

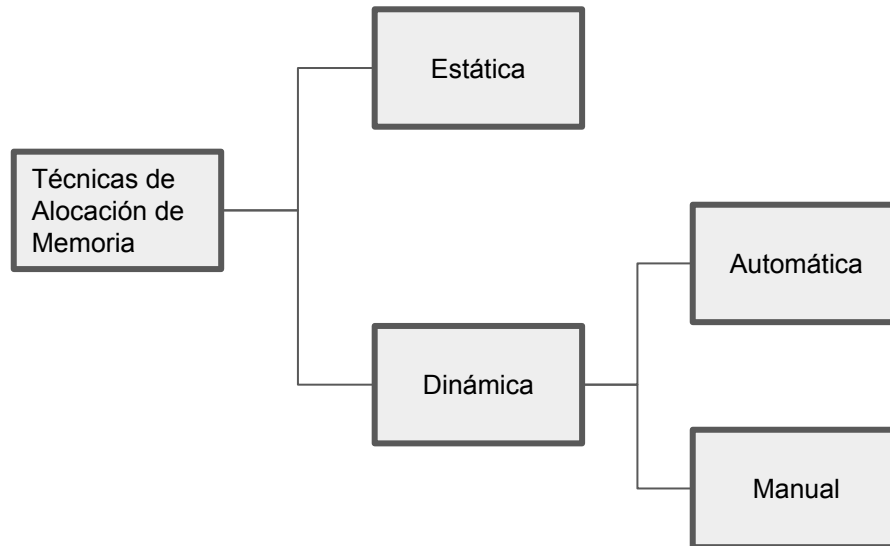
# Carrera de Especialización en Sistemas Embebidos

## Sistemas Operativos en Tiempo Real II

### Clase 1: Alocación de memoria en RTOS

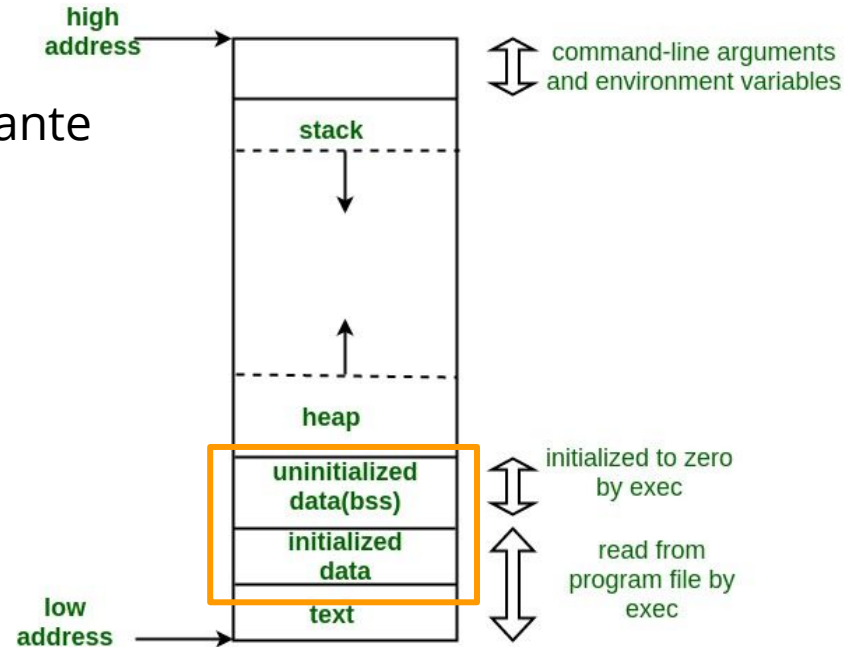
# ALOCACIÓN DE MEMORIA en RAM

- La asignación de memoria habla de cómo un programa "hace lugar" en la memoria RAM para ser utilizada para almacenar una cierta estructura de datos.
  - Depende del lenguaje



# MEMORIA: ALOCACIÓN ESTÁTICA

- La memoria estática es la memoria que se reserva en tiempo de compilación.
  - AKA: Variable global
- La "vida" de esta memoria se extiende durante TODA la ejecución del programa.
- El mapa generado de este tipo de memoria, es conocido desde antes que se ejecute el programa.
  - Utilización determinista.



# MEMORIA: ALOCACIÓN AUTOMÁTICA

- La memoria dinámica automática es la que se reserva, se usa y se desecha cuando la ejecución de un programa entra en un cierto scope (entorno)
  - AKA: Variable local
  - La zona de memoria usada es el stack (OJO con el compilador)
- La "vida" de esta memoria se extiende durante la ejecución del scope que la utiliza, y después queda automáticamente "liberada" (otra zona del "programa" puede pisarla)

## Ejemplo A

```
char procesar(char cantidad)
{
    char buffer[cantidad];

    recibir_por_uart( buffer , cantidad );

    char cmd = procesar_paquete(buffer,cantidad);

    return cmd;
}
```

