# **Deelvraag 1**

**Inleiding**

Om de functionaliteiten van deze rtos-libary wat duidelijker te maken zal ik hieronder de verschillende onderdelen toelichten.

*De deelvraag voor dit hoofdstuk is:* wat zijn de kenmerkende eigenschappen van tasks en de concurrency mechanismen van het Arduino RTOS?

*Mijn hypothese hiervoor is*: In de Arduino RTOS zitten de volgende concurrency mechanismen: pool, channel, flag, clock, timer en mutex. Daarnaast heeft arduino RTOS een mechanisme om de taken te regelen, dit is essentieel want er is maar één processorkern is.

Met behulp van de documentatie van het Arduino RTOS ga ik alle elementen beschrijven, om daarna te kunnen bepalen of mijn hypothese klopt

**Task**

Voordat je met de synchronisatie onderdelen van deze specifieke rtos kan werken zal eerst een taak aangemaakt moeten worden. Een taak kan naast andere taken werken. In deze rtos kun je een taak aanmaken door een klas overerving te geven van de klas rtos::task. Hierbij roep je de taak functie aan met een naam, prioriteit(tot maximaal 1000) en een stackgrootte.

class my\_task\_class : public rtos::task{

public:

my\_task\_class( const char \* name):

task(

name,

10,

16384

){}

private:

void main(void){}

};

De stackgroote staat standaard op 4 kb. In de main kunnen de functionaliteit van de klas worden geschreven.

Je zou nu een taak kunnen maken door de volgende instructie:

my\_task\_class task\_1(“This is the task name”);

Deze taak kan nu via een aantal verschillende methodes een status geven. De taak heeft twee vlaggen eentje die aangeeft of de taak aan het wachten en eentje die aangeeft of de taak op pauze is gezet. Een taak mag gerunt worden op het moment dat deze niet aan het wachten is en ook niet op pauze staat. Met de functie suspend() kun je de taak op pauze zetten. Hierdoor is de taak niet uitvoerbaar totdat de functie resume() is uitgevoerd. Met de functie release() kun je de CPU tijdelijk terug geven om een taak met een hogere prioriteit uitevoeren. Als er geen taken met een hogere prioriteit zijn die uitgevoerd moeten worden gaat de huidige taak gewoon door. Daarnaast kan met sleep() de taak voor een bepaalde periode stil worden gezet. In deze tijd kan de taak niet worden uitgevoerd, zodra de periode van de sleep voorbij is kan de taak weer uitgevoerd worden. Daarnaast kun je met een aantal functies uitvinden of een taak klaar is om te runnen of niet. Tot slot kan een taak met de operatie wait() ook nog wachten op een aantal waitables (bijvoorbeeld een flag)

**Waitables**

Een waitable is iets waar een taak op kan wachten. Alle asynchrone concurrency mechanisme (zoals een flag) zijn waitables. Een waitable is altijd verbonden met een taak en heeft twee states set en clear. De klas waitable is een abstracte klasse en wordt alleen aangeroepen als een waitable mechanisme zoals een flag aangeroepen wordt. Met task::wait() kun je wachten op een specifieke waitable. Daarnaast is er een operator+ waarmee op meerdere waitables gewacht kan worden.

**Flag**

Een flag is een concurrency mechanisme dat een event kan doorgeven aan verschillende taken zonder dat daarbij data wordt doorgegeven. Een flag heeft twee operaties set en wait. De set operatie kan worden aangeroepen worden door een flag::set() of task::set(flag) operaties. Een task kan een flag aanzetten doordat een bepaald event plaats vindt, bijvoorbeeld: een knop wordt ingedrukt. Een andere klas kan dan wachten tot de flag aangezet is met het flag::wait() of task::wait(flag) operaties. Zodra de taak klaar is met wachten op de flag wordt de flag weer uitgezet zodat de taak de flag weer aan kan zetten. Hieronder staat de rtos code voor het maken van een flag

rtos::flag flag\_name;

flag\_name(this, “flag\_name”);

wait(flag\_name);

flag\_name.set();

