ACUBESAT

Memory Analysis Report
8/11/2018
Konstantinos Kanavouras

1 Summary

This report describes the allocation of **read-only** and **read-write** memory on FreeRTOS applications, as well as methods for **measuring** and **analysing** it.

On an embedded system, knowing the ins and outs of memory management is critical, due to energy, spacing and cost constraints that reduce the amount of available memory on-board. On a Personal Computer, processes that use 100MB or more RAM space are not unheard of, but a microcontroller with 192kB of available RAM needs special care on the definition and usage of variables. The basic points of this report are very important to know when developing error-prone embedded systems.

This report centres itself on ARM Cortex-M3 microcontrollers, and more specifically on the STM32 variety. It makes heavy use of hexadecimal notation.

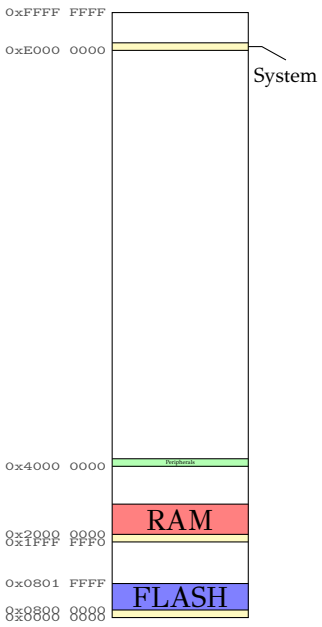


Figure 1: Memory map of STM32F103C8

2 Memory Allocation

2.1 Microcontroller architecture

ARM Cortex-M processors are based on a **32-bit** architecture, so they support up to 2^{32} RAM addresses, or about 4GB of theoretical maximum memory. However, as can be seen on Figure 1, most of this space is not used, since the microcontroller doesn't contain enough physical memory to correspond with these values.

Accessing memory on Cortex-M is straightforward: The value `0x2000 0000` corresponds to *the first word in the RAM*, the value `0x2000 0004` corresponds to *the second word in the RAM*, and so on. To instantly access this data in C, you need a *pointer* to the correct address: `*(0x2000000)`.

Note that the data in the memory is typically **aligned** in words. A **word** is equal to 4 bytes, or 32 bits. This is due to the fact that memory access instructions of the processor work with 32-bits of data each time, owing to the 32-bit architecture.

2.2 FLASH & RAM

The microcontrollers we are using contain two types of memory:

FLASH memory FLASH is the **"read-only"** memory (ROM) of our application. It is **non-volatile**, meaning the data stays safe even after a power off. It corresponds to the hard-drive of a general purpose machine.

FLASH is where the **executable program** of our microcontroller is stored. It can be easily read in C through a pointer, but it cannot easily be written to, unless an external programmer or the FLASH peripheral is used.

An STM32F103C8T6 chip contains 64kB of FLASH memory, located between `0x0800 0000` — `0x0800 FFFF`.

RAM Random Access Memory (RAM) is the **read & write** memory of our application. It is **volatile**, meaning its data will be removed after a microcontroller reset, but it is much faster to read and write than FLASH.

RAM is where all the **variables** of the running program are stored. Marking a non-constant variable with the `volatile` keyword on C makes sure that it will be stored on the RAM, and won't be optimized away as a register.

An STM32F103C8T6 chip contains 20kB of RAM memory, located between `0x2000 0000` — `0x2000 4FFFF`.

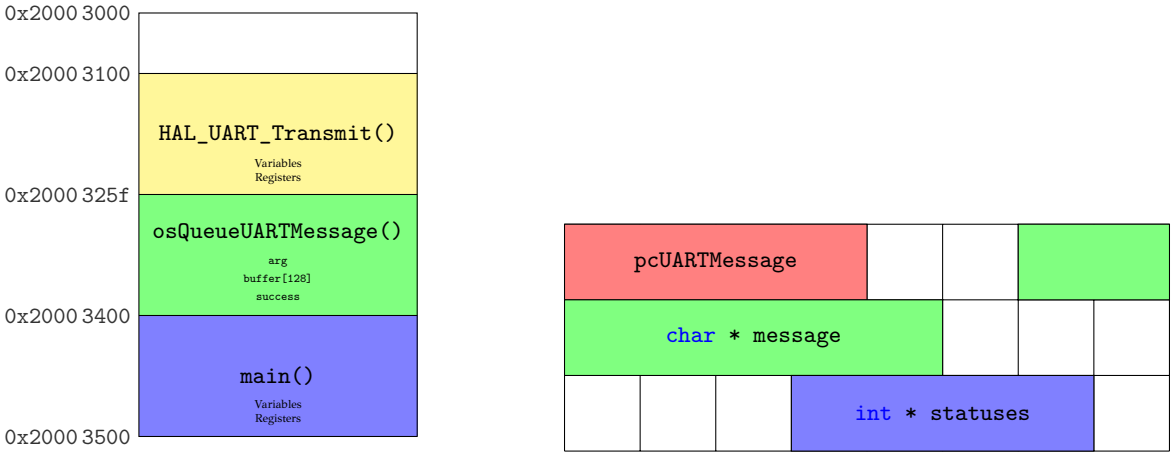
2.3 RAM utilisation: The stack & the heap

