**Управление памятью.**

RTOS ядро выделяет RAM память всякий раз, когда создается очередь, мьютекс, программный таймер или семафор. Функции стандартной C библиотеки malloc() и free() могут иногда использоваться для этой цели, но...

1. Они не всегда доступны во встраиваемых приложениях,
2. Они занимаю много места (кодового пространства),
3. Они не являются потокобезопасными и
4. Они не являются детерминированными (время, необходимое для выполнения этих функций меняется от вызова к вызову).

... так что чаще всего требуется альтернативная реализация выделения памяти.

В разных embedded / real time системах требования к ОЗУ и временным характеристикам могут сильно различаться, поэтому единственный алгоритм выделения памяти всегда пудет подходить ограниченному подмножеству приложений. Для решения этой проблемы FreeRTOS оставляет API выделения памяти в ее аппаратно зависимом уровне. Аппаратно зависимый уровень выходит за рамки исходных файлов, которые реализуют функциональность ядра RTOS, позволяя таким образом разработать специализированную реализацию приложения с требуемыми временными параметрами. Когда ядро RTOS запрашивает RAM, вместо вызова malloc(), оно вызывает pvProtMalloc(). Когда память требуется освободить, вместо вызова free(), RTOS ядро вызывает vPortFree().

**Реализации выделения памяти, включенные в дистрибутив RTOS.**

Дистрибутив RTOS включает 4 простых релизации выделения памяти, каждая из которых описана в следующих подразделах. Эти подразделы также включают информацию о том, когда каждая из предоставленных реализаций может быть выбрана как наиболее подходящая.

Каждая реализация содержится в отдельном файле исходного кода (heap\_1.c, heap\_2.c, heap\_3.c и heap\_4.c), которые размещены в каталоге Source/Portable/MemMang дистрибутива RTOS. Если требуется, могут быть добавлены другие реализации. Ровно один из этих файлов должен быть включен в проект.

**heap\_1.c.**

Это самая простая реализация из всех. Она не позволяет освобождать память после того, как она была выделена. Несмотря на это, heap\_1.c подходит для большого числа встраиваемых приложений. Это потому, что большинство глубоко встраиваемых приложений создают все задачи, очереди, семафоры и пр., требуемые во время загрузки системы и затем использует все эти объекты в течении всего жизненного цикла программы (до тех пор, пока приложение не выключится снова, или не перезагрузится). Никогда ничего не удаляется.

Реализация просто делит одиночный массив на небольшие блоки. Общий размер массива (общий размер кучи) устанавливается константой configTOTAL\_HEAP\_SIZE - который определен в файле FreeRTOSConfig.h. API функция xPortGetFreeHeapSize() возвращает общее количество пространства кучи, которое осталось пока не выделенным, что позволяет оптимизировать общий размер кучи configTOTAL\_HEAP\_SIZE.

Реализация heap\_1:

* Может быть использована, если ваше приложение никогда не удаляет задачи, очереди, семафоры, мьютексы и т.д. (на самом деле таковых большинство, в которых используется FreeRTOS).
* Всегда детерминирована (всегда занимает одно и тоже время для выполнения).

**heap\_2.c.**

Эта схема использует best fit алгоритм и, в отличие от схемы 1, позволяет освобождать прежде занятые блоки памяти. Однако она не комбинирует смежные свободные блоки в один большой блок (она не включает алгоритм коалесценции, см. heap\_4.c, реализация которого выполняет коалесценцию свободных блоков).

Общее количество доступного пространства кучи устанавливается константой configTOTAL\_HEAP\_SIZE, которая определена в файле FreeRTOSConfig.h.

API функция xPortGetFreeHeapSize() возвращает общее количество пространства кучи, которое осталось не выделенным (что позволяет оптимально выбрать размер кучи), но не предоставляет информации о том, как фрагментирована невыделенная память.

Реализация:

* Может быть использована всегда, когда приложения периодически удаляют задачи, очереди, семафоры, мьютексы и пр. принимая во внимание факт фрагментации памяти, который оговорен ниже.
* Не должна использоваться, если выделенная и освобожденная память имела случайный размер. Например:
  + Если приложение динамически создает и удаляет очереди, и область хранения очереди одна и таже в каждом случае (область хранения очереди это размер элемента очереди, умноженный на длину очереди), то heap\_2.c может быть использован в большинстве случаев. Однако, если область хранения очереди не одинакова в каждом случае, то доступная свободная память может стать фрагментированной во много малениких блоков, в конечном счете приводящим к провалу выделения памяти. heap\_4.c был бы лучшим выбором в этом случае.
  + Если приложение динамически создает и удаляет задачи, и размер стека, выделенного под задачи всегда один и тотже, то heap\_2.c может использоваться в большинстве случаев. Однако, если размер стека, выделенного под задачи не всегда тот же самый, то доступная свободная память может быть фрагментирована на несколько мелких блоков, в итоге это приводит к провалу при попытке выделения памяти heap.c\_4 был бы лучшим выбором для этого случая.
  + Вызов приложением функций pvPortMalloc() и vPortFree() предпочтительнее, чем просто косвенный вызов через другие API функции RTOS.
* Может привести к возможным проблемам, если ваше приложение создает очереди, задачи, семафоры и пр. в непредсказуемом порядке. Это маловероятно почти для всех приложений, но должно учитываться.
* Не детерминирована - но более эффективна, чем большинство стандартных C реализаций.

heap\_2.c подходит для большинства небольших систем реального времени, которые динамически создают задачи.

**heap\_3.c.**

Эта схема реализует просто обертку стандартных C функций malloc() и free(), которые в большинстве случаев поддерживаются выбранным вами компилятором. Эта оболочка просто делает функции malloc() и free() потокобезопасными.

Эта реализация:

* Требует, чтобы линковщик установил кучу, и библиотеку компилятора, обеспечивающую malloc() и free() реализации.
* Не детерминирована.
* Возможно значительно увеличит размер ядра.

Заметьте, что configTOTAL\_HEAP\_SIZE устанавливаемый в файле FreeRTOSConfig.h не действует, когда используется модуль управления памятью heap\_3.c.

**heap\_4.c.**

Эта схема использует first fit алгоритм и в отличие от схемы 2, она комбинирует смежные свободные блоки памяти в один большой блок (она включает алгоритм коалесценции).

Общее количество доступного пространства кучи устанавливается константой configTOTAL\_HEAP\_SIZE - которая определена в конфигурационном файле FreeRTOSConfig.h.

API функция xPortGetFreeHeapSize() возвращает общее количество незанятого пространства кучи (что позволяет оптимизировать общий размер кучи), но не предоставляет информации о том, как фрагментирована не выделенная память.

Эта реализация:

* Может использоваться всегда, когда приложение периодически удаляет задачи, очереди, семафоры, мьютексы и т.д.
* Гораздо меньше вероятность, по сравнению с реализацией heap\_2, что куча будет ужасно фрагментирована в многочисленные маленькие блоки - даже когда выделяемая и освобожденная память имеет случайный размер.
* Не детерминирована, но намного эффективнее, чем большинство стандартных C функций управления памятью.

heap\_4.c особенно полезна для приложений, которые хотят использовать переносимые схемы выделения памяти прямо в коде приложения.