**Timer/Clock**

Een timer kan met de timer::set() aangezet worden. Na de ingestelde tijd geeft de timer een puls. Een timer is dus best te vergelijken met een flag, de tijd kan aangezet worden en de taak die aan het wachten is dan klaar met wachten. Met de instructie timer::cancel kan je de timer uitzetten. Als je de tijd overschrijft van de timer van bijvoorbeeld 1 ms naar 3 ms, gaat de timer 3 ms wachter. Hij overschrijft dus de oude tijd. Een clock kan ingesteld worden op een bepaalde frequentie en geeft dan een puls op het moment dat de interval periode voorbij is. Hieronder staat de rtos code voor het maken van een timer of clock:

//Timer

rtos::timer name\_of\_timer;

name\_of\_timer(this, “name\_of\_timer”);

name\_of\_timer.set(delay /\*bijv: 200 ms\*/);

wait(name\_of\_timer);

// Clock

rtos::timer name\_of\_clock;

name\_of\_clock(this, delay,“name\_of\_clock”);

wait(name\_of\_timer);

**Pool**

Een pool is een plek om data op te slaan. Een pool heeft echter geen synchronisatie mogelijkheid, dit betekent dat er tegelijkertijd in de pool geschreven en gelezen kan worden. Om een pool aan te maken in de rtos-libary staat hieronder weergeven.

rtos::pool< unsignend int > seconds;

seconds(“seconds”);

seconds.read();

seconds.write();

**Pool with mutex**

Een mutex is een pool met synchronisatie. Deze synchronisatie is in de vorm van een sleutel die doorgegeven kan worden. De mutex kan door een taak gebruikt worden door de operatie mutex::wait(). Zodra de mutex dan vrij is en de taak aan de beurt is kan de taak de data lezen uit de pool. Met de operatie mutex::signal() geeft de taak de pool weer terug en kan een andere taak de pool gebruiken. Hieronder staat de c++ code voor het aanmaken van een mutex:

auto mutex\_name = rtos::mutex(“mutex\_name”)

class class\_you\_want\_to\_use\_mutex{

private:

rtos::mutex & mutex\_name

public:

class\_you\_want\_to\_use\_mutex(rtos::mutex &mutex\_name):

mutex\_name( mutex\_name )

{}

mutex\_name.wait();

mutex\_name.signal();

Deze code kan gecombineerd worden met die van een pool.

rtos::pool<unsignend int > seconds;

seconds(“seconds”);

mutex\_name.wait();

seconds.write();

mutex\_name.signal()l

**Mailbox**

Een mailbox is geen waitable en zit dus niet vast aan een bepaalde taak. De mailbox is initieel leeg en kan worden data doorgeven. Als de schrijver er met een write() operatie iets in de mailbox stopt wordt deze taak geblokkeerd totdat de ontvanger met de read() operatie de data uit de mailbox heeft gehaald. Als er niks in de mailbox staat wordt de ontvanger met een read() operatie geblokkeerd tot dat er iets in de mailbox wordt geschreven. Hieronder is te vinden hoe dit in rtos geïmplementeerd kan worden:

mailbox<usignend int> mailbox\_to\_use;

mailbox\_to\_use.read();

mailbox\_to\_use.write();

Zoals gezien kan worden is dit heel vergelijkbaar met een pool.