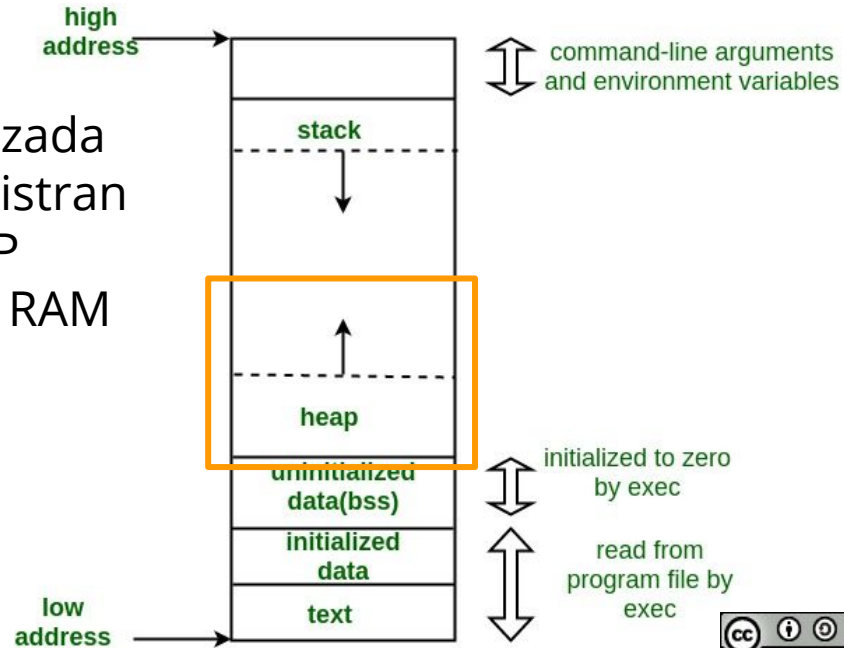
## Ejemplo B

```
void procesar_protocolo(uint32_t protocol)
{
    int delimitador;

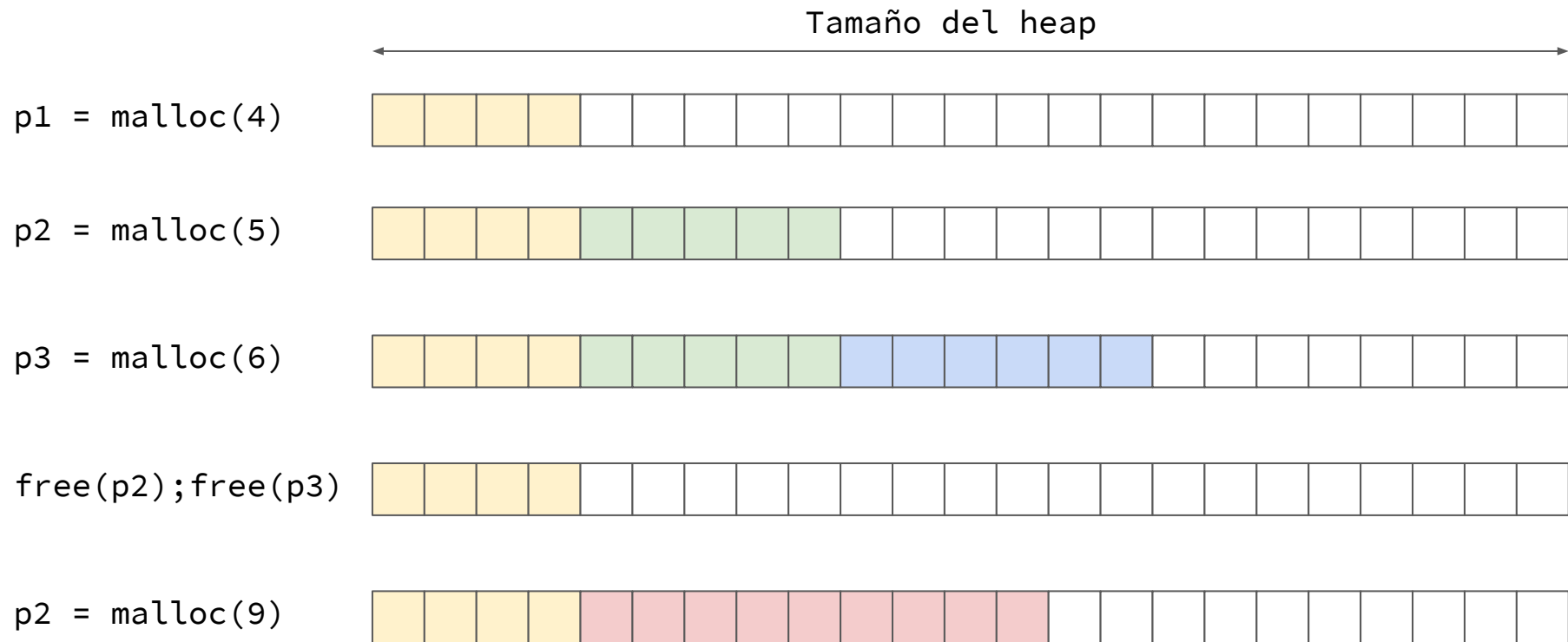
    if( PROTOCOLO_A == protocol )
    {
        int delimitador;
        delimitador = get_delimiter_protoclo_A();
        procesar_delimitador(PROTOCOLO_A , delimitador );
    }
    if( PROTOCOLO_B == protocol )
    {
        int delimitador;
        delimitador = get_delimiter_protoclo_B();
        procesar_delimitador(PROTOCOLO_B , delimitador );
    }
}
```

# MEMORIA: ALOCACIÓN DINÁMICA MANUAL

- La memoria dinámica manual es la que el programador "manualmente" solicita o devuelve.
  - Se trabaja a través de punteros.
- La "vida" de esta memoria es controlada por el programador.
- La gestión de "memoria dinámica" es realizada por un conjunto de programas que administran un espacio de memoria denominado HEAP
  - El heap en general está definido por "la RAM que sobre".
- En c estándar se utiliza:
  - malloc (para "pedir" memoria al heap)
  - free (para "devolver" memoria al heap)



