REVISTA ELECTRO, Vol. 45 pp. 8-14, Oct 2023, Chihuahua, Chih. México
https://itchihuahua.mx/revista_electro
ISSN 1405-2172

DESARROLLO DE UNA LIBRERÍA CONCURRENTE EN FREERTOS EN POO

Beltrán-Gómez Oscar, Márquez-Gutiérrez Pedro-Rafael, Ruiz-Varela Oscar-Ramsés, Sandoval-Rodríguez Rafael, Trujillo-Preciado Edgar
Tecnológico Nacional de México / Instituto Tecnológico de Chihuahua
Ave. Tecnológico #2909, Chihuahua, Chih. México. C.P. 31310 Tel.. +52 (614) 2-01-2000
oscar.bg@chihuahua.tecnm.mx, pedro.mg@chihuahua.tecnm.mx, oscar.rv@chihuahua.tecnm.mx, rafael.s@chihuahua.tecnm.mx, edgar.tp@chihuahua.tecnm.mx

RESUMEN.
La percepción generalizada de las tecnologías de información ha generado una búsqueda de respuestas instantáneas independientes del dispositivo o de su tecnología. Esta solicitud exhaustiva de recursos computacionales ha obligado a la pronta evolución de las tecnologías y sistemas embebidos. Para solventar la demanda del alto nivel computacional fue necesario el desarrollo del Multiprocessor System-on-Chips (MPSoC). Este avance, incluye sus propios retos como la brecha en el uso de lenguajes y técnicas modernas de programación.Sin embargo, la luz al final del camino es la aparición y el uso de implementaciones modernas como FreeRTOS para la implementa concurrencia y paralelismo en varias marcas y modelos de MPSoC's y SoC's, pero aunque FreeRTOS es un excelente framework aún se nota la separación entre las implementaciones de alto nivel como las que proporciona la Programación Orientada a Objetos. En este artículo se desarrolla una librería que incorpora un nivel de abstracción al Framework FreeRTOS haciendo uso de la POO para el desarrollo de sistemas concurrentes y paralelos en ambientes embebidos.

Palabras Clave: Actores, FreeRTOS, librería, POO, C++.

ABSTRACT.
The widespread perception of information technologies has generated a search for instant answers independent of the device or its technology. This exhaustive request for computational resources has forced the rapid evolution of embedded technologies and systems. To meet the demand for a high computational level, it was necessary to develop the Multiprocessor System-on-Chips (MPSoC). This advance includes its own challenges such as the gap in the use of modern programming languages and techniques. However, the light at the end of the road is the emergence and use of modern implementations such as FreeRTOS for implementing concurrency and parallelism in various brands and models of MPSoC's and SoC's, but although FreeRTOS is an excellent framework, the separation between the high-level implementations such as those provided by Object Oriented Programming.

Keywords: Actors, FreeRTOS, library, OOP, C++.

1. INTRODUCCIÓN.

Las fronteras tenues entre las tecnologías de información suponen una percepción genérica de las mismas; hoy se solicita la ejecución de varias tareas y se solicita la respuesta al instante sin importar el tipo de dispositivo su tamaño o capacidad.

Esta solicitud exhaustiva de recursos computacionales ha obligado a la pronta evolución de tecnologías en donde los sistemas embebidos juegan un papel importante.

Usualmente los sistemas embebidos se basan en **SoC** ó **S**ystem **o**n a **C**hip, por sus siglas en inglés. Sin embargo, para solventar la demanda del alto nivel computacional actual era necesario la combinación de varios **SoC's** que compartían a su vez el control y procesamiento [1]. La solución natural fue el desarrollo del **M**ultiprocessor **S**ystem-**o**n-**C**hips (**MPSoC**), un **SoC** con múltiples memorias y unidades de procesamiento (muchas veces heterogéneas), conectadas y comunicadas entre sí; a partir de esto, entonces cada unidad de procesamiento podría satisfacer alguna tarea en específico. [2]

Este avance, como todos, incluye sus propios retos; tal vez el más desapercibido es la brecha que aún se tiene en el uso de lenguajes y técnicas modernas de programación en los ya mencionados **SoC** y **MPSoC.** Por ejemplo, aunque **Bjarne Stroustrup**, el creador del lenguaje **C++**, menciona que uno de los objetivos principales de **C++ 11** fue *"Mejorar el rendimiento y la capacidad de trabajar directamente con el Hardware"* [3], lo cierto es que aún existe bastante renuencia en el uso del mismo en **SoC's** y **MPSoC's**, este desapego incrementa incluso si hablamos del uso de técnicas y paradigmas de programación modernas. También, aquí, se ve el caso de la **P**rogramación **O**rientada a **O**bjetos (**POO**) que a pesar de su amplio éxito no han sido tan aceptadas en el desarrollo de software para sistemas embebidos [4].

