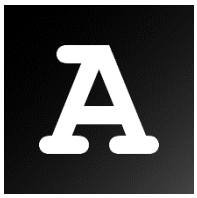


ANTiPoDE Framework

Another Pic32 Development Environment
https://github.com/ANTiPoDE-Framework/AFramework
http://www.antipode-dev.org
info@antipode-dev.org - milazzo.ga@gmail.com

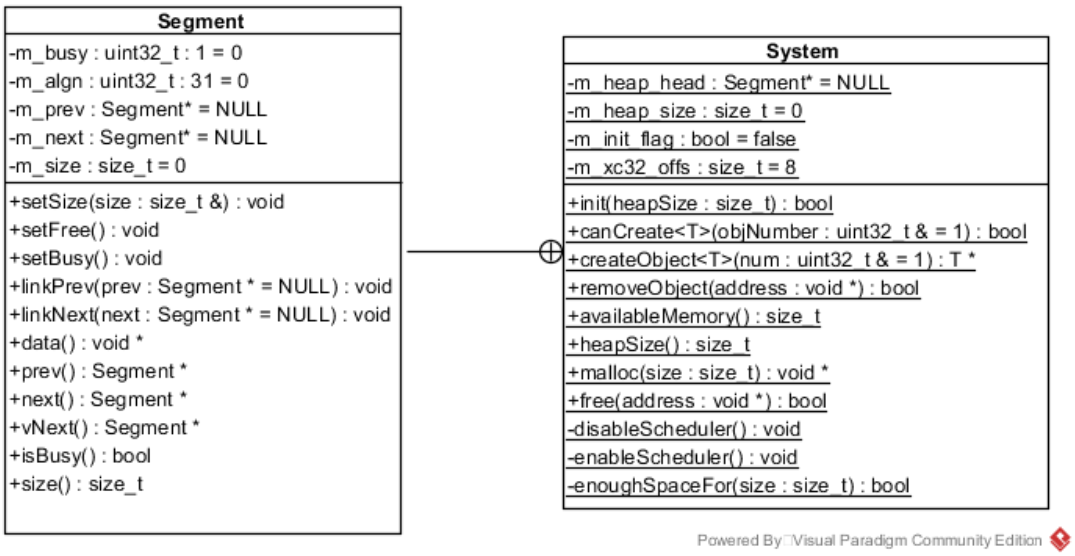
1 - Gestione della memoria

1.1 - Introduzione

L'heap è un'area di memoria non inizializzata usata per l'allocazione dinamica della memoria tramite le funzioni della libreria standard del C, quali `calloc`, `malloc` e `realloc`, o del C++ con l'operatore `new`. Tramite l'IDE MPLAB X fornito da Microchip è possibile specificare la dimensione dell'heap nelle impostazioni di progetto ed automaticamente tale dimensione sarà passata durante la fase di compilazione al linker xc32-ld [(1)]. Questa procedura seppur molto semplice, presenta i seguenti limiti:

- La libreria standard non offre alcuna funzione per sapere quanta memoria è occupata o disponibile.
- L'utilizzo dell'operatore `new` non è sicuro in quanto, dal momento che il supporto alle eccezioni non è ancora implementato, un fallimento nell'allocazione tramite detto operatore causa il crash del microcontrollore [(2)].

Per sopperire a ciò si è scelto di implementare dei metodi per la gestione dinamica della memoria interni al framework, che permettono di allocare, de-allocare, deframmentare e tracciare lo spazio in memoria, amministrando l'heap creato tramite il linker xc32-ld; tali metodi sono statici e posti all'interno della classe `AFramework::System` (files `ACore.h` e `ACore.cpp`) il cui diagramma UML è riportato di seguito [(a)].



Dal momento che l'heap costituisce un blocco continuo di memoria, quello che i metodi fanno è costruire una lista doppiamente concatenata su tale blocco dove ogni nodo, costituito dalla classe `AFramework::System::Segment` privata, rappresenta un'unità di allocazione di dimensione variabile.