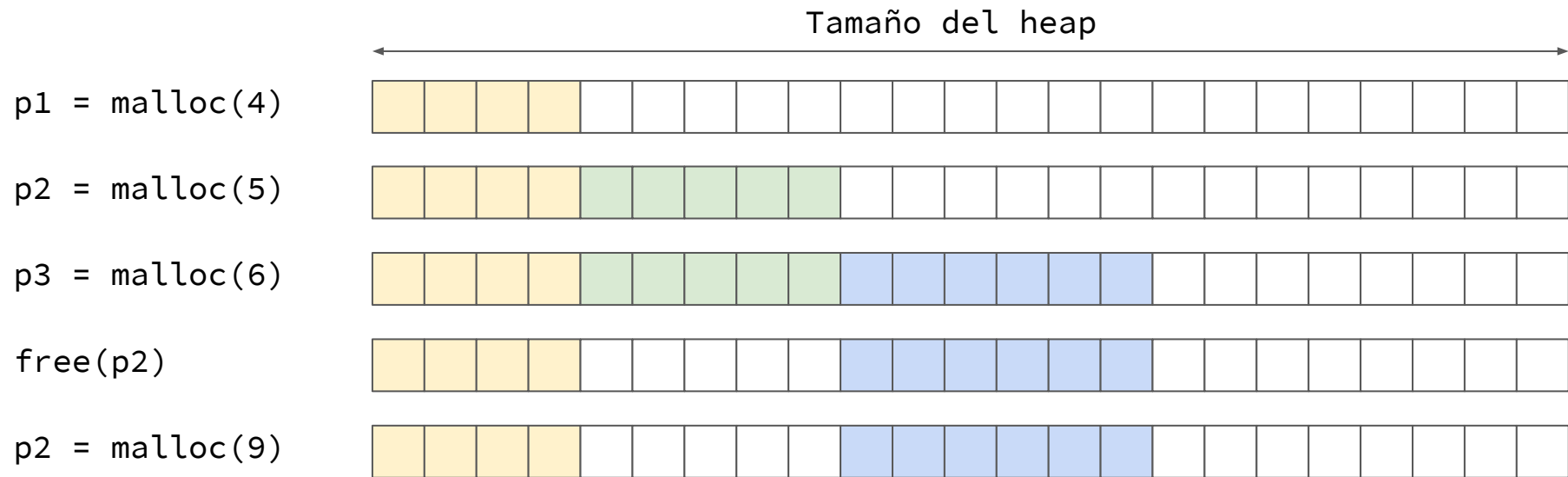
# ALOC. DINÁMICA: FUNCIONALIDAD



# ALOC. DINAMICA MANUAL: PROBLEMÁTICAS

- Fragmentacion:
  - El uso de malloc y free de manera poco controlada podría "agujerear" el heap de manera que el programador puede quedarse sin memoria al "solicitar" un cierto espacio, no pudiendo realizar la tarea asociada al uso de la misma.
- Memory Leaks
  - El usuario puede olvidarse de liberar una zona al no usarla más, quedando un hueco desperdiciado en el heap.
- Liberación antes de tiempo
  - El usuario libera la memoria antes de terminar de utilizarla. Esto podría ocasionar bugs bastante complicados de encontrar (la memoria liberada podría estar siendo escrita por otro hilo de ejecución)

# MEMORIA DINÁMICA: FRAGMENTACIÓN

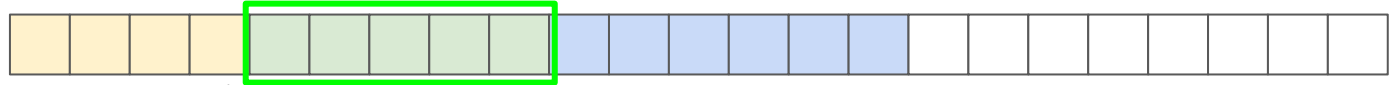


p2 va a valer NULL porque no pudo allocarse memoria.

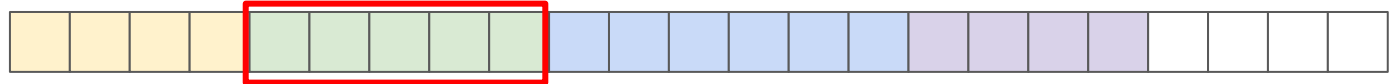


# MEM. DINÁMICA: MEMORY LEAK

p1 = malloc(5)



```
int HacerAlgo(void)
{
    char * p1 = malloc( 5 );
    /* Hacer algo */
    return 0;
}
```



AL SALIR DE LA FUNCIÓN ESTA ZONA DEL HEAP NADIE PODRÁ LIBERARLA.

# MEM. DINÁMICA: LIBERACIÓN TEMPRANA

## ALGORITMO CON FALLA

```
p1 = malloc(10)
```

```
free(p1);
```

```
recibir_de_uart1(p1,5)
```

```
if(p1[0]==':')  
{  
    /* procesar modbus  
ASCII*/  
}
```

LIBERA ANTES DE USAR

CAMBIO DE CONTEXTO

```
char *p = malloc(15);
```

```
recibir_de_uart2(p , 10)
```

```
if(p[10]=='\n')  
{  
    /* procesar otro protocolo */  
}
```

```
free(p);
```

# RESOLUCIÓN DE LEAKS EN OTROS LENGUAJES

- Garbage Collection:
  - Algoritmo que corre en otro hilo de ejecución, que rastrea los espacios alocados sin punteros que los referencien.
- Reference Counting:
  - Cada vez que un espacio de memoria se iguala a un puntero, un registro interno incrementa, y cada vez que el puntero se libera, se decrementa.
  - El algoritmo evalúa los registros, y los bloques con el contador en cero son liberados efectivamente.