A C program contains quite a lot of memory definitions, many of which are done in different ways. The location of each allocation, and whether its size is known at compile time or not, are closely related to how that object is stored on the memory.

2.3.1 Global variables

Perhaps the easiest variables to store are the ones that have a **global** scope and are virtually accessible from any part of our program:

```
char cDMA_RX_Buffer[10] = {"\0"};
char cDMA_TX_Buffer[4] = { 't' , 'e', 's', 't' };
char currentStatus[] = "RUNNING";
int currentNumberOfFireworks = 5;
```



(a) Visualisation of the stack (b) Visualisation of the heap

These are stored directly to the memory, and this allocation is fully known at compile-time. These variables have constant memory addresses. This is called **static allocation** of memory.

2.3.2 Local variables: The Stack

Variables inside functions are said to have **local** scope and are accessible only from within the function:

```
void osQueueUARTMessage(const char * format, ...) {
  va_list arg;
  char buffer[128];
  uint8_t success = 0;
  // ...
}
```

These variables need to be **allocated every time the function is called**, and **freed every time we get out of the function.**[1]

As such, all these are stored in a **stack**. Each task/process has its own stack that gets filled whenever a function is called, and downsized when a function returns.

The size of the variables stored in the stack should be known at compile-time. However, the stack is allocated on the memory during the run-time. As a result, it's quite possible to run *out of memory* when there is no more stack space for a function call. This is known as a **stack overflow**.

Note that on ARM Cortex-M processor, the stack is typically increased **downwards**[2] (see Figure 2a). This means that the stack of a task might begin at `0x2000 4efc` and end at `0x2000 3948`.

2.3.3 `malloc()`: The Heap

Sometimes we need to allocate variables without knowing their size beforehand (for example, when the user provides the number of items to store in a list). In C, this is done through the `malloc()` (or, in FreeRTOS calls, `vPortMalloc()`) calls. This is called **dynamic memory allocation**.

```
void vUARTTask(void* pvParameters) {
  UARTMessage_t pcUARTMessage = pvPortMalloc(size);
  char * message = pvPortMalloc(receivedBytes);
  int * statuses = pvPortMalloc(num_tasks);
}
```

Variables allocated with `malloc()` are stored in a structure called **heap**, which contains memory data in various positions, and is managed by an internal logic of the operating system. Variables stored in the heap need to be freed by the programmer, otherwise they will remain allocated forever, something called a **memory leak**. When the memory is full, calls to **malloc()** will fail due to an **out-of-memory** error.

2.3.4 Constant variables

Variables defined as `const`ant, large values in `#define`s, and literals are not typically stored on the RAM, but they are typically stored on the read-only **FLASH** and accessed by the software whenever needed.

```
#define BUFFER_SIZE 128
const unsigned int Enable_Satellite 1
const unsigned int Enable_Transmission 0
char message[] = "This is a string literal";
```

2.3.5 Summary

All the above can be summarised in this table:

What?	Where?
Global variables	Statically
Local variables	Stack
`malloc()`	Heap
`const`	FLASH

Further reading: What and where are the stack and heap?

2.4 Memory sections

Further reading: What resides in the different memory types of a microcontroller?
Further reading: text, data and bss: Code and Data Size Explained

Further reading: Wikipedia — Data Segment
Further reading: C++ links: executable and object file formats...

The compiled program is stored in an **executable** file, such as `FreertosMockup.elf`, which is an ELF file. These files contain all the **binary data** of the program, separated in sections. Each section has a different role and is stored in a different way in some place of the memory. Usual section names are `.text`, `.rodata`, `.data` and `.bss`.

2.4.1 `.text`

The `.text` section contains the **executable** code of the program. It includes all the instructions that the processor will run in machine code, and is probably the most important section of the application.

To reduce the size of `.text`, the programmer just needs to include less code in the program. This can be done by simplifying existing functions, not using bloated libraries, or by enabling the compiler *optimization* and linker *dead code removal* options.