Esta apatía toma como base la reducción del desempeño o incluso el incremento del consumo de memoria en el desarrollo de sistemas embebidos, pero se tendría que evaluar si tal precio es incosteable al cubrir la necesidad de abstracción, modularidad y una base sólida para la computación concurrente en multiprocesadores [5].

La luz al final del camino es que, a pesar del desánimo en el uso de implementaciones modernas, se han creado frameworks y librerías para trabajar sobre ello. **FreeRTOS** es uno de los más aceptados [6], este framework implementa concurrencia y paralelismo en varias marcas y modelos de **MPSoC's** y **SoC's**. Sin embargo y aun cuando **FreeRTOS** es un excelente framework aún se hace notar la separación que tiene con las implementaciones de alto nivel como las que proporciona la **P**rogramación **O**rientada a **O**bjetos.

En este artículo se desarrolla una librería que incorpora un nivel de abstracción al Framework **FreeRTOS** haciendo uso de la **POO** para el desarrollo de sistemas concurrentes y paralelos en ambientes embebidos.

2. Bases para la Implementaciones de la Librería

Una biblioteca es un componente de software reutilizable que ahorra tiempo al proporcionar acceso al código que realiza una tarea de programación [7] que podría llegar a representar una abstracción de algún artefacto o concepto.

Esta última idea no interfiere con la representación de esas tareas como sucesos ocurridos en el mismo umbral de tiempo durante la ejecución, es decir, de forma concurrentemente, potencialmente paralela, como partes de un cálculo o proceso sin importar el orden particular de ejecución [5].

Para que eso se lleve a cabo se deben de implementar diferentes características dentro de tal librería como son:

- **POO**.- **Clases**, **Herencia** y **Polimorfismo** para especializar objetos que tienen una base en común. Esto genera la posibilidad de crear diferentes tipos de tareas.
- **Lambdas**.- Las **lambdas** con la sentencia **USING** proporcionan una forma de terminar la implementación, es decir, agrega los mecanismos para asignar funciones anónimas con tipos definidos a la ejecución de cada tarea.
- **Colas**.- Las **colas** de mensajes o **queues** son uno de los mecanismos más usados para la comunicación entre tareas de **FreeRTOS**, el uso de colas bajo este modelo brinda un sistema de comunicación alternativo y concurrente entre objetos.

La lista previamente mencionada proporciona una base para el desarrollo de las clases e implementación de una librería para tareas genéricas para su ejecución concurrente.

3. Abstracción de las Tareas de FreeRTOS en Clases

Envolver las tareas de **FreeRTOS** en clases y objetos agrega varias ventajas, la primera sin duda es la conceptualización de las tareas como objetos con un comportamiento definido.
El paradigma de **POO** aplicado a las tareas de **FreeRTOS** genera implementaciones de las mismas como objetos, lo cual permite asignarle una prioridad, un nombre y/o incluso acceso a comportamientos como detener su ejecución, todo esto desde una única "*referencia*". Esta ventaja abre la puerta a la creación de métodos adicionales que enriquezcan el comportamiento de las tareas.

Para generar las clases que "*envuelven"* las tareas de **FreeRTOS**, hace falta ver el contexto de ejecución de cada tarea. La sentencia **xTaskCreate** (código 1), ya definida en **FreeRTOS**, proporciona una buena introducción.

```
BaseType_t xTaskCreate(
                TaskFunction_t pvTaskCode,
                const char * const pcName,
                configSTACK_DEPTH_TYPE usStackDepth,
                void *pvParameters,
                UBaseType_t uxPriority,
                TaskHandle_t *pxCreatedTask
                        );
```

Código 1 - Cabecera de la función xTaskCreate [8].

Como se puede ver, esta función recibe 6 parámetros que describen su marco de ejecución.

- El primer parámetro, es la función que va a desempeñar, es decir, el código de ejecución de la misma tarea.
- El siguiente parámetro es el nombre, este parámetro es sólo informativo y da la posibilidad de generar un seguimiento más personal.
- El tercero es la memoria asignada a la tarea.
- El cuarto representa los parámetros de la función descrita en el primer parámetro.