**Channel**

Een channel is een queue waarop gewacht kan worden (een waitable dus). Een channel heeft een bufferlengte geven Als je over deze lengte heen schrijft krijg je een error. Als een taak iets in het channel schrijft wordt deze taak niet geblokkeerd. Als de ontvangende taak iets leest wordt die waarde uit het channel gehaald en alle andere waarde opgeschoven. Hieronder is te vinden hoe een channel in rtos aangemaakt en gebruikt kan worden.

rtos::channel<char, 2048 /\*(length of buffer)\*/> buffer;

buffer(this, “buffer”);

auto c = Buffer.get();

buffer.write(‘h’);

**Slot**

De kenmerkende eigenschappen van het Arduino RTOS zijn hierboven , uitgelegd. Hierbij gaat het vooral om de verschillende concurrency mechanisme die zich in het Arduino RTOS bevinden. Dit zijn:

Pool, channel, flag, clock, timer en mutex. Daarnaast heeft deze RTOS ook een mechanisme om met taken te werken. Deze taken kunnen namelijk op verschillende manieren stop gezet worden (met bijv. resume, suspend, wait, sleep). Mijn hypothese klopte dus.

**Beschikbare RTOS-en**

In het geval van open source RTOS-en zijn er vele verschillende mogelijkheden Een RTOS kan gemaakt zijn voor een speciale taak, aparte hardware, specifiek werk afdeling, ondersteuning van specifieke programmeertalen en dergelijke. Kijkende naar RTOS-en met een aantal eigenschappen zoals het ondersteunen van ARM controllers waarop de RTOS op zal draaien, het compatibel zijn met C en C++ programmeertalen waardoor met deze programmeertalen de verdere software kan worden geschreven. Ook moet de RTOS recent zijn geüpdatet (maximaal jaar geleden) en waardevolle documentatie over de functionaliteiten. Zijn er nog steeds vele opties met de specificaties waar we ons op hebben gericht.

**FreeRTOS**

Als eerst hebben we FreeRTOS, gemaakt door Real Time Engineers Ltd. en is vrijgegeven onder de GNU general public license . deze RTOS ondersteunt 35 verschillende microcontrollers en bestaat al 12 jaar waarin de jaren de RTOS meervoudig is aangevuld en verbeterd. FreeRTOS in de loop der jaren uitgegroeid tot 1 van de markteider onder de RTOS-en. FreeRTOS is ontworpen om kleinschalig te zijn en simpel te werken hierdoor heeft FreeRTOS een kleine memory foodprint en heeft een snelle uitvoering. De code van FreeRTOS heeft als voordeel dat het bestaat uit 3 .C bestanden voor de RTOS, hierdoor is het overzichtelijk om de code te bekijken en vooral om te vinden waar welke functie staat. Er zijn ook RTOS-en waarbij de functies in veel verschillende files bevinden het onoverzichtelijk wordt om te bekijken. De overige files in het .zip file van FreeRTOS zijn demonstratie files waarbij. FreeRTOS bevat een goede aansluiting voor verschillende beschikbare API’s . FreeRTOS bevat standaard de nodige synchronisatie middelen zoals: mutex, pool, channel , mailbox, etc. het bevat ook de benodigdheden voor een low power applictation. Behalve de ARM architectuur wordt door de FreeRTOS ook nog andere architecturen ondersteunt waaronder de Texas Insturments MSP430 ,Atmel SAM3 en nog vele meer. De FreeRTOS is dus een volledig rtos waarmee vele toevoegingen mogelijk zijn.

**mbed**

Als tweede hebben we mbed, gecreëerd door ARM speciaal voor het ‘internet of things’ onder een apache license. Het mbed RTOS bevat de noodzakelijke synchronisatie middelen zoals: threads, mutex, semaphore, signals, pools, channel, mailbox, timeouts, system interrupts en een header waarmee de nodigheden voor het verbinden met andere systemen worden geinclude . Doormiddel van de header worden api (application programming interface) aan de rtos toe gevoegd die vele mogelijkheden bieden. zoals voor besturing van input en output onder anderen voor analoge en digitale poorten, pwm signaal, bus, en input voor interrups. Er zijn ook verschillende api beschikbaar in mbed voor dergelijke interfaces waar onder serial, SPI, I2C, CAN en verschillende interfaces gebaseerd op usb, waaronder voor mouse en keyboard ondersteuning. Omdat mbed speciaal is ontworpen voor het “internet of things” is zijn er api beschikbaar waarmee bluetooth, ethernet en WIFI verbinding kan worden gerealiseerd. Door de vele mogelijkheden die beschikbaar zijn bij het gebruik van mbed is het een RTOS die vele mogelijkheden bied op het gebied van verbinding met andere systemen.