`.text` is only stored on **FLASH** memory.

2.4.2 `.data`

The `.data` section contains all the **global variables** of the program. This includes all the data that is **statically allocated** at compile-time, and the exact structure of the entire `.data` section is stored in the executable file. The advantage of storing variables on `.data` instead of making them local is that they can be measured in an exact way before running the program, and there are no risks of overflows. However, using global variables is often considered a bad programming practice.

To reduce the size of `.data`, the programmer needs to minimise the amount or size of allocated variables.

`.data` is stored on **FLASH** and **RAM** memory.

2.4.3 `.bss`

The `.bss` section also contains **global variables** (and static function variables) of the program, but only those that are **initialized to zero**. As such, they don't take up any data space on the FLASH and can only be allocated in the RAM. For example, a `buffer[128]` will be initialised in `.bss`, since it contains 0 data.

To reduce the size of `.bss`, the programmer needs to minimise the amount or size of allocated variables.

`.bss` is only stored on **RAM** memory.

Further reading: Wikipedia — `.bss`

```
00* 0x08000000 |-----------#########-------------------------------------------------------------------------------------| 0x0800a348 40.8K mrwx  LOAD0
01  0x20000000 |--------------------------#-------------------------------------------------------------------------------| 0x20003bb4  1.6K mrw-  LOAD1
02  0x20003bb4 |--------------------------#-------------------------------------------------------------------------------| 0x20004304     0 mrw-  LOAD2
03  0x08000000 |-----------#-------------------------------------------------------------------------------------------| 0x0800010c   268 r--   .isr_vector
04* 0x08000110 |-----------#########-------------------------------------------------------------------------------------| 0x08009acc 38.4K r-x   .text
05  0x08009ad0 |--------------------#------------------------------------------------------------------------------------| 0x0800a338  2.1K r--   .rodata
06  0x0800a338 |--------------------#------------------------------------------------------------------------------------| 0x0800a340     8 r--   .ARM
07  0x0800a340 |--------------------#------------------------------------------------------------------------------------| 0x0800a344     4 rw-   .init_array
08  0x0800a344 |--------------------#------------------------------------------------------------------------------------| 0x0800a348     4 rw-   .fini_array
09  0x20000000 |--------------------------#-------------------------------------------------------------------------------| 0x20000680  1.6K rw-   .data
10  0x20000680 |--------------------------#-------------------------------------------------------------------------------| 0x20003bb4     0 rw-   .bss
11  0x20003bb4 |--------------------------#-------------------------------------------------------------------------------| 0x20004304     0 rw-   ._user_heap_stack
12  0x00000000 |-------------------------#--------------------------------------------------------------------------------| 0x00000029    41 ----  .ARM.attributes
13  0x00000000 |-------------------------##############################-------------------------------------------------| 0x00024ae0 146.7K ----  .debug_info
14  0x00000000 |-----------------------------------------------------######-------------------------------------------| 0x0000510b 20.3K ----  .debug_abbrev
15  0x00000000 |-----------------------------------------------------##########---------------------------------------| 0x0000bdbb 47.4K ----  .debug_loc
16  0x00000000 |-----------------------------------------------------#------------------------------------------------| 0x00000dd0  3.5K ----  .debug_aranges
17  0x00000000 |-----------------------------------------------------###----------------------------------------------| 0x000017f8  6.0K ----  .debug_ranges
18  0x00000000 |-----------------------------------------------------########-------------------------------------------| 0x00009872 38.1K ----  .debug_line
19  0x00000000 |-----------------------------------------------------######---------------------------------------------| 0x00005d34 23.3K ----  .debug_str
20  0x00000000 |-----------------------------------------------------#------------------------------------------------| 0x0000007c   124 ----  .comment
21  0x00000000 |-----------------------------------------------------####-------------------------------------------------| 0x00003714 13.8K ----  .debug_frame
22  0x00000000 |-----------------------------------------------------#####----------------------------------------------| 0x000045d0 17.5K ----  .symtab
23  0x00000000 |-----------------------------------------------------##-------------------------------------------------| 0x00001ec0  7.7K ----  .strtab
24  0x00000000 |-----------------------------------------------------#--------------------------------------------------| 0x000000ee   238 ----  .shstrtab
28  0x08000000 |#-------------------------------------------------------------------------------------------------------| 0x08000034    52 mrw-  ehdr
=>  0x08004a60 |-------------------------------------------------------------------------------------------------------| 0x08004a5f
```

