# Inleiding

Het ‘internet of things’ is een echte rage geworden. Tegenwoordig is alles verbonden met het internet, de koelkast, een wasmachine en zelfs verlichting. Omdat het internet altijd aan staat en je als apparaat eigenlijk altijd online moet zijn is een embedded systeem vaak een goede optie. Zo’n embedded systeem, voor een koelkast bijvoorbeeld, moet naast de connectie tot het internet ook nog andere dingen kunnen doen.

Voor onze opdracht moeten we een vergelijkbaar systeem maken maar dan voor lasergamen. Dit lasergame spel wordt gespeeld met behulp van een arduino due. Deze due beschikt over een ARM-processor met 1 kern. Dit brengt een problemen met zich mee hoe kunnen twee processen bijvoorbeeld tegelijk runnen op maar één processorkern.

Om dit soort problemen te kunnen oplossen gaan wij het arduino RTOS gebruiken. Hierover kunt u later meer vinden in ons onderzoeksrapport. Het arduino RTOS is echter niet de enige soort RTOS die hiervoor gebruikt kan worden. In dit onderzoeksrapport gaan wij de geschiktheid van de verschillende soorten RTOS-systemen bekijken. Hiermee willen we bereiken dat we naast het arduino RTOS ook nog een andere RTOS hebben die we voor het project zouden kunnen gebruiken.

De opbouw van dit rapport ziet er als volgt uit: Na de inleiding wordt de probleemstelling en de daarop gebaseerde deelvragen besproken. Daarnaast gaan we het hebben over welke methodes in het onderzoek gebruikt zijn. Hierna gaan we eerst goed kijken naar de arduino RTOS zelf. Ook gaan we het hebben over de verschillende soorten RTOS-systemen die er zijn en wat deze voor overeenkomsten en verschillen hebben. Daarnaast hebben we het over hoe we die verschillen eventueel kunnen overbruggen met oplossingen die binnen de RTOS mogelijk zijn. Tot slot geven we de conclusie van het onderzoeksrapport en een aanbeveling.

# Probleemstelling en onderzoeksvragen

Het arduino RTOS heeft een aantal functionaliteiten die wij in ons concurrency diagram gebruikt hebben. Deze functionaliteiten zijn: een channel, een clock, een pool, een flag, een mailbox en een mutex. Voor dit onderzoek gaan we vooral gebruik maken van de arduino RTOS-documentatie, daarnaast zullen in onze bronvermelding alle webpagina’s staat die gebruikt worden om de voor de andere RTOS-systemen. De functionaliteiten van de arduino RTOS moeten deels terugkomen in de andere RTOS. Aan de hand van deze informatie hebben we de volgende hoofdvraag opgesteld:

*Met het behulp van welk open source realtime operating system kunnen tasks en de concurrency mechanismen, pool, channel, flag(group), clock timer en mutex, zoals aangeboden door het Arduino RTOS, met zo weinig mogelijk overhead worden gerealiseerd*

Hierna hebben we de volgende deelvragen opgesteld:

* *Wat zijn de kenmerkende eigenschappen van tasks en de concurrency mechanismen van het Arduino RTOS?*

Hierbij hebben gaan we vooral kijken naar wat het arduino RTOS voor eigenschappen heeft en hoe deze worden gebruikt.

* *Welk open source RTOS-en zijn beschikbaar?*

Aan deze deelvraag hebben we nog wat extra eisen verbonden om de RTOS-systemen geschikter te maken. Deze staan hieronder weergeven:

* *Het RTOS moet beschikbaar zijn voor het ARM-platform*
* *Het RTOS moet C en C++ ondersteunen*
* *Het RTOS moet binnen het afgelopen jaar een update hebben gekregen*
* *Het RTOS moet beschikken over documentatie waarin de geboden functionaliteit is beschreven*
* *Welk van de beschikbare RTOS-en biedt de meeste van de concurrency mechanismen van het Arduino RTOS aan zonder enige modificatie en biedt dezelfde functionaliteit om taken te realiseren?*

Bij deze deelvraag gaan we de informatie van de vorige deelvragen gebruiken en de verschillende RTOS-systemen met elkaar vergelijken

* *Welke mechanismen van het Arduino RTOS worden niet ondersteund door de beschikbare RTOS-en?*

Bij deze deelvraag zijn we vooral bezig met het vinden van verschillen in de RTOS-systemen

* *Hoe kunnen de mechanismen van het Arduino RTOS die niet direct worden ondersteund door de beschikbare RTOS-en worden gerealiseerd m.b.v. van deze RTOS-en?*

Bij deze deelvraag gaan we opzoek naar oplossingen voor de gevonden verschillen.