# ALOCACIÓN. DINÁMICA: ALGORITMOS

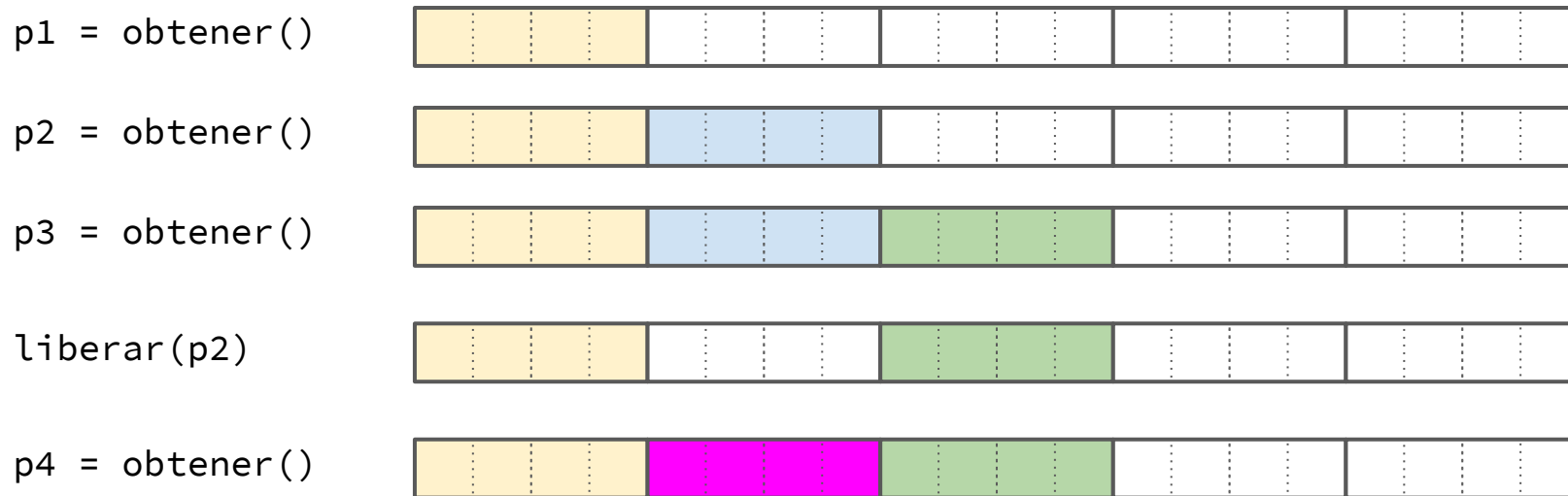
- ¿Cómo se elige un algoritmo de asignación de memoria?
  - Tiempo de malloc, lo mas chico posible.
  - Tiempo de malloc, lo más constante posible.
  - Capacidad de unir bloques que quedaron libres.
  - Overhead de la implementación.
- Algoritmos de memoria dinámica:
  - Sequential Fit (variantes [Best Fit](#), [Worst Fit](#), [First Fit](#), [Next Fit](#)) : listas enlazadas
  - [Buddy Allocator](#) : Listas enlazadas de listas de bloques del mismo tamaño
  - [Indexed Fit](#) : Indexa la zonas de memoria libres por tamaño para obtener.
  - [Bitmapped Fit](#) : Utiliza un mapa de memoria para identificar bloques usados.
  - [Two Level Segregation](#)

# OTRO ESQUEMA DE MEMORIA DINÁMICA

- Técnicas de pools de bloques para evitar la fragmentación.
  - El mecanismo de memory pools que permite:
    - Hacer alocaiones de tamaño fijo (la mínima unidad no es el byte, sino el tamaño de bloque)
    - Este mecanismo es bueno para cubrir algunas necesidades (incluso para uso interno del OS que los incorpora).
    - Si un OS no lo incorpora, se puede implementar o descargar de algun sitio. También se puede "tomar prestado" de algún framework (MBed, [QP](#), y otros)

# ALOCACIÓN DINÁMICA POR BLOQUES

- Cada pool de memoria (se pueden instanciar varios en una misma aplicación) puede dar un bloque de tamaño fijo.
  - Ej: pool de 5 bloques de 4 bytes

