüket egy asztal körül töltik. Mindegyikük előtt egy tányér, amelyből sohasem fogy ki a rizs. A tányér mellett jobb és bal oldalon is egy-egy pálcika található, az ábra szerinti elrendezésben. A filozófusok életüket az asztal melletti gondolkodással töltik. Amikor megéheznek, étkeznek, majd ismét gondolkodóba esnek a következő megéhezésig. És ez így megy az idők végezetéig. Az étkezéshez egy filozófusnak meg kell szereznie a tányérja mellett mindkét pálcikát. Ennek következtében amíg eszik, szomszédjai nem ehetnek. Amikor befejezte az étkezést, leteszi a pálcikákat, amelyeket így szomszédjai használhatnak.



Elemezzük, milyen problémákkal szembesülhetnek a filozófusaink:

Holtpont - előfordulhat-e olyan eset, amikor valamelyik filozófus nem eszik, és nem is gondolkodik?

Kiéheztetés - előfordulhat-e olyan eset, hogy valamelyik filozófus éhen hal?

A probléma modellezése

Filozófusok: egy-egy **programszál**, amely véletlen hosszúságú időtartamig várakozik (gondolkodik), majd megpróbálja megszerezni a két pálcikát (megosztott használatú erőforrások) és véletlen időtartamig várakozik (étkezés), ezután leteszi a pálcikákat (elengedi az erőforrásokat) és kezdődik újra a gondolkodás.

Pálcikák: A pálcikák megosztott erőforrások, ezért ezeket egy-egy **mutex** képviseli.

A holtpontról elkerülése a programozó felelőssége. Ha egyszerre minden filozófus megragadja a jobb kezénél levő pálcikát, akkor máris holtpontra jutottunk!

Stratégia: Ha valamelyik filozófus nem tudja megszerezni mindkét pálcikát, akkor leteszi a kezéből az elsőnek felvettét is, hogy a szomszédjai étkezni tudjanak, majd véletlenszerűen választott rövid ideig tartó várakozás után újra próbálkozik.

Algoritmus: Az i . programszál futásakor a **trylock()** metódussal ellenőrizzük, hogy sikerült-e a jobboldali pálcikához tartozó $i-1$. mutexet lefoglalni. Ha sikerült, akkor megpróbáljuk lefoglalni a másikat (az $i\%5$. mutexet) is. Ha ez is sikerült, akkor kezdődhet a falatozás, majd letesszük a pálcikákat és kezdődhet a gondolkodás. Ha a második mutexet nem sikerült lezárni, akkor feloldjuk az első mutexet, s véletlen időtartam elteltével próbálkozunk újra. Azért kell $i\%5$ az i helyett, mert az 5. filozófus bal kezénél az első (0. sorszámú) pálcika van!

10_rtos_dpp_mutex 1/2

```
#include "mbed.h"
#include "rtos.h"
Mutex stdio_mutex;
Mutex chopstick[5]; //Array of mutexes representing the 5 chopsticks

void notify(int num, int state) {
    stdio_mutex.lock();
    if(state) {
        printf("Philosopher %d is EATING \n\r", num);
    }
    else {
        printf("Philosopher %d is thinking \n\r", num);
    }
    stdio_mutex.unlock();
}
```

- ❑ Az öt mutexet tömbként definiáljuk, hogy indexelhetők legyenek.
- ❑ A kiíratás megosztott használata miatt egy további mutexre is szükségünk van a hozzáférés védelmére.