**Chibi-OS/rt**

Als derde hebben we Chibi-OS/rt en is ontworpen door Giovanni Di Sirio en is beschikbaar vanaf 2006 met wekelijkse fixes en updates. Deze rtos is een basis rtos en bevat de nodige functies met een mogelijkheid voor API’s. De kernel van chibiOS/ RT bevat een aantal services en modules voor onder andere de base kernel services, hiermee wordt bedoelt de functies voor scheduler, system management, threads en time and virtual timers. synchronisatie middelen zoals de condition variabls, channels, mailboxes, semaphores, event flags en mutexes bevinden zich ook in een van de modules. Memory management is verwerkt in een module me de mogelijkheden voor een Heap, memory pool, core management management en dynamic threads. De Chibi-Os/rt is ontworpen om met meerdere architecture te kunnen werken zoals ARM, Almes mega AVR, NXP, PIC32MX.

**Atom threads**

Als vierde RTOS kwam Atom threads aan de orde dit is een rtos gemaakt door atomthreads. Deze rtos heeft als hoofdtaak echt zo minimaal te zijn als mogelijk waarbij zoals bij FreeRTOS er als nog bepaalde functies aan toegevoegd zijn is er bij atom threads het tot het minimaal is gebracht en het is ook bedoelt voor systemen die alleen een scheduler en RTOS primitives. Het rtos bevat : preemptive scheduler with 255 priority levels, round-ribin at same priority level, semaphore, mutex, channel, timers. Het rtos is gemaakt in 11 .h en .C files waardoor het overzichtelijk blijft en makkelijk te begrijpen is. Doordat de atom threads zo minimaal is, is het mogelijk gemaakt om vele architecturen te gebruiken. En door een specific architecture module te gebruiken. wel is het mogelijk om API’s toe te voegen aan het rtos.

**RIOT**

Als vijfde is RIOT gekozen. Deze rtos heeft een andere benadering dan de andere rtos-en die eerder genoemd zijn. RIOT is gebaseerd op het concept Internet of Things. Hierbij bevat deze RTOS behalve de RTOS primitives ook de benodigde modules voor een verbinding in een netwerk met ondersteunende systemen. Het systeem heeft ondersteuningen voor onder andere IPv6, RPL,UDP, CoAP, TCP en CBOR. Met volledige ondersteuning Voor speciaal low voltage applicaties wordt ook het 6LoWPAN ondersteund. Qua architecturen worden 8-bit(zoals, AVR Atmega ) , 16-bit (zoals, TI MSP430), 32-bit(zoals, ARM Cortex) ondersteund. Het is ook mogelijk om riot als een proces te runnen onder linux en os x. RIOT bevat voor verschillende modules de nodige drivers, zoals verschillende omgeving sensoren maar ook ondersteuning voor verschillende servo motors.

**Vergelijkend met Arduino RTOS**

Nu uitgelegd is wat de functionaliteiten zijn van de Arduino RTOS en er vijf open source RTOS-en besproken zijn gaan we nu kijken wat de overeenkomsten zijn tussen de gekozen RTOS-en en de Arduino RTOS met instelling op de handeling van tasks en de beschikbare concurrency mechanisme.