1.2 - Dati Membro Della Classe System

Nome	*Tipo*	*Valore Inizale*	*Funzione*
`m_heap_head`	**`System::Segment *`**	**`NULL`**	Puntatore alla testa della lista dei segmenti.
`m_heap_size`	**`size_t`**	`0`	Dimensione dell'heap in byte.
`m_init_flag`	**`bool`**	**`false`**	Flag per tenere traccia dell'avvenuta inizializzazione del framework.

m_xc32_offs	**size_t**	8	Costante per l'overhead introdotto dal compilatore xc32-gcc o xc32-g++

1.3 – Metodi Pubblici Della Classe System

Prototipo	**static bool** init(**size_t** heapSize)	
Descrizione	Alla versione attuale inizializza il framework allocando l'heap (nelle successive versioni saranno presenti altri parametri come il clock di sistema, etc.).	
Parametri	**size_t** heapSize	Dimensione dell'heap dichiarata nelle impostazioni di progetto.
Restituisce	**true**	Se l'inizializzazione è andata a buon fine.
	false	Se il framework è già stato inizializzato oppure si verifica un errore nell'allocazione dell'heap.

Prototipo	**template** <**class** T> **static bool** canCreate(**const uint32_t** & objNumber = 1)	
Descrizione	Verifica che nell'heap vi sia una zona di memoria libera di dimensione maggiore o uguale a quella richiesta tramite il tipo T con cui è invocato e il parametro objNumber dove quest'ultimo indica il numero di oggetti che si vogliono allocare rendendo sicura l'allocazione della memoria sopperendo ai limiti presentati dall'architettura e dalla libreria standard del C/C++	
Parametri	**const uint32_t** & objNumber	Numero di oggetti che si vogliono creare (di default vale 1)
Restituisce	**true**	Se esiste un blocco libero che soddisfa i requisiti di spazio.
	false	Se non esiste un blocco libero che soddisfa i requisiti di spazio.

Prototipo	**template** <**class** T> **static** T * createObject(**const uint32_t** objNumber = 1)	
Descrizione	Richiama il metodo malloc passandogli come argomento la dimensione del tipo T con cui è chiamato moltiplicata per il parametro objNumber (nelle versioni future, una volta che la gerarchia di classi sarà completata, imposterà anche in flag per l'auto-rimozione dall'heap).	
Parametri	**const uint32_t** & objNumber	Numero di oggetti che si vogliono creare (di default vale 1).
Restituisce	T *	Se l'allocazione è andata a buon fine.
	NULL	Se l'allocazione non è andata a buon fine (non è stato trovato spazio sufficiente nell'heap).

Prototipo	**static bool** removeObject(**void** * address)	
Descrizione	Sinonimo per **static bool** free(**void** * address)	
Parametri	-	-
Restituisce	-	-

Prototipo	**static size_t** availableMemory()	
Descrizione	Scorre la lista dei segmenti sommando ad una variabile temporanea la dimensione del blocco se questo è libero.	
Parametri	-	-
Restituisce	-	La somma delle dimensioni dei segmenti contrassegnati come liberi.

Prototipo	**static size_t** heapSize()	
Descrizione	-	
Parametri	-	-
Restituisce	-	La dimensione dell'heap passata in fase di inizializzazione (se il framework non è stato inizializzato, ovviamente, zero).

Prototipo	**static void** * malloc(**const size_t** & size)
Descrizione	Ricerca nella lista dei segmenti un blocco di memoria libero da restituire al chiamante. Per evitare spreco di risorse viene prima ricercato un blocco di dimensione esattamente uguale a quella richiesta tramite size; se questo blocco viene trovato sarà direttamente restituito dopo essere stato marcato come occupato, altrimenti si procede alla ricerca del blocco di dimensione massima tra tutti i blocchi liberi che riesce a soddisfare i requisiti di spazio richiesti (in questo modo si riduce la frammentazione della memoria). Trovato il massimo viene scisso in due per formare due nuovi blocchi: il primo sarà marcato come occupato e restituito al chiamante mentre il secondo sarà marcato come libero e quindi reso disponibile per altre allocazioni. Se invece non viene trovato un