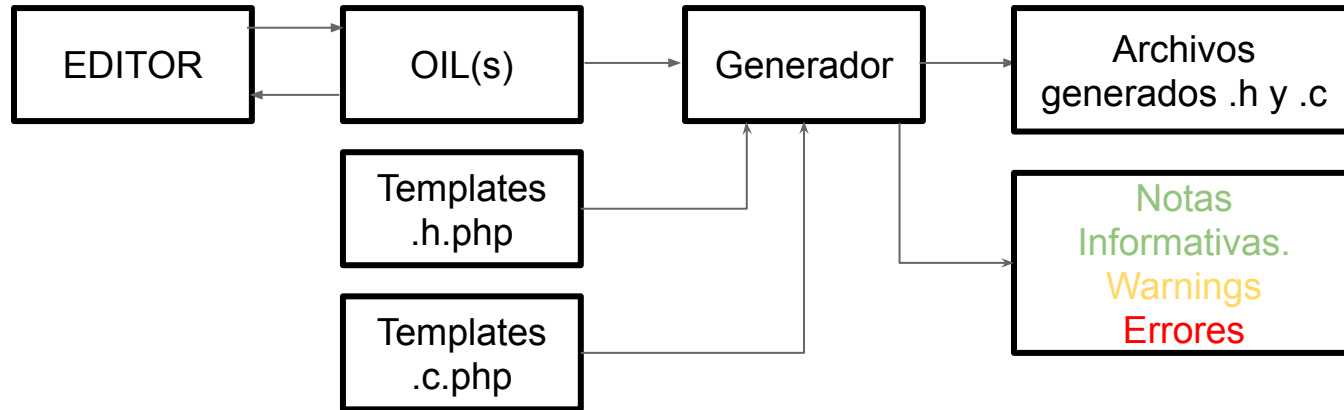


# ARQUITECTURAS DE LOS RTOS (MEMORIA)

- Existen:
  - RTOS con instanciación de objetos 100% estática (ej: OSEK)
  - RTOS con instanciación de objetos parcialmente estática (ej: FreeRTOS )
  - RTOS con instanciación de objetos manera dinámica eligiendo el algoritmo a utilizar (uno de los que trae FreeRTOS o alguno de terceros) (ej: FreeRTOS )

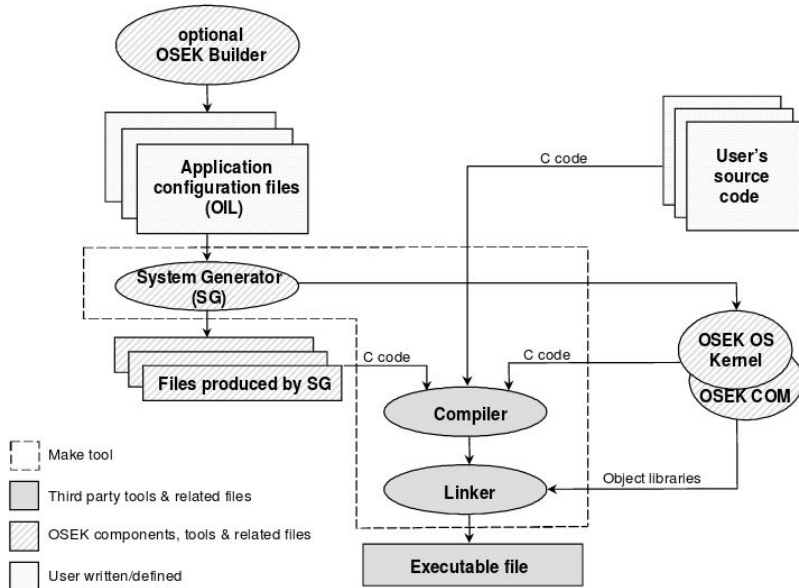
# Ej: OSEK: Arquitectura de compilación

- Utilizando una aplicación de SW un grupo de archivos de configuración se traducen a un grupo de archivos .c y .h
- El **proceso de generación** utiliza al conjunto de archivos .OIL y a una serie de templates.





# Ej: OSEK




- Ventajas
  - Portabilidad
  - Determinismo
  - Mayor desempeño
  - Menos gasto de recursos
  - Testeabilidad
- Desventajas
  - Cada vez que se realiza una generación nueva, se recompila todo

# ALOCACIÓN ESTÁTICA en FreeRTOS



- El freertos permite al usuario declarar objetos, pero utilizando memoria asignada de manera estática (global).
- Esto permite más control al usuario porque:
  - Se pueden ubicar en lugares específicos del mapa de RAM
  - Se conoce de antemano si la RAM necesaria para la aplicación, está disponible.
- Las funciones de la API para instanciar objetos con asignación estática, finalizan con **...Static**.

 `xTaskCreateStatic()  
xQueueCreateStatic()  
xTimerCreateStatic()  
xSemaphoreCreateBinaryStatic()  
xSemaphoreCreateCountingStatic()  
xSemaphoreCreateMutexStatic()`

`freeRtosconfig.h`

- Para habilitar asignación estática, deberá definirse `configSUPPORT_STATIC_ALLOCATION` en 1
- Si solo quiere usarse asignación estática, `configSUPPORT_DYNAMIC_ALLOCATION` deberá definirse en 0

# ALOCACIÓN ESTÁTICA en FreeRTOS



- Ejemplo creación de una tarea, con asignación estática.

```
/* Tamaño del stack requerido en words (32 bytes en caso de que el CPU sea de ese ancho) */  
#define STACK_SIZE 200
```

```
/* Estructura q va destinada al bloque de control de la tarea (TCb - Task Control Block) */  
StaticTask_t bloque_de_control;
```

```
/* Array que definirá al stack de la tarea */  
StackType_t stack_de_tarea[ STACK_SIZE ];
```

```
/* Función asociada a la tarea */  
void Tarea( void * pvParameters )  
{  
    for( ;; )  
    {  
        /* Task code goes here. */  
    }  
}
```

```
/* Function that creates a task. */  
void main( void )  
{  
    /* ... */
```

```
    TaskHandle_t xHandle = NULL;
```

```
/* Creacion de la tarea usando asignación estática */  
xHandle = xTaskCreateStatic(  
    Tarea,  
    "NAME",  
    STACK_SIZE,  
    ( void * ) 1,  
    tskIDLE_PRIORITY,  
    stack_de_tarea,  
    &bloque_de_control );
```

```
    /* ... */  
}
```

# ALOCACIÓN DINÁMICA en FreeRTOS



- La facilidad de uso de FreeRTOS se basa en que todos los objetos del kernel utilizan zonas de memoria reservadas de manera dinámica. Esto se hace en tiempo de ejecución.
- El OS, "Pide" memoria cuando crea un objeto, y la "devuelve" cuando dicho objeto es eliminado.