# Onderzoeksmethoden

Wij gaan ons onderzoekrapport uitvoeren door veel research te doen. We gaan dus op zoek naar informatie, en leggen geen interviews of enquêtes af. We gaan dus vooral literatuurstudie die en vergelijken bestaande oplossingen

Onze onderzoeksmethode voor literatuurstudie zal vooral bestaan uit het doorlezen van doxygen documentatie voor het arduino RTOS en de andere RTOS-systemen. Op het internet kunnen heel erg veel oplossingen gevonden worden voor de problemen die we ongetwijfeld tegen zullen komen. Hierbij passen wij onze tweede onderzoeksmethode toe, vergelijken bestaande oplossingen. Deze methode wordt ook vooral toegepast op de laatste deelvraag. Hierbij gaan we kijken naar de oplossingen voor ontbrekende delen in de andere RTOS-systemen

# Wat zijn de kenmerkende eigenschappen van tasks en de concurrency mechanismen van het Arduino RTOS?

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?

Met behulp van de documentatie van het Arduino RTOS ga ik alle elementen beschrijven, om daarna te kunnen bepalen wat de kenmerkende eigenschappen zijn van tasks en concurrency mechanisme in het Arduino RTOS

**Task**

Voordat de synchronisatie mechanisme, van deze specifieke rtos, besproken kunnen worden zullen we eerst moeten kijken naar de werking van taken. 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 hiervoor twee vlaggen één die aangeeft of de taak aan het wachten is en eentje die aangeeft of de taak op pauze is gezet. Een taak mag uitgevoerd 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 teruggeven om een taak met een hogere prioriteit uit te voeren. 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 (de taak is namelijk aan het wachten). Daarnaast kun je met een aantal functies uitvinden of een taak klaar is om te runnen of niet, je kunt met deze functies dus eigenlijk de vlaggen checken. Tot slot kan een taak met de operatie wait ook nog wachten op een aantal waitables of een event (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 taak kan een flag aanzetten wanneer 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, als de flag dus aangezet is, 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 krijgt een puls zodra de tijd voorbij. 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. De timer is niet actief totdat hij weer opnieuw wordt ingesteld.

Een clock kan ingesteld worden op een bepaalde frequentie en geeft dan een puls op het moment dat de interval periode voorbij is, een clock is altijd actief, hij hoeft dus niet meerdere keren ingesteld worden. 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. Een pool is dus een geheugenplaats waar taken in kunnen schrijven en lezen. Hieronder staat de RTOS-code voor het maken van een pool

rtos::pool< unsignend int > seconds;

seconds(“seconds”);

seconds.read();

seconds.write();

**Pool met mutex**

Een mutex is een synchronisatie oplossing voor een pool. 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. Een mailbox is initieel leeg en kan 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. 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’);

# 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 een ARM-controller waarop de RTOS volgens onze eisen moet 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 moet aanwezig zijn. Zijn er nog steeds vele opties met de specificaties waar we ons op hebben gericht. Hieronder staan een aantal RTOS-en die voldoen aan deze eisen.

**FreeRTOS**

Als eerst hebben we FreeRTOS, gemaakt door Real Time Engineers Ltd. FreeRTOS is vrijgegeven onder de GNU general public license. FreeRTOS ondersteunt 35 verschillende microcontrollers en bestaat al 12 jaar. In deze jaren is FreeRTOS meervoudig 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 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 application. 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 toegevoegd 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 waaronder serial, SPI, I2C, CAN en verschillende interfaces gebaseerd op usb waaronder muis en toetsenbord 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 dat vele mogelijkheden biedt 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 bedoeld de functies voor scheduler, system management, threads,time and virtual timers. Synchronisatie middelen zoals de condition variables, channels, mailboxes, semaphores, event flags en mutexes bevinden zich ook in een van de modules. Memory management is verwerkt in een module met de mogelijkheden voor een Heap, memory pool, core 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 komt Atom threads aan de orde, dit is een RTOS gemaakt door Atom threads. Deze RTOS heeft als hoofdtaak echt zo minimaal te zijn. Waarbij net als FreeRTOS er alsnog bepaalde functies aan toegevoegd zijn. Bij atom threads is het tot het minimale gebracht, daarnaast is Atom threads bedoelt voor systemen die alleen een scheduler en RTOS primitives hebben. Het RTOS bevat: preemptive scheduler met 255 priority levels, round-ribin met dezelfde priority level, semaphore, mutex, channel, timers. Het RTOS is gemaakt in 11 header en c-files waardoor het overzichtelijk blijft en makkelijk te begrijpen is. Doordat de Atom threads zo minimaal is, is het mogelijk om vele architecturen te gebruiken. En door een specific architecture module te gebruiken 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 met een netwerk. Het systeem heeft ondersteuningen voor onder andere IPv6, RPL,UDP, CoAP, TCP en CBOR. Ook heeft het volledige ondersteuning voor speciaal low voltage applicaties en wordt ook het 6LoWPAN ondersteund. Qua architecturen worden 8-bit(bijv. AVR Atmega ) , 16-bit (bijv. TI MSP430), 32-bit( bijv. ARM Cortex) ondersteund. Het is ook mogelijk om RIOT als een proces uit te voeren onder linux en os x. RIOT bevat voor verschillende modules de nodige drivers, zoals verschillende omgeving sensoren maar ook ondersteuning voor verschillende servo motors.