Voor het vergelijken van de vijf open source RTOS-en met de Arduino RTOS gaan we deze één voor één langs beginnend met FreeRTOS

/\* Task to be created. \*/

void vTaskCode( void \* pvParameters )

{

/\* here comes code\*/

}

/\* Function that creates a task. \*/

void vOtherFunction( void )

{

BaseType\_t xReturned;

TaskHandle\_t xHandle = NULL;

/\* Create the task, storing the handle. \*/

xReturned = xTaskCreate(

vTaskCode, /\* Function that implements the task. \*/

"NAME", /\* Text name for the task. \*/

STACK\_SIZE, /\* Stack size in words, not bytes. \*/

( void \* ) 1, /\* Parameter passed into the task. \*/

tskIDLE\_PRIORITY,/\* Priority at which the task is created.\*/

&xHandle ); /\* Used to pass out the created task's handle. \*/

if( xReturned == pdPASS )

{

/\* The task was created. Use the task's handle to delete the task. \*/

[vTaskDelete](http://www.freertos.org/a00126.html)( xHandle );

}

}

FreeRTOS heeft voor het aanmaken van task een andere benadering genomen dan de Arduino RTOS, namelijk waar bij de Arduino RTOS het gebaseerd is op objecten dus dat een klas een overerving krijgt van rtos::task. Bij FreeRTOS wordt een ander systeem gebruikt hierbij wordt een functie(in voorbeeld **void vTaskCode**) aangemaakt die het embedded system moet runnen en door het gebruik van een extra functie(**in voorbeeld vOtherFunction**) de task aangemaakt die de functie bevat.

Voor de aanmaak van een task moet bij FreeRTOS een aantal parameter worden meegegeven: de desbetreffende functie, de task naam, stack\_size , de parameters die die de functie nodig hebben, de prioriteit van de task en de xHandle voor het kunnen handelen met de task. In tegenstelling tot Arduino rtos waar bij aanmaak voor een task in de klas drie parameters moeten worden gegeven: naam van de task, prioriteit en de stack grote. Voor het handelen met de tasks heeft de FreeRTOS een aantal functies. VtaskDelay() wordt gebruikt voor het voor een bepaalde periode de task stil te zetten dit is vergelijkbar met de sleep() functie in de Arduino RTOS. De functies vTasksuspend() en vTaskResume() zijn de vergelijkbar functies in de Arduino RTOS suspend() en Resume(). De vTasksuspend() kan de task op een pauze worden gezet en door gebruik van vTaskResume() kan de task weer worden uitgevoerd. De functies vTaskPriorityGet() geeft de priority van een task weer en de functie vTaskPrioritySet() kan de prioriteit van een task veranderen. In de Arduino RTOS bevindt zich een equivalent voor vTaskPriorityGet() genaamt priority() deze geeft ook als unsigned int return de prioriteit van een task.

Voor de concurrency mechanisme bevat de Arduino RTOS de volgende mechanism: flag - timer/clock – pool – mailbox – channel - mutex.

De FreeRTOS heeft in tegenstelling tot ArduinoRTOS niet een aparte functie voor het gebruik van een flag wel heeft het een verzameling van meerdere flags door middel van xEventGroup(). Deze functie kan gebruikt worden als flag wanneer er wordt gewacht op 1 flag uit de groep maar het is dus ook mogelijk om te wachten tot dat meerdere bits in de group op TRUE staan. FreeRTOS bevat de nodige functies om een clock en timer te maken doormiddel van XTimer(). Deze functie bevat de zelfde opties als de Arduino RTOS timer bevat. FreeRTOS bevat de functie Xqueuecreate() hiermee kan een channel gemaakt worden. Het grote verschil in het creëren van een channel, in Arduino rtos wordt gebruikt gemaakt van template waardoor verschillende types de zelfde class kunnen worden gebruikt, waar FreeRTOS juist gebruik maakt van een een struct dat zelf aangemaakt moet worden waarin een array wordt aangemaakt waar de gegevens in komen te staan. Voor het lezen en schrijven van het channel wordt in de Arduino RTOS drie bepaalde functies gebruikt read() het lezen van een item van het channel, write() het schrijven van een item op het channel en clear() het leeghalen van het channel. In FreeRTOS is het mogelijk om het channel te lezen en om leeg te maken. Maar voor het schrijven bevat FreeRTOS ook nog de mogelijkheid om van de andere kant te schrijven of van een bepaalde positie te overschrijven.