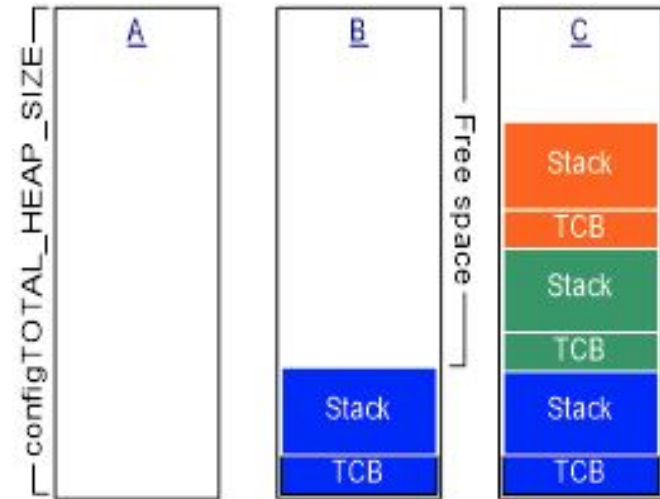
<b>Ventajas</b>	<b>Desventajas</b>
Simplificación de la API	Latencias
Flexibilidad en tiempo de ejecución	Riesgos de fragmentación
Menor carga al usuario	Otros errores en tiempo de ejecución
Reutilización de RAM	Mayor huella de memoria

- FreeRTOS implementa los algoritmos de memoria dinámica, de manera portable. Esto permite al usuario decidir cual usar.
- FreeRTOS provee de cinco implementaciones diferentes. El usuario puede incorporar otras librerías.
- El tamaño del heap puede definirse en FreeRTOSConfig.h a través de la macro configTOTAL\_HEAP\_SIZE
- Los nombres que utiliza son pvPortMalloc y pvPortFree.

# FreeRTOS: Implementacion heap1.c



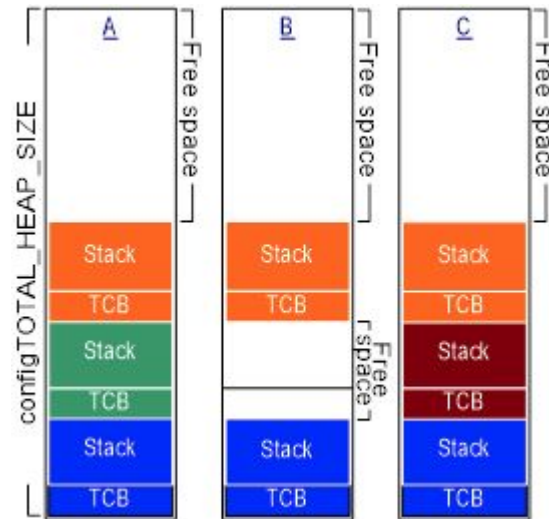
- No implementa pvPortFree
- Implementa una versión MUY reducida de pvPortMalloc
- Tiene sentido utilizarla cuando la aplicación no necesita destruir objetos en tiempo de ejecución (requerimiento que abarca una gran cantidad de sistemas)
- La memoria no sufre de fragmentación.
- Es determinista.



# FreeRTOS: Implementacion heap2.c



- Implementa pvPortFree
  - Pero dos bloques libres adyacentes (físicamente) no se unifican en uno solo.
- En pvPortMalloc Implementa el algoritmo de "best fit" para ubicar en el heap bloques libres del tamaño solicitado.
  - Recorre la lista de bloques libres, buscando el más pequeño que permita almacenar el tamaño solicitado.
  - Del bloque elegido, se separa, y lo que "sobra" queda como bloque vacío.
- Es útil cuando se necesita dinamismo (malloc/free) de un cierto tipo de objeto y que siempre posea el mismo tamaño.
  - Cuando se quiere mayor dinamismo, en varios objetos o diferentes tamaños, no sirve.



!! NRND !!

- Utiliza las implementaciones de la librería estándar de malloc/free.

