**heap\_2.c**

В начале файла определяется макроконстанта:

#define configADJUSTED\_HEAP\_SIZE

(configTOTAL\_HEAP\_SIZE - portBYTE\_ALIGNMENT)

С комментарием: несколько байтов могут быть потеряны из-за байтового выравнивания начального адреса кучи.

configTOTAL\_HEAP\_SIZE - это общий размер кучи, определенный в FreeRTOSConfig.h. Кстати файл конфигурации FreeRTOS можно найти только в демо проектах. К ядру он никакого отношения не имеет. Для Cortex-M4F демо это значение равно (**75 \* 1024**).

portBYTE\_ALIGNMENT - макроконстанта, определенная в файле portmacro.h из каталога Source\portable\[компилятор]\[архитектура], в моем случае - Source\portable\IAR\ARM\_CM4F. Ее значение равно **8**. Константа определяет выравниевание в памяти. Т.е. все данные в куче должны располагаться по адресу, кратному 8. Начальный адрес кучи также должен быть обязательно выровнен по границе 8ми байтового слова. Т.е. адрес любого элемента данных в куче должен иметь три нулевых младших бита.

***NB***: выравнивание адреса к памяти необходимо, т.к. это существенно сказывается на производительности процессора. Более того, некоторые процессоры не поддерживают доступ к данным по не выровненному адресу (например попытка чтения 4ех байтов информации по адресу 0x001 вырождается в несколько операции чтение 4ех байтов по адресу 0x000, затем отсечка младшего, затем сдвиг старших трех, затем чтение следующих четырех байтов по адресу 0x004, отсечка старших трех, снова сдвиг и, наконец, объединение. Некоторые процессоры не имеют столь «умных» механизмов и просто выбрасывают исключение, предоставляя их обработку системе).

Таким образом, получаем, что скорректированный размер кучи:

configADJUSTED\_HEAP\_SIZE = 75 \* 1024 - 8 = 76792.

***NB***: 8 байт резервируется на процедуру выравнивания адреса. Это легко пояснить следующим образом. Адрес первого элемента кучи &ucHeap[0] может быть каким угодно, и не обязательно кратным 8ми. Поэтому при инициализации кучи младшие три бита искусственно отсекаем, чтобы адрес был выровненным. При этом адрес первого элемента массива может «отскочить» назад на число байтов, не превышающих portBYTE\_ALIGNMENT. По этому и требуется имеено такой резерв. И тогда первым байтом кучи будет уже ucHeap[portBYTE\_ALIGNMENT], отсечка адреса которого никогда не станет меньше &ucHeap[0].

Далее объявляется статическая функция, инициализирующая структуры кучи перед ее первым использованием.

static void prvHeapInit( void );

Память под кучу выделяется как статический массив элементов типа unsigned char, размером configTOTAL\_HEAP\_SIZE.

static unsigned char ucHeap[ configTOTAL\_HEAP\_SIZE ];

Определяется тип связанного списка. Он используется, чтобы связать свободные блоки в порядке их размера.

typedef struct A\_BLOCK\_LINK

{

/\*<< следующий свободный блок в списке. \*/

struct A\_BLOCK\_LINK \*pxNextFreeBlock;

/\*<< размер свободного блока. \*/

size\_t xBlockSize;

} xBlockLink;

Константа portBYTE\_ALIGNMENT\_MASK. Определена в заголовочном файле Source\include\portable.h. Значение этой константы определяется на основе portBYTE\_ALIGNMENT. В нашем случае оно равно 8, поэтому маска выравнивания равна 0x0007. Т.е. 0b0111. Если инвертировать это значение получим: ~portBYTE\_ALIGNMENT\_MASK = **0xFFF8**.

Создается пара статических линков списка, чтобы пометить начало и конец списка.

static xBlockLink xStart, xEnd;

Объявляется статическая переменная с инициализатором, которая отслеживает число оставшихся свободных байтов, но ничего не говорит о фрагментации. Переменная инициализируется значением скорректированного размера кучи, т.к. куча пока вся свободна.

static size\_t xFreeBytesRemaining = configADJUSTED\_HEAP\_SIZE;

Для уменьшения глубины вызовов функций, некоторые статические функции определены как макросы. Далее следуют определения этих функций.

**Вставка блока в список. prvInsertBlockIntoFreeList().**

Вставляет блок в список свободных блоков, который упорядочен по размеру блока. Маленькие блоки – вначале списка и большие блоки – в конце списка.

**Инициализация памяти prvHeapInit().**

Рассмотрим определение функции инициализации кучи. Функция является статической, и в ее названии присутствует префикс "prv": prvHeapInit().

