# **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 tik 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 schrijven 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.