- El quinto parámetro es la prioridad de la tarea; ya que las tareas compiten por un tiempo de procesador es necesario asignar prioridades para un buen manejo de este recurso.

- El último es el ***pxCreatedTask**, y es la referencia que usa **FreeRTOS** para apuntar a la tarea recién creada.

Se considera que cada parámetro es importante pero solo los parámetros de entrada (***pvParameters**) y el ***pxCreatedTask** pueden pasarse como nulos.

4. Implementación de las Clases

Podemos ver, en la línea 1, código 2, la declaración de una clase base. Como se ha comentado anteriormente es la encargada de instancias los objetos que fungirán como tareas. En la clase **xbTask**, expone la clase más básica de la librería y se aprecia que desde la línea 3 a la línea 8 se describen los parámetros de entrada, así mismo en la línea 11 se puede ver el apuntador de la función que va a ejecutar.

La línea 13 describe el constructor, este trae preasignado el núcleo en el cual se ejecutará la tarea, esta asignación por defecto se usa para omitir tal parámetro en futuras implementaciones en placas que solo cuentan con un solo procesador.

El método **run**, línea 23, envuelve la creación de la tarea con los parámetros antes mencionados, se etiqueta como `virtual` para su posterior re-escritura en clases derivadas.

Por último, el método **setTask**, recibe la función que ejecutará tal tarea; la firma de tal función es

`void` (*)(`void` ***pvParams**);

y es representada por el parámetro `TaskFunction_t pvTaskCode` (**imagen 1**) del método `xTaskCreate`.

En la línea 1, del código 3, se observa el uso de la sentencia **using** para generar un **alias** que define el tipo de la función anterior. El empleo de la sentencia proporciona semántica, un aspecto fundamental para la aceptación de la librería.

La clase **xTask** presentada en el código 3, línea 4, es una clase derivada de la **xbTask**, se observa que su dimensión es mínima en comparación a su clase padre, sin embargo su funcionalidad es basta, primero se presenta como una plantilla; este mecanismo integra la posibilidad de trabajar con un tipo de dato específico y designado al momento de crear cualquier instancia de la clase. Esta funcionalidad es apreciable porque concede un comportamiento genérico independiente del tipo de dato.

El segundo punto a considerar es que la especialidad de la clase **xTask** radica en que cuenta con una instancia de **xQueue<T>**, línea 6 del código 3.

La clase **xQueue**, también etiquetada como plantilla (línea 1, código 4), implementa el sistema de comunicación, su función es incorporar una estructura de datos tipo cola para que funja como mecanismo de comunicación; el concepto detrás del uso de **colas** es contar con un segmento de memoria en que las tareas puedan publicar elementos procesables para que otras a su vez puedan evaluar dichos elementos.

```
1  class xbTask {
2     protected:
3      int16_t          memSize;
4      int8_t           priority;
5      char*            name;
6      TaskHandle_t     tHandle = NULL;
7      void*            params  = NULL;
8      int8_t           xnCore;
9
10    public:
11     tTask task;
12
13     xbTask(  char* n,
14              int16_t mS,
15              int8_t p,
16              int8_t xnC = 0
17              ):
18              name(n),
19              memSize(mS),
20              priority(p),
21              xnCore(xnC) {  }
22
23      virtual void run() {
24       tHandle = xTaskCreatePinnedToCore
25 (
26                    task,
27                    name,
```

```
28                      memSize,
29                      params,
30                      priority,
31                      &taskHandle,
32                      xnCore
33                  );
34              }
35
36              xbTask* setTask( tTask t ) {
37                  task = t;
38                  return this;
39              }
40  };
```

Código 2 - Clase base, envoltura.

Para complementar la semántica de la librería se opta por agregar la sobrecarga del operador "**>>**", que como se verá más adelante esto proporciona un flujo de trabajo. La sobrecarga solo es una abreviación del uso del método **send**, línea 8 del código 3.

```
1   using tTask = void (*)(void *pvParams);
2
3   template <class T>
4   class xTask : public xbTask {
5      public:
6           xQueue<T>   queue;
7
8           void send (const T elem) {
9               queue.send(elem);
10          }
11  };
12
13  template <typename T>
14  void operator >> (const T elem, xTask<T>*
15  task)
16  {
17     task->send(elem);
18  };
```

Código 3 - Clase derivada de **xbTask**.

5. Caso de Uso para Concurrencia..

Se plantea la lectura del puerto serie al tiempo que un "led" se prende y se apaga con un retardo definido, el efecto parpadeo es implementado usualmente para comprobar el buen funcionamiento del sistema.