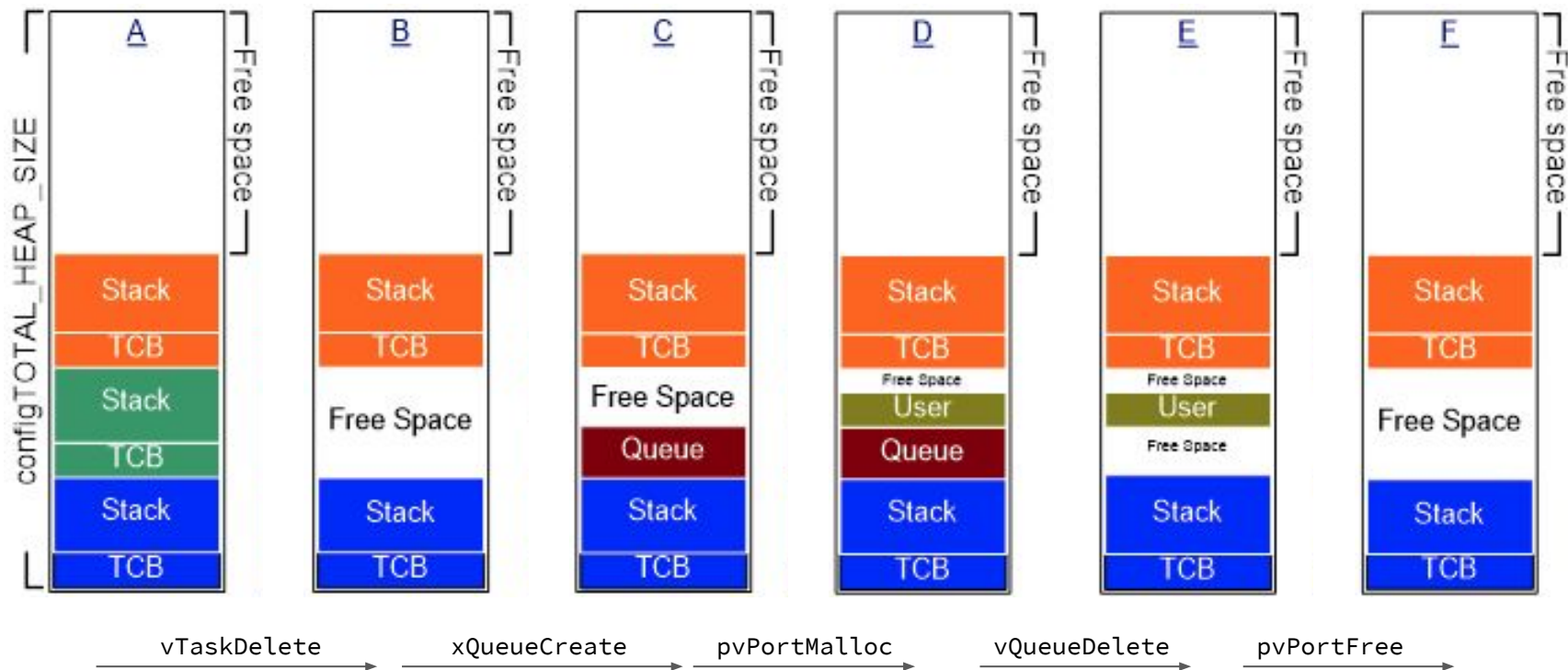
## **DEPENDENCIA DEL COMPILADOR**

- La definición de configTOTAL\_HEAP\_SIZE no tiene efecto, y el tamaño del heap queda definido por la configuración del linker file (ld)
- El llamado a pvPortFree o pvPortMalloc se realiza dentro de una zona crítica.



- Implementa pvPortFree
  - Dos bloques libres adyacentes (físicamente) se unifican en uno solo.
  - No evita la fragmentación, pero la minimiza.
- En pvPortMalloc Implementa el algoritmo de "first fit" para ubicar en el heap bloques libres del tamaño solicitado.
  - Recorre la lista de bloques libres, buscando el primero que permita almacenar el tamaño solicitado.
  - Del bloque elegido, se separa, y lo que "sobra" queda como bloque vacío.
- El usuario puede declarar el espacio estático de memoria donde ubicar el heap. Para esto debe configurarse config APPLICATION\_ALLOCATED\_HEAP en 1 en el FreeRTOSConfig.h
  - Esto es útil en uC que permitan extender el mapa de RAM usando chips externos, y el tiempo de acceso sea distinto a la RAM interna.

# FreeRTOS: Implementacion heap4.c



- Es igual al algoritmo de heap\_4
- Difiere de la versión 4 porque se puede definir al heap, no como un único array estático sino que permite el uso de espacios de memoria separados.
- La desventaja es que el algoritmo debe inicializarse.

```
uint8_t array1[ RAM1_HEAP_SIZE ];
uint8_t array2[ RAM2_HEAP_SIZE ];
uint8_t array3[ RAM3_HEAP_SIZE ];

const HeapRegion_t regiones[] =
{
    /* ref del array      , tamaño en bytes */
    { array1 , RAM1_HEAP_SIZE },
    { array2 , RAM2_HEAP_SIZE },
    { array3 , RAM3_HEAP_SIZE },
    { NULL, 0 }
};
```

```
int main( void )
{
    /* Inicialización del algoritmo de heap5 */

    vPortDefineHeapRegions( regiones );

    /* resto del main */
}
```

# ¿Cual Elegir?

- Si no se necesita dinamismo, usar la versión 1
  - (durante la materia RTOS1 podrían haber usado siempre v1)
- Si se está limitado en RAM, y las tareas del sistema no necesitan estar corriendo todas al mismo tiempo. Usar v4.
- Si la aplicación necesita de dinamismo absoluto (ej: un router) considerar usar v3 (la nativa del compilador) , v4 o portar algún algoritmo de malloc/free con alguna operatoria más óptima (ej, [TLSE](#))
- Si tu mapa de RAM está segmentado (ej: memoria externa u alguna arquitectura diferente) utilizar v5.

# ¿Cómo dimensionar al heap?

- En general la recomendación es asignarle un tamaño holgado para desarrollar y posteriormente, cuando el sistema está listo analizar minuciosamente su utilización y eventual corrección del tamaño.
- `xPortGetFreeHeapSize()` devuelve el tamaño libre de heap, si se quisiera tener la medida en tiempo de ejecución.
- `xPortGetMinimumEverFreeHeapSize()` devuelve el tamaño mas chico que tuvo el heap en algún momento de la ejecución. Si el sistema es muy dinámico, puede evaluarse en tiempo de ejecución para tener una cota, y dimensionarlo posteriormente.
- Usar la herramienta que se presentará a continuación.