10_rtos_dpp_mutex 2/2

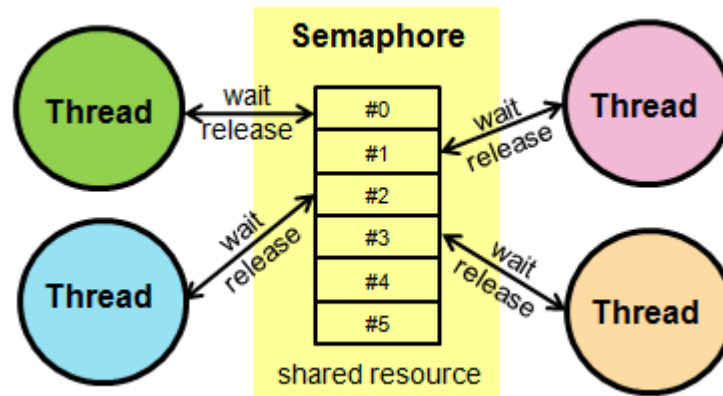
```
void philosopher(void const *args) {
    while (true) {
        if(chopstick[(int)args-1].trylock()) {
            if(chopstick[(int)args%5].trylock()) {
                notify((int)args,1);           //Start EATING
                Thread::wait(1000+rand()%1000);
                chopstick[(int)args%5].unlock(); //Release chopsticks
                chopstick[(int)args-1].unlock();
                notify((int)args,0);           //Start Thinking
                Thread::wait(2000+rand()%2000); //Get's hungry after this time...
            }
            else {
                chopstick[(int)args-1].unlock();
                Thread::wait(100+rand()%100); //Wait for random time if failed
            }
        }
        else {
            Thread::wait(100+rand()%100); //Wait for random time if failed
        }
    }
}

int main()
{
    Thread t2(philosopher, (void *)2U);
    Thread t3(philosopher, (void *)3U);
    Thread t4(philosopher, (void *)4U);
    Thread t5(philosopher, (void *)5U);
    philosopher((void *)1U);
}
```

(int)args a futó programszál sorszáma
(int)args-1 a jobboldali pálcika indexe
(int)args%5 a baloldali pálcika indexe

A szemafor

A **szemafor** olyan absztrakt adattípus, amit az osztott erőforrások egy készletéhez való hozzáférés szabályozásához, illetve programszálak szinkronizálásához használnak a többszálú környezetekben. Megalkotása Edsger Dijkstra holland matematikusnak, a programozás egyik úttörőjének nevéhez fűződik.



[Michaell Barr hasznos írása \(Mutexes and Semaphores Demystified\)](#) megemlíti, hogy a **mutexek** és **szemaforok** fogalmának összekeverése történelmi eredetű, s egyszerű analógiák segítségével tisztázza, hogy mi a különbség köztük.

A szemafor önmagában nem oldja meg a több azonos erőforrás megosztásának problémáját, ehhez további, kiegészítő információra, illetve eszközre is szükség van!

A szemafor „producer – consumer” viszonylatban is használható szinkronizálásra..

A Semaphore objektumosztály tagfüggvényei

Az osztott erőforrások egy készletéhez való hozzáférést szabályozó semaforok létrehozása és kezelése a **Semaphore** objektumosztály segítségével végezhető, tagfüggvényeit az alábbi táblázatban foglaltuk össze. Az **mbed-rtos**-ban implementált **Semaphore** objektumosztály ún. számláló semaforokat kezel, amelyek értékének nincs felső határa (csak a számábrázolásból eredő **int32_t** típushoz tartozó fizikai korlát).

Függvénynév	Funkció
Semaphore név(szám)	Létrehoz egy "név" nevű semafor objektumot és inicializálja a megadott számmal (a szám a kezdetben rendelkezésre álló erőforrások száma).
wait(time)	Várakozik, amíg a semaforral kezelt erőforrások valamelyike elérhetővé válik, majd lefoglalja (eggyel csökkenti a semafor értékét). Ha a semafor nulla (nincs elérhető erőforrás), a várakozás alapértelmezetten korlátlan ideig tarthat, vagy a <i>time</i> paraméterben ezredmásodpercekben megadott idő leteltekor időtúllépéssel kilép a várakozásból.
release()	Felszabadítja (eggyel növeli a semafor értékét) a korábban wait() metódussal lefoglalt semafort.

Étkező filozófusok esete – némi könnyítéssel

Bevezetünk egy könnyítést: a filozófusok nem csak a tányérjuk melletti pálcikákat vehetik fel, hanem **bármelyik két szabad pálcikát!** Mivel így nem kell nyilván tartanunk, hogy melyik erőforrás szabad, mutexek helyett elég lesz egy szemafor is, amelyben nyilvántartjuk, hogy hány erőforrás van még szabadon.