Объявляется две локальные переменные-указатели:

xBlockLink \*pxFirstFreeBlock;

unsigned char \*pucAlignedHeap;

Вся куча представляет собой последовательность блоков различных размеров. В начале каждого блока присутствует заголовок блока - структура типа xBlockLink. В заголовке указывается адрес следующего свободного блока и размер текущего блока. pxFirstFreeBlock - это указатель на первый свободный блок. Тип указателя xBlockLink позволяет ему корректно уложить в куче оба его поля. Но сначала необходимо получить его адрес. Для хранения адреса первого блока создан указатель pucAlignedHeap.

pucAlignedHeap - стартовый адрес кучи, выровненный по границе 8 байтов, т.е. имеющий три нулевых младших разрядов адреса. Для получения этого адреса необходимо выполнить операцию "AND" над адресом 8-ого байта буфера кучи (нумерация байтов в куче начинается с нуля) и маской байтового выравнивания адреса. Тогда каким бы не оказался адрес 8ого байта, стартовый адрес кучи всегда будет в пределах массива ucHeap - см. "NB" выше. Первый оператор тела функции выполняет эту операцию:

pucAlignedHeap =   
(unsigned char\*)(((portPOINTER\_SIZE\_TYPE) &ucHeap[portBYTE\_ALIGNMENT]) &   
((portPOINTER\_SIZE\_TYPE)~portBYTE\_ALIGNMENT\_MASK));

xStart используется для хранения указателя на первый элемент в списке свободных блоков. void используется для предотвращения выдачи компилятором предупреждений. Иными словами, xStart - это маркер первого свободного блока.

xStart.pxNextFreeBlock = ( void \* ) pucAlignedHeap;

xStart.xBlockSize = ( size\_t ) 0;

xEnd используется для пометки конца списка свободных блоков, т.е. фактически, является маркером последнего свободного блока.

xEnd.xBlockSize = configADJUSTED\_HEAP\_SIZE;

xEnd.pxNextFreeBlock = NULL;

После того, как получен стартовый адрес кучи, его можно присвоить указателю pxFirstFreeBlock, и затем, с помощью этого указателя инициализировать заголовок блока:

pxFirstFreeBlock = ( void \* ) pucAlignedHeap;

pxFirstFreeBlock->xBlockSize = configADJUSTED\_HEAP\_SIZE;

pxFirstFreeBlock->pxNextFreeBlock = &xEnd;

**Выделение памяти pvPortMalloc().**

Прототип функции имеет вид:

void \*pvPortMalloc( size\_t xWantedSize );

Вначале функции объявляются три указателя на структуру xBlockLink:

xBlockLink \*pxBlock, \*pxPreviousBlock, \*pxNewBlockLink;

pxBlock - указатель на текущий блок, pxPreviousBlock - указатель на предыдущий блок, pxNewBlockLink - указатель на новый блок.

Далее объявляется булевский признак инициализации динамической памяти, со значением pdFALSE по умолчанию.

static portBASE\_TYPE xHeapHasBeenInitialised = pdFALSE;

Объявляется указатель на выделенный блок, который возвращается пользователю. По умолчанию имеет значение NULL.

void \*pvReturn = NULL;

Приостанавливается выполнение всех задач с помощью вызова API функции:

vTaskSuspendAll();

{

Если pvPortMalloc() вызывается в первый раз (что определяется по признаку xHeapHasBeenInitialised), то необходимо выполнить инициализацию памяти:

if( xHeapHasBeenInitialised == pdFALSE )

{

prvHeapInit();

xHeapHasBeenInitialised = pdTRUE;

}

Желаемый размер памяти, запрошенный пользователем должен быть увеличен на значение длины заголовка блока, которое представляет собой размер структуры xBlockLink, выровненный по границе 8 байтов, и хранящийся в константе heapSTRUCT\_SIZE. Затем получившийся результат необходимо выровнять по границе 8 байтов. Очевидно, операция выделения памяти имеет смысл только при положительном запрашиваемом размере.

if( xWantedSize > 0 )

{

xWantedSize += heapSTRUCT\_SIZE;

if((xWantedSize & portBYTE\_ALIGNMENT\_MASK)!=0)

{

xWantedSize+=(portBYTE\_ALIGNMENT-(xWantedSize&portBYTE\_ALIGNMENT\_MASK));

}

}

*Пример 1*.

Пользователь запрашивает 150 байтов памяти. Запрашиваемая память должна быть увеличена на длину заголовка и выровнена по границе 8 байтов. Предположим, размер структуры xBlockLink с учетом выравнивания составляет 16 байтов. Тогда желаемый размер:

xWantedSize = 150 + 16 = 166 байтов.

Проверим на необходимость выравнивания: 166 = 0xA6. Маска выравнивания равна 0x07. Тогда выражение 0xA6 & 0x07 даст результат 0x06, который не равен нулю. Следовательно, нобходимо выполнить выравнивание:

xWantedSize = 166 + (8 - 6) = 168 = 0xA8 с тремя нулевыми младшими битами. Следовательно, получившийся в результате размер, кратен 8ми.