El problema clásico del ejemplo anterior es la ejecución secuencial; la lectura del puerto serie depende estrictamente del tiempo que le tome al microcontrolador encender y a apagar el led para generar el parpadeo; además, el reinicio del parpadeo depende del tiempo que le toma al microcontrolador leer y procesar la información que arriba desde el puerto serie.

La librería que aquí se desarrolla resuelve el problema anterior con la clase **xbTask**. El código de la **imagen 6**, implementa la respuesta al caso de uso.

Se puede observar en las primeras dos líneas las instancias de las tareas "***serial***" y "***parpadeo***", de los parámetros cabe destacar el parámetro que se refiere al núcleo, en este caso el parámetro está preasignado a cero, así que las dos tareas se ejecutarán en ese núcleo.

```
1   template <class T>
2   class xQueue {
3      public:
4           xQueueHandle xQ;
5
6      xQueue(int8_t size = 10) {
7           int msgSize = sizeof(T);
8           xQ = xQueueCreate(size, msgSize);
9      }
10
11     void send (T _elemt) {
12          if ( xQueueSendToBack( xQ, &_elemt,
13                  2000/portTICK_RATE_MS) !=
14                  pdTRUE )
15             Serial.println("error");
16     }
17  };
```

Imagen 4 - Clase **xQueue<T>.**

Otro parámetro es la prioridad de la tarea, en este caso la tarea "***serial***" tiene una prioridad más alta que la de "***parpadeo***"; aquí recordamos que las clases de la librería son envoltorios de las funciones de **FreeRTOS** y el manejo de la prioridad depende de **FreeRTOS** y no de la librería.

En la línea 5 y 20, código 5, se observa cómo se pasa la función en forma de **lambda** que representa el código de la tarea a ejecutar, lo más destacable es el alcance de la misma [9], aunque esto depende de la versión de **C++** que estemos usando por lo general se puede trabajar con un alcance más amplio y por referencia. Por último, los métodos **`run`**, líneas 18 y 32 del código 5, desencadenan la ejecución de las tareas; la **figura 1** muestra como **FreeRTOS** implementa el patrón de ejecución.

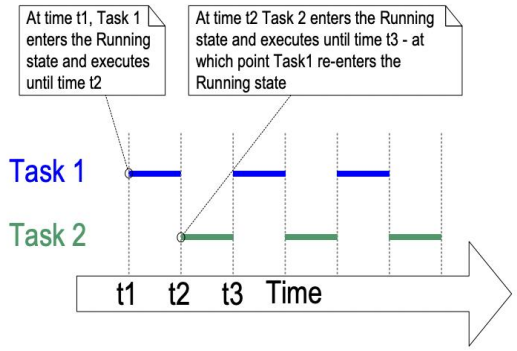


Figura 1 - El diagrama de ejecución de las tareas "***serial***" y "***parpadeo***" [10].

```
1  xbTask serial    = xbTask("Serial", 512, 3);
2  xbTask parpadeo  = xbTask("Parpadeo", 512,
3  2);
4
5  tTask tSerial = [] (void *pvParameter)-> void
6  {
7     for (;;)
8        {
9          if(Serial.available())
10         {
11           // proceso de la entrada serial
12         }
13       }
14 };
15
16 void setup() {
17
18  serial.setTask(tSerial)->run();
19
```

```
20  parpadeo.setTask([] (void *pvParameter)->
21 void
22     {
23       pinMode(led, OUTPUT);
24       for (;;)
25       {
26         digitalWrite(13, 1);
27         delay(1000);
28         digitalWrite(13, 0);
29         delay(1000);
30       }
31    }
32  )->run();
33 }
34
35 void loop()
36 { }
```

Código 5 - Implementación de respuesta para el caso de uso de concurrencia.

7. Caso de Uso de Paralelismo

No obstante, en esta situación, se requiere disponer de una placa que cuente con dos procesadores o **MPSoC**. Para lograr una ejecución en paralelo, es esencial que cada tarea se realice en una unidad de procesamiento separada. La figura 2 ilustra de manera gráfica el esquema de ejecución.