Voor mutex bevat FreeRTOS een soort gelijk functie xSemaphoreCreateMutex(). De functie SemaphoreTake() is de equivalent van de wait() van Arduino RTOS en de functie SemaphoreGive() is de equivalent voor signal().

FreeRTOS bevat zover bekent geen functie voor het creëren van een mailbox en het creëren van een pool.

FreeRTOS heeft voor de concurrency mechanisme die beschikbaar zijn meerdere opties dan de Arduino RTOS, alhoewel de Arduino RTOS wel beschikt over mailbox en pool en die niet bevinden in FreeRTOS.

De tweede open source RTOS die we hebben besproken is mbed. Mbed is een op klassen gebaseerd RTOS zoals het Arduino RTOS.

#include "mbed.h"

#include "rtos.h"

DigitalOut led1(LED1);

DigitalOut led2(LED2);

 void led2\_thread(void const \*args) {

    while (true) {

        led2 = !led2;

        Thread::wait(1000);

    }

}

 int main() {

    Thread thread(led2\_thread);

    while (true) {

        led1 = !led1;

        Thread::wait(500);

    }

}

in mbed word een task gemaakt door middel van de volgende code(zie tekstvak ) een verwarrend iets is dat in mbed ze de benaming thread gebruiken en in de Arduino-RTOS wordt gebruik gemaakt van tasks (voor begijpelijk houden gebruiken we de benaming task) maar in deze situatie doet elk precies het zelfde. Voor het aanmaken van een task moet een functie worden aangemaakt en meegegeven worden in de constructor voor een thread object. In de constructor kan onder andere de stack grootte en de prioriteit worden meegegeven zoals bij Arduino RTOS ook het geval is. Mbed bevat voor een task de functie wait(int milliseconde) die wacht ( in de parameter ingevulde ) hoeveelheid milliseconde. Voor de functies suspend() en resume() uit de Arduino RTOS zijn er geen equivalenten in mbed.

In het geval van de concurrency mechanisme mbed de volgende mechanisme.

Mbed beschikt over flags alleen wordt het dan benoemd als signals.

Signal bevat net als de Arduino RTOS voor een flag over een set() functie en een wait() functie. Met de zelfde benaming. Mbed beschikt over het gebruik van een mutex dat op een zelfde manier word aangemaakt in Arduino RTOS. De Arduino RTOS beschikt bij mutex over de functies wait() en signal() in mbed is de equivalent van wait() de functie lock() en voor signal() de functie unlock(). Mbed beschikt over een soortgelijk class voor een timer genaamt Rtostimer. Deze Rtos timer beschikt over twee functies stop() en start() deze functies stoppen en starten de timer, er is geen functie in mbed Rtostimer die het mogelijk maakt om de timer te gebruiken als een clock. voor gebruik van een channel bevat mbed een klas waarin twee functies bevinden put() en get(), put() laat data naar het channel schrijven en get() laat data van een channel afhalen. Put() is de equivalent van write() en get() is de equivalent van read() mbed beschikt alleen niet over een functie waarmee een channel kan worden gecleard. Voor pool bevat mbed een een klas genaamd memory pool. In de pool klas in Arduino RTOS bevinden zich twee functies write() en read() de memorypool klas in mbed bevat twee functies die equivalent zijn aan de write en read functies dat zijn alloc() en free(). Bij de memorypool klas wordt even als in de pool klas van de Arduino RTOS gebruik gemaakt van een template waarmee de datatype bepaald kan worden en de hoeveelheid plaatsen in de pool. Voor mailbox bevind er een mail klas deze maakt een mailbox aan. De mail klas bevat twee functies put() en get () de equivalenten voor write() en read().

Mbed is op de meest punten een volledig RTOS maar in vergelijking met de Aduino RTOS. bij de task is er geen equivalent voor suspend() en resume() en het beschikt niet over een clock functie.