Теперь можно попытаться выделить пользователю память в куче, при условии, что она положительна, и не превышает максимально допустимый размер. Для этого проходим по всему списку, вполоть до концевого маркера xEnd, пока не найдем свободный блок подходящего размера. В качестве итератора используется переменная-указатель pxBlock. Проход начинаем с первого блока, на которое указывает поле pxNextFreeBlock стартового маркера. Нам также понадобится указатель на предыдущий блок, устанавливаемый вначале в позицию стартового маркера.

if((xWantedSize>0) && (xWantedSize<configADJUSTED\_HEAP\_SIZE))

{

pxPreviousBlock = &xStart;

pxBlock = xStart.pxNextFreeBlock;

while((pxBlock->xBlockSize<xWantedSize)&&(pxBlock->pxNextFreeBlock!=NULL))

{

pxPreviousBlock = pxBlock;

pxBlock = pxBlock->pxNextFreeBlock;

}

Предположим, что функция выделения памяти вызвана в первый раз, и у нас единственный свободный блок, представляющий всю кучу. Тогда на первой же итерации условие цикла while будет провалено (по выражению pxBlock->xBlockSize<xWantedSize) и функция продолжит работу.

Если свободного блока не нашлось, т.е. bxBlock достиг маркера конца списка xEnd (условие итерации цикла pxBlock->pxNextFreeBlock!=NULL было провалено), считаем что память выделить не возможно и переходим к обработке факта невозможности выделения памяти. Хотя на самом деле память просто может быть сильно фрагментирована - суммарный объем свободной памяти может быть и достаточен, однако эта реализация менеджера памяти не предусматривает дефрагментацию, и оперирует только отдельными блоками.

Если в куче нашелся свободный блок, то можно определить его адрес с учетом заголовка. Для этого к адресу блока необходимо прибавить длину заголовка (выравнивание здесь не требуется, так как длина заголовка и размеры всех блоков уже выровнены).

if( pxBlock != &xEnd )

{

pvReturn = (void\*) (((unsigned char\*)pxPreviousBlock->pxNextFreeBlock) + heapSTRUCT\_SIZE);

Далее, так как этот блок теперь используется пользователем, его надо исключить из списка. Затем от него необходимо выделить свободную часть (если таковая имеется), добавить к ней заголовок, и вставить новый получившийся блок в список.

// "короткое замыкание" блока в списке

pxPreviousBlock->pxNextFreeBlock = pxBlock->pxNextFreeBlock;

if((pxBlock->xBlockSize - xWantedSize) > heapMINIMUM\_BLOCK\_SIZE)

{

// Новый блок начинается с адреса:

pxNewBlockLink = (void\*)(((unsigned char\*)pxBlock) + xWantedSize);

// Добавить размеры в новый блок и только что выделенный (видимо это   
 // пригодится при освобождании).

pxNewBlockLink->xBlockSize = pxBlock->xBlockSize - xWantedSize;

pxBlock->xBlockSize = xWantedSize;

// Осталось только добавить новый блок в список.

prvInsertBlockIntoFreeList( ( pxNewBlockLink ) );

}

Емкость оставшейся памяти теперь будет:

xFreeBytesRemaining -= pxBlock->xBlockSize;

Конец процедуры выделения.

}// if( pxBlock != &xEnd )

}// ((xWantedSize>0) && (xWantedSize<configADJUSTED\_HEAP\_SIZE))

}// vTaskSuspendAll();

Теперь возобновляем работу задач:

xTaskResumeAll();

Если в файле конфигурации FreeRTOSConfig.h операционной системы константа configUSE\_MALLOC\_FAILED\_HOOK определена как 1, то в случае провала выделения памяти вызывается обработчик:

#if( configUSE\_MALLOC\_FAILED\_HOOK == 1 )

{

if( pvReturn == NULL )

{

extern void vApplicationMallocFailedHook( void );

vApplicationMallocFailedHook();

}

}

#endif

В конце функции расположен оператор возврата адреса выделенной пользователю памяти, или NULL, в случае невозможности выполнения этого действия.

return pvReturn;

}

**Освобождение памяти vPortFree().**

Прототип функции имеет вид:

void vPortFree( void \*pv );

В теле функции сперва объявляется два указателя:

unsigned char \*puc = (unsigned char \*) pv;

xBlockLink \*pxLink;

puc -хранит адрес блока, подлежащего освобождению. Так как в начале блока распологается его заголовок в виде структуры xBlockLink, то необходимо скорректировать этот адрес:

puc -= heapSTRUCT\_SIZE;

Следующее приведение типов нужно только для того, чтобы избежать предупреждения выравнивания некоторых компиляторов:

pxLink = ( void \* ) puc;

Теперь приостанавливаем все задачи, добавляем блок обратно в пул с попутным изменением значения числа оставшихся байтов, и возобновляем задачи.

vTaskSuspendAll();

{

// Вернуть блок в список

prvInsertBlockIntoFreeList( ( ( xBlockLink \* ) pxLink ) );

// Размер блока был определен на этапе выделения памяти

// Осталось байтов в куче

xFreeBytesRemaining += pxLink->xBlockSize;

}

xTaskResumeAll();