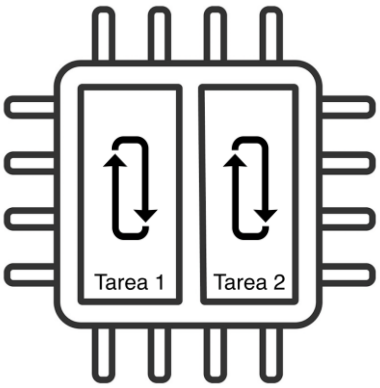


Figura 2 - El patrón de ejecución de las dos tareas de forma paralela.

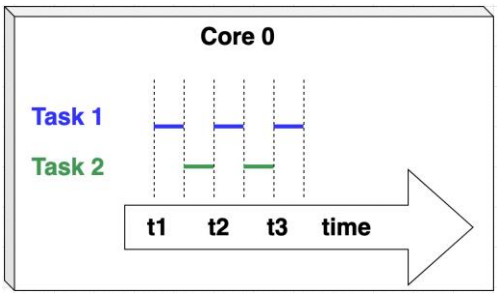
```
1  xTask serial   = xTask("Serial",   512, 3,
2  0);
3  xTask parpadeo = xTask("Parpadeo", 512, 3,
```

```
4   1);
5
6   void setup() {
7    serial.setTask([] (void *pvParameter) ->
8   void {
9         //código para leer el puerto serie
10      }
11   )->run();
12
13   parpadeo.setTask(
14      [] (void *pvParameter) -> void
15      {
16         //código para el parpadeo
17      }
18   )->run();
19  }
20
21  void loop()
22  { }
```

Código 6 - Implementación de respuesta para el caso de uso paralelo.

El código del código 6, es esencialmente el mismo del caso de uso para la concurrencia, sin embargo, la línea 1 y 2 incorporan el parámetro que indica el núcleo que ejecutará la tarea.

Es importante mencionar que bajo este esquema cada núcleo puede e implementa concurrencia, es decir, cada tarea asignada a un núcleo en particular competirá con las demás por él tiempo de ejecución del mismo, figura 3 .



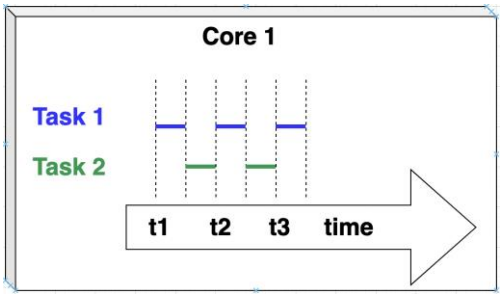


Figura 3 - Ejemplificación de tareas paralelas bajo un esquema concurrente.

8. El Modelo de Actores

En la programación orientada a objetos es habitual y funcional representar objetos mediante métodos o acciones de entrada y salida, los cuales establecen un protocolo claro y significativo de interacción. Este enfoque es el que se utiliza en el modelo de actores [11].

Los actores, al igual que las tareas vistas en este artículo, son componentes autónomos, interactivos e independientes de un sistema informático que se comunican mediante el paso de mensajes asincrónicos [12].

*"El **término actor** fue introducido por **Carl Hewitt** en el MIT a principios de la década de **1970** para describir el concepto de agentes razonadores. Se ha refinando a lo largo de los años en un modelo de concurrencia"* [5].

Las acciones ó primitivas básicas del comportamiento de un actor son : **create**, **sent to** y **become**.

create (**crear**): Se refiere a crear un actor con una funcionalidad bajo su propio contexto de ejecución; esto se hace al pasarle la función **tTask** a la clase **xTask**.

send to (**enviar a**): envía un elemento procesable de forma asíncrona que se coloca en el buzón (en la **xQueue** del objeto) del actor como una forma de comunicación con él. En caso de esta librería la dirección de cada Actor es la referencia de la instancia misma en la que podemos usar el método send o el operador **">>"** para tal envío;

become (**convertirse en**): describe que un actor adapta su ejecución; como cada instancia maneja su propio contexto el estado del mismo depende de sus propias variables, en esta librería incluso se podría cambiar la función que se ejecuta por otra a partir del método **setTask**.

En el modelo de actores, el cambio de estado se especifica usando comportamientos de reemplazo **setTask**.

Estas primitivas, aunque básicas, dan pie a abstracciones y paradigmas de alto nivel en programación concurrentes [13].

A partir de este momento, cada vez que un actor procesa un elemento de su buzón, también calcula su comportamiento en respuesta al próximo envío que pueda procesar [14].

Los elementos que llegan al buzón de cada actor en esencia pueden ser denominados eventos y representan la forma en que los actores pueden comunicarse. La información contenida en el evento, es variada, sin embargo, basta el agregar la referencia del actor que lo envía (en remitente) para generar un ciclo de trabajo.