10_rtos_dpp_easy 1/2

```
#include "mbed.h"
#include "rtos.h"

Semaphore s(5); //a pool of 5 chopsticks
Mutex stdio_mutex; //Mutex required for Cortex-M0
Timer mytime;

void notify(const char* name) {
    stdio_mutex.lock();
    printf("%s acquired two chopsticks %8.1f\n\r", name, mytime.read());
    stdio_mutex.unlock();
}
```

10_rtos_dpp_easy 2/2

```
void test_thread(void const* args) {
    while (true) {
        Thread::wait(1000+rand()%500); //Thinking
        s.wait();
        s.wait();
        notify((const char*)args);
        Thread::wait(500+rand()%500); //Eating
        s.release();
        s.release();
    }
}

int main (void) {
    mytime.start();
    Thread t2(test_thread, (void *)"Philosopher 2");
    Thread t3(test_thread, (void *)"Philosopher 3");
    Thread t4(test_thread, (void *)"Philosopher 4");
    Thread t5(test_thread, (void *)"Philosopher 5");
    test_thread((void *)"Philosopher 1");
}
```

Megosztott erőforrások kezelése

- ❖ A következő példában három programszál verseng két erőforrás valamelyikének használatáért, s tudni akarjuk azt is, hogy mikor, melyik programszál melyik erőforráshoz fér hozzá. A szabad erőforrások számát egy **sem** nevű szemafor segítségével tartjuk nyilván, a két erőforrás foglaltságát illetve a hozzáférés kizárólagosságát egy-egy mutex (**m1** és **m2**) használatával adminisztráljuk.

Ha egy programszálnak sikerült hozzáférési jogot szereznie, akkor még ki kell derítenie, hogy a két erőforrás közül melyik szabad. Ehhez a **trylock()** metódust fogjuk használni, amelyik nem várakozik, hanem egy logikai értékkel jelzi, hogy sikeres volt-e a foglalás. Ha az első próbálkozásunk sikertelen volt, akkor a másik erőforrásnak kell szabadnak lennie, ezért ott már bátran használhatjuk a **lock()** metódust.

Mi itt a szemafor szerepe, ha az erőforrások tényleges lefoglalását úgyszólván egy-egy mutex segítségével végezzük? Elsősorban az, hogy a beérkező kérelmeket sorbarendeze. Ennek két haszna van:

- ❖ Nem fordulhat elő, hogy az egyik erőforrásra többen is várakoznak, miközben a másik erőforrás szabadon áll.
- ❖ A szemafor egyetlen várakozási sorából az ütemezőnek lehetősége van arra, hogy mindig a legmagasabb prioritású programszálhoz engedje tovább.

10_rtos_semaphore

```
#include "mbed.h"
#include "rtos.h"

Semaphore sem(2); //manges two tokens
Mutex m1, m2; //two mutexes for two resources
Mutex stdio_mutex; //Control shared access to printf
DigitalOut led1(LED1); //Red LED
DigitalOut led2(LED2); //Green LED
DigitalOut led3(LED3); //Blue LED
DigitalOut ledarray[]= {led1,led2,led3}; //Just needed for indexing the objects...
Timer mytime;

void notify(int tid, int res) {
    stdio_mutex.lock();
    if(res > 0) {
        printf("Task %d: acquired << resource %d. at %8.1f\n\r",tid,res,mytime.read());
    } else {
        printf("Task %d: released >> resource %d. at %8.1f\n\r",tid,-res,mytime.read());
    }
    stdio_mutex.unlock();
}
```