# Vergelijkingen 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 het Arduino RTOS met focus op de handeling van taken en de beschikbare concurrency mechanisme.

Voor het vergelijken van de vijf open-source RTOS-en met het Arduino RTOS gaan we deze een voor een langs.

**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 taak een andere benadering genomen dan het Arduino RTOS. Bij het Arduino RTOS zijn de taken gebaseerd op objecten, dus dit betekent 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 taak aangemaakt die de functie bevat.

Voor de aanmaak van een taak moet bij FreeRTOS een aantal parameter worden meegegeven: de desbetreffende functie, de taak naam, stack\_size, de parameters die je aan een functie moet doorgeven, de prioriteit van de taak en de xHandle (voor suspend en resume). In tegenstelling tot Arduino RTOS waar bij aanmaak voor een taak in de klas drie parameters moeten worden gegeven: naam van de taak, prioriteit en de stackgroote. Voor het handelen met de taken heeft de FreeRTOS een aantal functies. VtaskDelay() wordt gebruikt voor het stilzetten van een bepaalde periode, dit is vergelijkbaar met de sleep() functie in de Arduino RTOS. De functies vTasksuspend() en vTaskResume() zijn de vergelijkbaar functies in de Arduino RTOS suspend() en Resume(). De vTasksuspend() kan de taak op een pauze worden gezet en door gebruik van vTaskResume() kan de taak weer worden uitgevoerd. De functies vTaskPriorityGet() geeft de priority van een task weer en de functie vTaskPrioritySet() kan de prioriteit van een taak veranderen. In de Arduino RTOS bevindt zich een equivalent voor vTaskPriorityGet() genaamd priority() deze geeft ook als unsigned int return de prioriteit van een taak. Een prioriteit kan echter niet worden veranderd in het Arduino RTOS.

Voor de concurrency mechanisme bevat de Arduino RTOS de volgende mechanisme: flag, timer/clock, pool, mailbox, channel en mutex.

De FreeRTOS heeft in tegenstelling tot Arduino RTOS 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 groep op TRUE staan. FreeRTOS bevat de nodige functies om een clock en timer te maken doormiddel van XTimer(). Deze functie bevat dezelfde opties als de Arduino RTOS timer. 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 dezelfde class kunnen worden gebruikt, waar FreeRTOS juist gebruik maakt van een struct die zelf aangemaakt moet worden, in deze struct wordt een array 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 get() 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 functies voor specifieke concurrency mechanisme dan de Arduino RTOS, alhoewel de Arduino RTOS wel beschikt over mailbox en pool en die niet bevinden in FreeRTOS.

**mbed**

De tweede open source RTOS die we hebben besproken is mbed.

#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);

    }

}

Mbed is een op klassen gebaseerd RTOS zoals het Arduino RTOS.

In mbed wordt een taak 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 (om het begrijpelijk te houden gebruiken we de benaming taak) maar in deze situatie doet elk precies hetzelfde. Voor het aanmaken van een taak 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 taak 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 dezelfde benaming. Mbed beschikt over het gebruik van een mutex die op eenzelfde manier wordt 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 genaamd 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 get() mbed beschikt alleen niet over een functie waarmee een channel kan worden gecleard. Voor pool bevat mbed 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 bevindt 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 is erbij de task geen equivalent voor suspend() en resume() en het beschikt niet over een clock functie.

**Chibi-OS/rt**

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

Voor het aanmaken van een taak (in deze RTOS weer als thread ) wordt gebruik gemaakt van de chThdCreateI() functie, aan de functie moet worden meegegeven: een 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 bevindt 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 ChibichThdSuspendS() deze functie zet de desbetreffende thread op pauze en kan daarna weer worden gestart met 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 bevindt zich niet standaard een clock functie. Voor de flag bevindt zich in Chibi een event klas. In het Arduino RTOS flag klas bevind zich twee tal functies wait() en set(), wait() laat wachten tot de flag is aangezet en doormiddel van set() wordt en flag aangezet. De equivalenten functies voor wait en set zijn chEvtWaitOne() en chEvtSignal().