Aunque la **OTP** u Open Telecom Platform, por sus siglas en Inglés, en Erlang y Elixir proporcionan ya un conjunto de bibliotecas y principios de diseño estamos seguros que su adopción en plataformas y lenguajes orientados a software embebido será más amplia. Prueba de ello es **Nerves** que es una plataforma de código abierto que combina la máquina virtual **BEAM** (la máquina virtual de Erlang) y el lenguaje Elixir para construir sistemas embebidos [15].

9. Conclusiones.

El uso de colas en **FreeRTOS** está abierto a la implementación del usuario final, si bien esto proporciona libertad creativa, cuando se habla de concurrencia es mejor tener un modelo o guía de referencia.

El **Modelo de Actores** (**MA**) es la guía que aquí se ha adoptado para este fin. La clase **xTask** junto con la clase **xQueue** pueden llegar a representar la implementación de un **Actor** y generar la aproximación al **MA** en esta librería.

Son perceptible las condiciones descritas por **Nelson Rodríguez** al comentar que "*El advenimiento de la concurrencia masiva a través de la computación en la nube y las arquitecturas de computadora multi-núcleo ha estimulado el interés en el Modelo de Actores*" [16], sin embargo, se cree que aún está por venir un mayor auge para este modelo.

10. Referencias.

[1] Chicaiza, W. M., & Verdesoto, D. G. (2013). Diseño e Implementación de un Multiprocessor Systems-on-Chip (MPSoC) Interconectado por una Networks-on-Chip (NoC). MASKAY, 3(1), 40–48. https://doi.org/10.24133/maskay.v3i1.129

[2] https://journal.espe.edu.ec/ojs/index.php/maskay/article/view/129/pdf

[3] Schranzhofer, Andreas & Chen, Jian-Jia & Thiele, Lothar. (2010). Dynamic Power-Aware Mapping of Applications onto Heterogeneous MPSoC Platforms. Industrial Informatics, IEEE Transactions on. 6. 692 - 707. 10.1109/TII.2010.2062192.

[4] Bjarne Stroustrup"Mejorar el rendimiento y la capacidad de trabajar directamente con el Hardware" https://www.stroustrup.com/C++11FAQ.html

[5] puedo citar Alberto Pacheco, http://electro.itchihuahua.edu.mx/revista/2021/C-Sub32.pdf

[6] Gul Agha. 1990. Concurrent Object-Oriented Programming. Commun. ACM 33, 9 (sep 1990), page 126.

[7] AWS FreeRTOS. (1 de 6 de 2023). Un sistema operativo de tiempo real para dispositivos limitados por los recursos. Obtenido de https://docs.aws.amazon.com/iot/latest/developerguide/iot-sdks.html.

[8] Colebourne, S. & Java Magazine Written by the Java community for Java and JVM developers [Java Magazine]. (2020, January 20). *Designing and Implementing a Library*. Java Magazine. Retrieved May 8, 2023, from https://blogs.oracle.com/javamagazine/post/designing-and-implementing-a-library

[9] FreeRTOS Tasks and Co-routines. (1 de 6 de 2023). FreeRTOS Tasks and Co-routines. Obtenido de https://www.freertos.org/taskandcr.html

[10] Lambda Expressions. (1 de 6 de 2023). C++ Reference Lambda Expressions. Obtenido de https://en.cppreference.com/w/cpp/language/lambda

[11] Barry, R. & FreeRTOS. (2016). Mastering the FreeRTOSTM Real Time Kernel, A Hands-On Tutorial Guide (Pre-release 161204 Edition.). © Real Time Engineers Ltd. 2016. https://www.freertos.org

[12] L. Eunsang. Developing a low cost microcontroller based model for teaching and learning. European Journal of education research. 2020 pages 921-934.

[13] Agha, G. (2004). ACTORS: A Model of Concurrent Computation in Distributed Systems. Cambridge, Massachusetts: MIT Artificial Intelligence Laboratory, pages 141.

[14] Gul Agha. 1990. Concurrent Object-Oriented Programming. Commun. ACM 33, 9 (sep 1990), page 128.

[15] Nerves Project. (n.d.). © The Nerves Project Authors 2023. Retrieved July 24, 2023, from https://nerves-project.org/

[16] Rodríguez, N. R., Murazzo, M. A., & Runco, T. (2019). El modelo de programación de actor aplicado a Edge Computing utilizando Calvin. Web