De derde RTOS die eerder is besproken is Chibi-OS/rt.

Voor het aanmaken van een task (in deze RTOS weer als thread ) wordt gebruik gemaakt van de chThdCreateI() functie, aan de functie moet worden meegegeven pointer naar de toegewezen thread stack, de grote van de stack, de prioriteit van de thread, de functie die moet draaien in de thread, en in dien nodig een argument. Zoals in de Arduino RTOS bevind in Chibi de functies om een thread voor een bepaalde tijd op stil te zetten. Voor de suspend() functie in Arduino RTOS is er een equivalent in Chibi chThdSuspendS() deze functie ze de desbetreffende thread op pauze en die kan weer worden gestart me de equivalent van de resume() functie genaamd cdThdResume(). Chibi bevat onder anderen ook een functie waarmee de prioriteit kan worden opgevraagd chThdGetPriority(void).

Voor timer bevat Chibi een klas virtual\_timer\_t doormiddel van de functie chVTObjectInit () is het mogelijk om een timer aan te maken. Voor het starten van de gemaakte timer wordt de functie chVTSet() gebruikt dit is de equivalent van start() functie in de Arduino RTOS. Voor het stoppen van de timer is de functie chVTReset() beschikbaar dit is de equivalent van de cancel(). In chibi bevind zich niet standaard een clock functie. Voor de flag bevind in Chibi een event klas. Een e. in de Arduino RTOS bevind zich twee tal functies wait() en set(), wait() laat wachten tot de flag is uitgezet en doormiddel van set() wordt en flag aangezet. De equivalenten functies voor wait en set zijn chEvtWaitOne() en chEvtSignal(). Voor pool bevat chibi een memorypool klas. Voor het aanmaken van een pool wordt gebruik gemaakt van een functie chPoolObjectinit() met als parameters een pointer naar een memory\_poot\_t (struct die door chibi wordt gebruikt als de pool) en de grootte .

In de Arduino RTOS bevinden zich de functies write() en read(). De equivalenten van de eerder genoemde functies in Chibi is chPoolAlloc() en chPoolFree(). Voor een mailbox heeft Chibi een mailbox klas. Voor het aanmaken van een mailbox is er een functie genaamd chMBObjectinit() met als parameters een pointer naar een object van mailbox, een pointer naar een array dat als message buffer werkt en een integer dat aangeeft hoeveel elementen er aanwezig zijn in de array. In de Arduino RTOS bevinden zich twee functies write() en read(), in chibi zijn er equivalenten voor deze functies. Voor het schrijven naar de mailbox(write() ) is er de functie chMBPost() en voor het lezen van de mailbox(read() ) is er de functie chMBFetch(). Voor het gebruiken van de channel bevat chibi een io\_queue klas. Een channel wordt aan gemaakt door middel van de functie chlQObjectinit() met als parameters naar een object dat werkt als het channel en de grootte van het buffer. Voor het schrijven naar het channel heeft chibi de functie chlQPutl() en voor het lezen van het channel heeft chibi de functie chOQGet() dit zijn equivalenten voor de read en write functie die in de Arduino RTOS bevinden. Chibi bevat ook nog een soort gelijk functie met clear() genaamd chOQReset(). In chibi bevind zich ook een klas waarmee mutexen kunnen worden gecreëerd. Voor het aanmaken van een mutex wordt de functie chMtxObjectinit() gebruikt met als parameter een pointer naar een object van ch\_mutex. Chibi bevat twee functies voor mutexen chMtxLock() en chMtxUnlock() dit zijn de equivalenten van wait() en signal() in de Arduino RTOS.

Kijkend naar de documentatie en de code van Chibi-os/rt bevat deze rtos alle soort gelijken funcies als de Arduino RTOS en dit zou dus een eventuele vervanging kunnen zijn voor de Arduino RTOS.

Als vierde RTOS hebben we atom threads.