Voor een 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 het Arduino RTOS bevinden zich de functies write() en read(). De equivalenten van de eerder genoemde functies in Chibi zijn 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, een mailbox kan dus in chibi ook gebruikt worden met een array. Het hoeft dus niet altijd een enkele waarde te zijn . 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 het 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 bevindt 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 functies als de Arduino RTOS en dit zou dus een eventuele vervanging kunnen zijn voor de Arduino RTOS.

**Atom threads**

In Atom threads worden taken(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, deze functies zijn 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 heeft Atom threads een functie genaamd atomQueueCreate(). En Atom threads heeft een drietal 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 get(). 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. Hieruit kan worden geconcludeerd dat Atom threads niet geschikt zou zijn om de lasergame code op te runnen.

**RIOT**

Voor het aanmaken van een taak (opnieuw genoemd als thread) wordt 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 het 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() vanuit het 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 wordt 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 het 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.

# Hoe kunnen de mechanismen van het Arduino RTOS die niet direct worden ondersteund door de beschikbare RTOSen worden gerealiseerd m.b.v. van deze RTOS-en?

**FreeRTOS**

FreeRTOS mist de concurrency mechanismen pool en mailbox. Een pool is simpelweg een stukje geheugen wat waardes kan houden en door meerdere klassen toegankelijk is. De missende pool is hierdoor heel makkelijk op te lossen. Een globale variabele of een klasse met de variabele erin is hier een goede vervanger voor. Wel moet elke klasse die de klasse oplossing van de pool gebruikt, deze "pool" ook kennen.

De mailbox heeft data-overdracht en een mechanisme wat de taken laat wachten totdat de dataoverdracht volledig voltooid is. Een andere oplossing voor een mailbox is dan ook een combinatie van een pool en 2 flags. Hierin is de pool voor de data-overdracht en kunnen de 2 flags gebruikt worden voor de synchronisatie van de mailbox. De eerste flag is voor het aangeven dat de data veranderd is. De tweede flag wordt gebruikt voor het aangeven dat de data ook werkelijk gelezen is.

**Mbed**  
Mbed is vrij volledig, maar het mist wel een klok. De klok is een systeem wat een in de zoveel tijd een event activeert waardoor een taak bijvoorbeeld iets gaat doen. Een timer kan dan ook als klok gebruikt worden. Hierbij stel je de tijd die gewacht moet worden in, ga je op het signaal hiervan wachten en wanneer dat signaal komt stel je de tijd opnieuw in na het uitvoeren van wat er bij het kloksignaal moet gebeuren.

**Atom threads**  
Atom threads mist echt heel veel concurrency mechanismen. Om deze reden is een API de beste manier om deze missende mechanismen te vervangen. Een API is een verzameling van definities die voor meerdere delen van het programma toegankelijk is.

**RIOT**  
Het systeem RIOT is vrij compleet. Het enige onderdeel wat mist is een channel. Een channel is een rij van waardes van hetzelfde type. Bij het schrijven wordt er achterin de rij een waarde bij gezet en bij een read voorin de rij iets weggehaald wordt en alle waardes doorgeschoven worden. Dit is makkelijk zelf te maken. Als eerste is er een template class nodig waarbij er een array type aangegeven moet worden en een maximale grootte. Deze array heeft een huidige index voor de schrijf plek en de lees plek zit aan het begin. Dan zijn er 2 functies, een leesfunctie en een schrijffunctie. De schrijffunctie checkt of de huidige schrijfplek al gelijk is aan de maximale grootte en indien dit niet zo is zet hij de waarde in de array. De leesfunctie leest de voorste waarde, verplaatst dan alles en geeft de gelezen waarde terug.

**Chibi-OS/rt**

Van alle mechanismen die wij gevonden hebben is Chibi-OS/rt wel de beste vervanging van de HU Arduino RTOS. Het bevat alle concurrency mechanismen. Ook heeft deze rtos support voor de Arduino Due.

# Conclusies en aanbeveling

De hoofdvraag van dit onderzoeksrapport luide:

*Met het behulp van welk open source realtime operating system kunnen tasks en de concurrency mechanismen, pool, channel, flag(group), colck timer en mutex, zoals aangeboden door het Arduno RTOS, met zo weinig mogelijk overhead worden gerealiseerd*

Door de verschillende deelvragen te beantwoorden zijn wij tot de conclusie gekomen dat het ….. RTOS-systeem hier het meest geschikt voor is.

**Hier uitleggen op welke deelvragen dit is gebaseerd.**

**Aanbeveling**

Op grond van deze conclusie doen we een aantal aanbevelingen over het gebruik van dit RTOS-systeem bij dit specifiek project:

**Aanbevelingen hier**