	blocco che soddisfa i requisiti di spazio allora la funzione restituisce **NULL**. Proprio per tale motivo è scoraggiato l'utilizzo di tale metodo se non per particolari scopi mentre si consiglia l'uso di `createObject` in combinazione con `canCreate` che forniscono, invece, un approccio più sicuro a tale problema. Questo metodo è predisposto per essere thread-safe infatti al suo interno viene immediatamente disabilitato lo scheduler per essere riattivato subito prima di restituire la memoria.	
Parametri	**const size_t** & size	Dimensione in byte dello spazio da allocare.
Restituisce	**void** *	Se l'allocazione è andata a buon fine.
	NULL	Se l'allocazione non è andata a buon fine (non è stato trovato spazio sufficiente nell'heap).

Prototipo	**static bool** free(**void** * address)	
Descrizione	Questo metodo, oltre a liberare la memoria precedentemente allocata con `malloc` (dove con questa si fa riferimento a quella implementata all'interno del framework) o `createObject` svolge la parte più delicata del problema della gestione della memoria, ovvero la deframmentazione. Partendo dalla testa della lista si cerca una corrispondenza con il parametro `address`, non appena questa viene trovata il blocco è azzerato e marcato come libero. Successivamente il metodo controlla se il blocco precedente è anch'esso libero, in questo caso I due blocchi vengono uniti in uno solo collegando questo con il blocco successivo. Se anche quest'ultimo è libero allora i due blocchi vengono nuovamente uniti per ricomporre un unico blocco. In questo modo si recuperano nel caso peggiore i soli byte occupati in origine, mentre nel caso migliore la somma dei byte dei tre blocchi coinvolti più 32 byte di overhead	
Parametri	**void** * address	Indirizzo di memoria che deve essere liberato.
Restituisce	**true**	Se sono state trovate corrispondenze con `address` ed il rilascio della memoria è andato a buon fine.
	false	Se non sono state trovate corrispondenze con `address` o il blocco era già libero oppure `address` è **NULL**.

1.4 – Futuri Miglioramenti

Allo stato attuale l'overhead introdotto per ogni unità di allocazione è 16 byte (4 byte per il bit-field di Segment, 8 byte per i due `Segment *` e altri 4 byte per la dimensione) e l'ordine computazionale dei metodi è sempre O(n). Una prima miglioria possibile, che si spera di riuscire ad introdurre prima della release del framework, è la riduzione dell'overhead per unità di allocazione ad 8 byte, infatti:

- I due `Segment *` sono stati inseriti per comodità ed è possibile eliminarne uno (al prezzo però di rendere più complesso il codice rendendo la lista non più doppiamente concatenata e utilizzando più memoria nello stack).
- Dal momento che in un microcontrollore PIC32 la memoria va da 4 KB a 64 KB per la famiglia PIC32MX o da 128 KB a 512 KB (estendibili a 64 MB con interfaccia EBI/SQI ed SRAM esterna) si ritiene che sia uno spreco utilizzare per la dimensione del segmento il tipo `size_t` a 4 byte quando questa può essere memorizzata nel bit-field (dove in questa versione 31 bit sono inutilizzati); ed anche in questo caso si riuscirebbero ad allocare blocchi di 2 GB.

1.5 – Storico Delle Revisioni

- 06.04.2016 – Completata versione 0.1 della documentazione.

1.6 – Note

(a) Sono riportati solo i metodi coinvolti nella gestione della memoria.

1.4 – Riferimenti Esterni

(1) "9.8 Dynamic Memory Allocation" - DS50001686H - MPLAB® XC32 C/C++ Compiler User's Guide
(2) "C++ Try & Catch Catch Fails Under Pic32MX" - http://www.microchip.com/forums/m780701.aspx