C programs you have to run for your midpoint report include:
driver_cpg.c
heap.c

You don't have to modify driver_cpg.c at all. All you should focus on is heap.c's _ralloc( ) and _rfee( ) implementation. You'll submit:

Table 12: Step-1 Submission

Materials	Remarks	Grade points (out of 25pts)
startup_tm4c129.s	From your Keil uVersion project	2pts
stdlib.s	From your Keil uVersion project	5pts
Two memory snapshots: stringA and stringB	From your Keil uVersion project	4pts
heap.c	From your Linux C program	10pts
a.out execution results	From your Linux C execution	4pts

To complete heap.c, use the heap_template.c file where you have to fill _ralloc( ) and _rfree( ). Rename heap_template.c heap.c and thereafter compile with
gcc driver_cpg.c heap.c

Then, run it with
a.out

Now, let's look at heap_template.c. This program simulates THUMB2's DRAM area using
char array[0x8000] // array[32768] or 32KB

array[0] corresponds to address 0x2000 0000
array[32767] corresponds to address 0x2000 7FFF

This conversion from an array index to address is done with a2m( ). The reverse conversion from address to an array index is done with m2a( ). Both utility functions have been implemented.

The driver_cpg.c main( ) program calls _malloc( ) and _free( ) to request and to release a memory space.

_malloc( ) initializes the heap space just once with _kinit( ). Note that _kinit( ) was implemented already. What _kinit( ) does zero-initializes a heap space at 0x2000 1000 – 0x2000 4FFF as well as MCBs at 0x2000 6804 – 0x2000 6BFF. Note that 0x2000 6800 is the very first MCB that should be initialized with the max heap size, (max_size = 0x0000 4000). Thereafter, _malloc( ) calls _kalloc( ) that is considered as a "k"ernel function.

_kalloc( ) checks if size is smaller than 32 bytes (which is our minimum memory allocation unit). If so, it adjusts the size to 32. Thereafter _kalloc( ) calls _ralloc( ) which is a recursive function.

_ralloc( ) originally receives a requested memory size as well as the first mcb, (i.e., mcp_top) and the very last mcb, (i.e., mcb_bot). Upon being called, it checks if a given size <= a half of the actual memory size this _ralloc( ) is looking at. If so, _ralloc( ) should recursively call _ralloc(size, mcb_top, middle mcb between mcp_top and mcb_bot) to check the left half. If it returns NULL, (which couldn't find an available space), it calls _ralloc(size, middle mcb between mcb_top and mcb_bot, mcb_bot) to check the right half. This is a divide and conquer such as binary search and merge sort. Let's make calculations much simpler.

At the top of _ralloc( ), you'd like to define some variables:

```
void *_ralloc( int size, int left, int right ) {
  // printf( "_ralloc: size=%d, left=%x, right=%x\n", size, left, right );
  // initial parameter computation
  int entire = right - left  + mcb_ent_sz; // the current MCB space to look at
  int half = entire / 2;                   // a half of the current MCB space
  int midpoint = left + half;              // midpoint is an MCB in the middle of left and right
  int heap_addr = NULL;                    // this is an address to return to a user
  int act_entire_size = entire * 16;      // the actual heap space of the current MCB space covers
  int act_half_size = half * 16;          // a half of the actual heap space
```

You may use these parameters in your submission.

So, if (size <= act_half_size), you'll call
void* heap_addr = _ralloc(size, left, midpoint – mcb_en_size);
If heap_addr is NULL, you'll call
_ralloc(size, midpoint, right);

What if (size > act_half_size)? The entire space that the current ralloc( ) is maintaining should be allocated.

Check the left MCB's bit #0. If it's not 0, the space is used. Return NULL! If it's 0, the space is not used. Then, you should update this left MCB with
actual_entire_size | 0x01
Now, this left MCB indicates the actual entire size in use.

How to compute the actual heap address from this left MCB?
heap_top + (left – mcb_top) * 16
Why 16? This is because each MCB needs 2 bytes whereas each MCB manages 32-byte actual heap space.

Hope this helps you complete _ralloc( ) in heap.c.

Go onto the next page to see how _free( ), _kfree( ) and _rfree( ) should work.

_free( ) simply calls _kfree( ) that is considered as a "k"ernel function. It simply relays its "void *ptr" to _kfree( ).

_kfree( ) validates a user-specified address, (i.e., *ptr) is in the range between 0x2000 0000 and 0x2000 8000. If not, it simply returns NULL as the address is invalid. Otherwise, _kfree( ) computes the address of mcb that controls the user-specified memory space (which will be freed) with:

```
int mcb_addr =  mcb_top + ( addr - heap_top ) / 16;
```

Finally, _kfree( ) calls _rfree(mcb_addr) to mark it available. Your task for the midpoint report is to implement this _rallow( ) which is a recursive function.

_rfree( ) originally receives the mcb address, (i.e., mcb_addr) that manages a user-given memory space. Upon being called, it checks if this mcb is the left side or the right size of its buddy mcb. For this purpose, _rfree( ) computes the mcb index from the mcb address. If it's an even number, the mcb is on the left. If it's an odd number, it's on the right. Then, it computes the address of its buddy mcb. If its buddy is not used, it should get united with this buddy. The united mcb's size becomes double. Then, _rfree( ) recursively calls itself with _rfree(mcb_addr) to seek for more chances to claim a larger, available mcb.

At the top of _rfee( ), you'd like to define some variables:

```
int _rfree( int mcb_addr ) {
  // printf( "_rfree( %x ): mcb[%x] = %x (%d in dec)\n", mcb_addr,
  //         mcb_addr, *(short *)&array[ m2a( mcb_addr ) ],
  //         *(short *)&array[ m2a( mcb_addr ) ] );

  short mcb_contents = *(short *)&array[ m2a( mcb_addr ) ]; // this mcb's contents in 2 bytes
  int mcb_offset = mcb_addr - mcb_top; // offset from mcb[0]
  int mcb_chunk = ( mcb_contents /= 16 ); // the mcb space which this mcb is covering
  int my_size = ( mcb_contents *= 16 ); // extract the memory size only with used bit cleared
```

You may use these parameters in your submission.

First, you need to clear this mcb's used bit with

```
*(short*)&array[ m2a( mcb_addr ) ] = mcb_contents; // you must cast array to a short!
```

Then, check if this mcb is on the left or the right. How?
If (mcb_offset / mcb_chunk) % 2 is 0, it's on the left.
Otherwise it's on the right.

Since the left and right logics are duel, let's focus on the left logic.

- Find my mcb buddy, (i.e., mcb_buddy) with mcb_addr + mcb_chuck.
- Validate it to see if it's not beyond mcb_bot.

- If it's valid, check if this mcb_buddy is in use. If not used, this mcb_buddy should get united.
- mcb_buddy must be cleared, while mcb_addr's size becomes double.

```
*(short *)&array[ m2a( mcb_addr + mcb_chunk ) ] = 0; // clear my buddy
my_size *= 2;
*(short *)&array[ m2a( mcb_addr ) ] = my_size; // merge my budyy
```

- Call _rfree(mcb_addr);

The right logic is pretty much similar. Think by yourself.

Hope this helps you complete _rfree( ) in heap.c.