A programszálak várakozásait a megfelelő színű (1: vörös, 2: zöld, 3: kék) LED felkapcsolásával jelezzük.


```

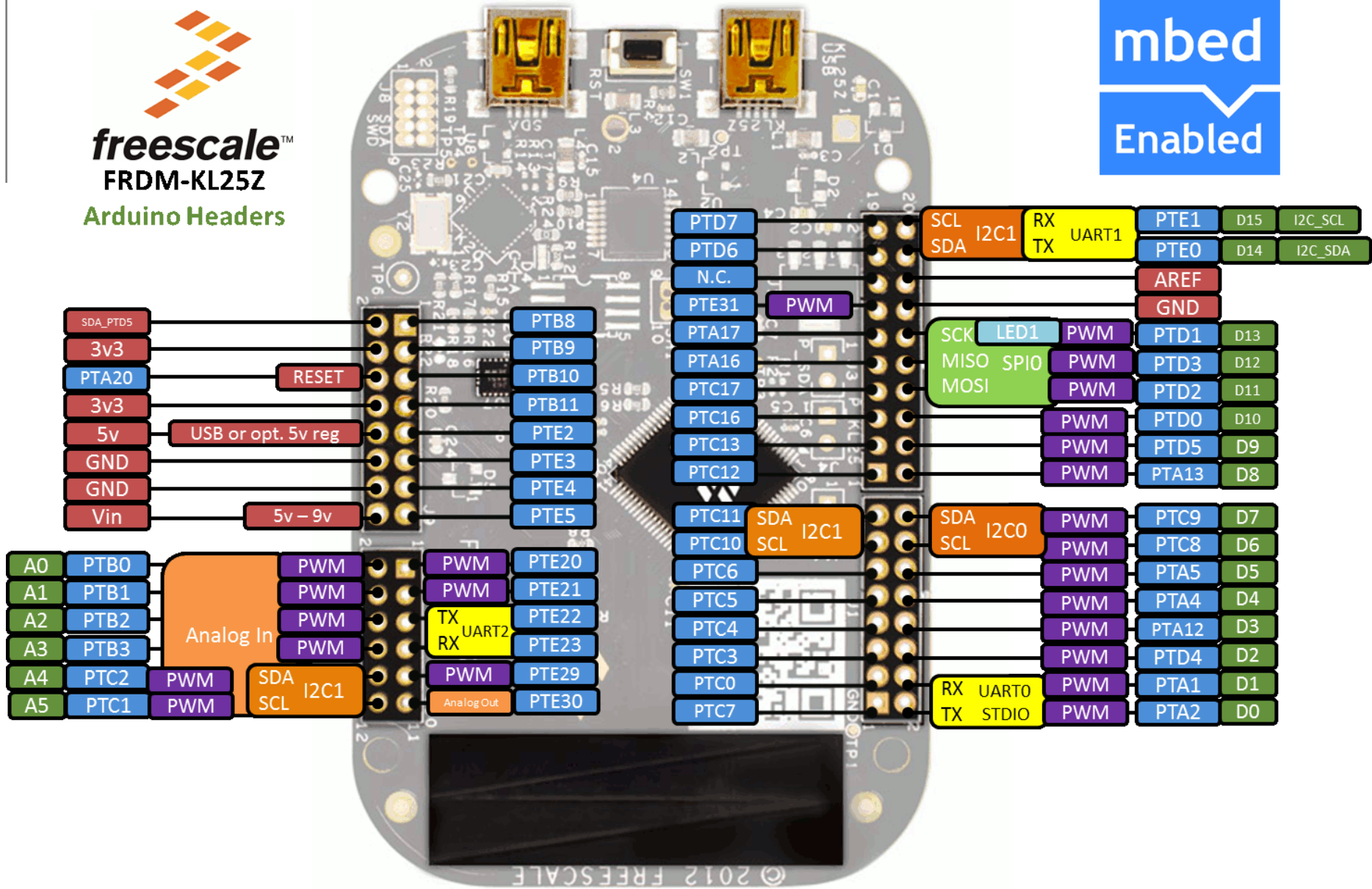
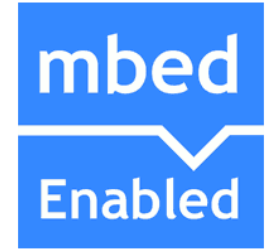
void mythread(void const* args) {
    while (true) {
        Thread::wait(500+rand()%500);
        ledarray[(int)args-1]=0;           //LEDx on
        sem.wait();                        //Wait for token
        if(m1.trylock()) {                 //Try to lock mutex #1
            ledarray[(int)args-1]=1;       //LEDx off
            notify((int)args,1);
            Thread::wait(1000+rand()%500);
            notify((int)args,-1);
            m1.unlock();
        } else {
            m2.lock();                     //Wait for mutex #2
            ledarray[(int)args-1]=1;       //LEDx off
            notify((int)args,2);
            Thread::wait(1000+rand()%500);
            notify((int)args,-2);
            m2.unlock();
        }
        sem.release();                    //Release token
    }
}

int main (void) {
    led1 = 1;
    led2 = 1;
    led3 = 1;                             //Switch off all LEDs
    mytime.start();                       //Start timer
    Thread t2(mythread, (void *)2U);       //Define and run thread2
    Thread t3(mythread, (void *)3U);       //Define and run thread3
    mythread((void const*)1U);            //Run thread1
}

```



freescale™
FRDM-KL25Z
 Arduino Headers





freescale™
FRDM-KL25Z

Additional Peripherals



- PTE24 SCL
- PTE25 SDA
- PTA14 INT1
- PTA15 INT2

freescale
MMA8451Q
Accelerometer

- LED1 PTB18 PWM
- LED2 PTB19 PWM
- LED3 PTD1 PWM

RGB
LED

Capacitive Touch Slider

- PTB16
- PTB17