In atom threads worden tasks(weer benoemd als thread) gemaakt door middel van de atom ThreadCreate() functie. Zoals in de Arduino RTOS er functies aanwezig zijn om een thread(task) te kunnen pauzeren stoppen en weer opnieuw te starten, zijn deze functies in atom threads niet aanwezig. Atom threads bevat een functie waarmee mutexen kunnen worden aangemaakt dit wordt gedaan met de functie atomMutexCreate() met als parameter een pointer naar een stuc van mutex\_timer. Verder bevat atom threads voor mutexen een get functie een put functie.

De functie atomMutexPut is de equivalent van de signal functie in de Arduino RTOS en de functie atomMutexPut() is de equivalent van de wait() functie. ook voor het maken van een channel heet atom threads een functie genaamd atomQueueCreate(). En atom threads heeft een drie tal functies die werken met een channel. atomQueueDelete() verwijdert het hele channel, de functie atomQueueGet() is de equivalent van de Arduino RTOS functie read() en de functie atomQueuePut() is de equivalent van de Arduino RTOS functie write(). De timer in atom threads bevat een functie genaamd atomTimerDelay() en deze functie is de equivalent van de Arduino RTOS functie sleep().

Atom threads is een zeer minimalistische RTOS en bij elke onderdeel dat de Arduino RTOS heeft mist atom threads wel een functie of is de hele functionaliteit helemaal niet aanwezig. Hier uit worden geconcludeerd dat atom threads net geschikt zou zijn om de lasergame code op te runnen.

Als vijfde RTOS hebben we nog ROIT

Voor het aanmaken van een task(opnieuw genoemd als thread) word aan aangemaakt door middel van de functie thread\_create() functie met als parameters de stackgrootte , de uitvoerende functie , prioriteit, aantal flags en een naam. Voor de functies suspend() en resume() zijn er equivalenten in RIOT. Voor het op pauze zetten van een thread wordt de functie thread\_sleep() gebruikt en voor het wakker maken van een thread wordt de functie thread\_wakeup() gebruikt.

RIOT beschikt over een functie mbox\_init() die een mailbox aan maakt. Als equivalent van de read() en write() functies uit de Arduino RTOS zijn er mbox\_put() voor het versturen naar de mailbox en mbox\_get voor het ophalen van een bericht uit de mailbox. Voor het aanmaken van een mutex bevat RIOT de functie mutex\_init(). Voor de equivalenten van de functies signal() en wait() van uit de Arduino RTOS zijn er de functies mutex\_unlock() en mutex\_lock(). Een timer kan worden aangemaakt door middel van de timer\_init() functie. voor het starten van de timer is er de timer\_set() functie en voor het uitlezen van de timer worth gebruik gemaakt van de timer\_read() functie. de timer.set() functie kan gebruikt worden als clock wanneer de timeout voorbij is en de functie timer\_read() niet is aangeroepen. Een flag wordt gemaakt doormiddel van thread\_flags\_t. voor het aanzetten van de flag wordt de functie thread\_flags\_set gebruikt() en voor het wachten op de flag wordt de functie thread\_flag\_wait\_one() gebruikt. Deze functies zijn de equivalenten van de set() en wait() functie van uit de Arduino RTOS. ROIT beschikt verder niet over functionaliteiten als een pool of een channel

Alhoewel ROIT een zeer compleet systeem is mist het de mogelijkheden voor een channel en de mogelijkheden voor een pool die nodig zijn om de applicatie goed op te kunnen draaien.

Van de vijf besproken RTOS-en is er één waarvan alle benodigheden zijn die de Arduino RTOS ook bevat. voor FreeRTOS - mbed – atom threads – RIOT missen er in de concurrency mechanisme bij elk 1 of meerdere en zijn niet volledig. Het zou mogelijk kunnen zijn om een API te schrijven die de benodigde concurrency mechanisme bevat zodat ze compleet zouden zijn in vergelijking met de Arduino RTOS.