# ¿Cómo portar a una LIB de Aloc. Dinámica ?

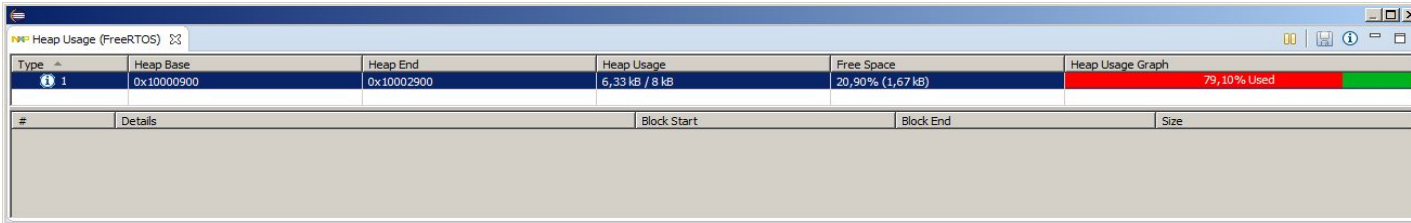
- Estudiar cómo funciona la librería a portar
  - Cómo define el espacio en RAM que gestionará
  - Que objetos estáticos se necesita instanciar
  - Mecanismo de inicialización.
- Estudiar cómo agregar a la compilación los .c y los .h que provee
  - Por ej, en firmware\_v3, si se agrega un "heapX.c" en la carpeta ..\libs\freertos\MemMang luego habrá que configurar al **config.mk** con `FREERTOS_HEAP_TYPE=X`
- Hacer un wrapper de pvPortMalloc y vPortFree utilizando como plantilla al archivo heap3.c ya incluido como parte del framework (por ej, nombrarlo heapX.c)

# Herramienta para Eclipse

Windows->Install New Software

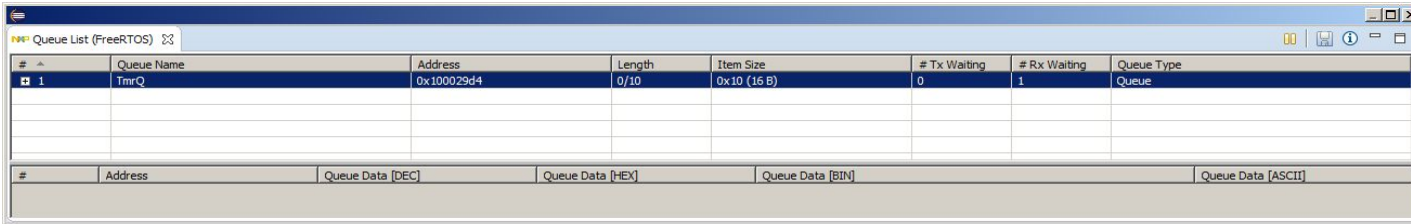
<http://freescale.com/lqfiles/updates/Eclipse/KDS>

¿Como utilizarlo?



Type	Heap Base	Heap End	Heap Usage	Free Space	Heap Usage Graph
1	0x10000900	0x10002900	6,33 kB / 8 kB	20,90% (1,67 kB)	79,10% Used

#	Details	Block Start	Block End	Size
---	---------	-------------	-----------	------



#	Queue Name	Address	Length	Item Size	# Tx Waiting	# Rx Waiting	Queue Type
1	TmrQ	0x100029d4	0/10	0x10 (16 B)	0	1	Queue

#	Address	Queue Data [DEC]	Queue Data [HEX]	Queue Data [BIN]	Queue Data [ASCII]
---	---------	------------------	------------------	------------------	--------------------



TCB#	Task Name	Task Handle	Task State	Priority	Stack Usage	Event Object	Runtime
1	TaskUART	0x10000ed8	Running	3 (3)	124 B / 712 B		0x0 (0,0%)
2	TaskLED1	0x10001210	Blocked	1 (1)	108 B / 712 B		0x0 (0,0%)
3	TaskLED2	0x10001548	Ready	1 (1)	108 B / 712 B		0x0 (0,0%)
4	TaskLED3	0x10001880	Blocked	1 (1)	108 B / 712 B		0x1 (0,0%)
5	TaskMEF_B2	0x10001bb8	Suspended	2 (2)	132 B / 712 B	Unknown (0x10000a1c)	0x0 (0,0%)
6	TaskMEF_B3	0x10001ef0	Suspended	2 (2)	132 B / 712 B	Unknown (0x10000a9c)	0x0 (0,0%)
7	TaskCalc	0x10002228	Ready	1 (1)	148 B / 712 B		0x0 (0,0%)
8	IDLE	0x10002ccc	Ready	0 (0)	44 B / 356 B		0x1free (100,0%)
9	Tmr Svc	0x10002d30	Blocked	4 (4)	112 B / 1,4 kB	TmrQ (Rx)	0x0 (0,0%)

# Bibliografia

- [Amazon FreeRTOS - Heap Management](#)
- Introducción a los Sistemas operativos de Tiempo Real, Alejandro Celery - 2014
- Introducción a Sistema Operativo OSEK, CESE, Franco Bucafusco, 2017
- Memory allocation for real time operating system, Asma'a Lafi, Jordan University of Science and Technology
- <http://www.cs.virginia.edu/~son/cs414.f05/>
- [Problem with Dynamic Memory Allocation, AticleWorld, Amlendra](#)
- [Embedded Basics - 7 Tips for Managing RTOS Memory Performance and Usage, Jacob's Blog, 2017](#)
- [FreeRTOS Kernel Awareness for Eclipse from NXP, MCU On Eclipse, Erich Styger, 2016](#)
- [How to Allocate Dynamic Memory Safely, BARR group, Niall Murphy, 2016](#)
- [Memory Management Reference](#)



# Licencia

---



"Alocación de memoria en RTOS"

Por Mg. Ing. Franco Bucafusco, se distribuye bajo una [licencia de Creative Commons Reconocimiento-CompartirIgual 4.0 Internacional](https://creativecommons.org/licenses/by-sa/4.0/)