Figure 3: List of the sections of the compiled file [3]

```
[0x08000110 0% 195 FreertosMockup.elf]> pd $r @ loc._t_2
            ;-- section..text:
            ;-- $t:
/ (fcn) sym.__do_global_dtors_aux 24
|   sym.__do_global_dtors_aux ();
|           0x08000110  ~   10b5           push {r4, lr}               ; [02] --r-x section size 39356 named .text section 2 va=0x08000110 pa=0x00010110 sz=39356 vsz=39
|           ;-- entry2.fini:
|              ; UNKNOWN XREF from 0x0800a344 (section_end..init_array + 0)
|           0x08000111      b5             unaligned
|           0x08000112      054c           ldr r4, loc._d_1            ; [0x8000128:4]=0x20000680 loc._edata
|           0x08000114      2378           ldrb r3, [r4]
|       ,=< 0x08000116      33b9           cbnz r3, 0x8000126          ;[1]
|       |   0x08000118      044b           ldr r3, [0x0800012c]        ; [0x800012c:4]=0
|      ,==< 0x0800011a      13b1           cbz r3, 0x8000122           ;[2]
|      ||   0x0800011c      0448           ldr r0, obj.__EH_FRAME_BEGIN    ; [0x8000130:4]=0x8009ab4 sym._init
|      ||   0x0800011e      aff30080       nop.w
|      ||      ; JMP XREF from 0x0800011a (sym.__do_global_dtors_aux)
|      `--> 0x08000122      0123           movs r3, 1                  ; 1
|       |   0x08000124      2370           strb r3, [r4]
|       |      ; JMP XREF from 0x08000116 (sym.__do_global_dtors_aux)
\       `-> 0x08000126      10bd           pop {r4, pc}
            ;-- $d:
               ; DATA XREF from 0x08000112 (sym.__do_global_dtors_aux)
            0x08000128      .dword 0x20000680 ; section_end..data ; section..bss ; loc.completed.8672 ; loc._d_5 ; loc._sbss ; loc.__bss_start ; loc._edata
               ; DATA XREF from 0x08000118 (sym.__do_global_dtors_aux)
            0x0800012c      0000           movs r0, r0
            0x0800012e      0000           movs r0, r0
               ; DATA XREF from 0x0800011c (sym.__do_global_dtors_aux)
            0x08000130      .dword 0x08009ab4 ; obj.__EH_FRAME_BEGIN ; loc._t_237 ; sym._init
            ;-- $t:
/ (fcn) sym.frame_dummy 16
|   sym.frame_dummy ();
|           0x08000134  ~   08b5           push {r3, lr}
|           ;-- entry1.init:
|              ; UNKNOWN XREF from 0x0800a340 (section_end..ARM + 0)
|           0x08000135      b5             unaligned
|           0x08000136      034b           ldr r3, [loc._d_3]          ; [0x8000144:4]=0
|       ,=< 0x08000138      1bb1           cbz r3, 0x8000142           ;[3]
|       |   0x0800013a      0349           ldr r1, loc.object.8677     ; [0x8000148:4]=0x20000684 loc.object.8677
```

Figure 4: Part of the disassembled `.text` section of the executable [4]

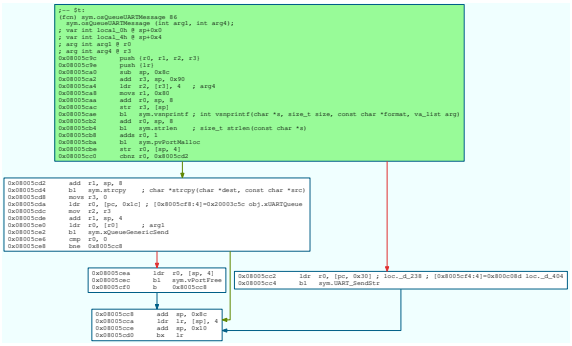


Figure 5: Call graph of `osQueueUARTMessage` [5]

2.4.4 `.rodata`

The **`.rodata`** section contains all the **constants** of the program. That includes `const` variables, `#define` literals, and other literals, that have not been optimized away by the compiler. For example, a list of 1024 different numbers (e.g. representing different legal transmission frequencies) would be stored on `.rodata`.

`.rodata` is typically only stored on **FLASH** memory.

2.4.5 `._user_heap_stack`

The **`._user_heap_stack`** section makes sure that there is enough space for the **stack** of all function calls, and the **heap** of classic `malloc()` calls. It should have a sufficient size, since that's where all the dynamic allocation of a program will take place.

Note that the heap and stack used by FreeRTOS are *not* using the `._user_heap_stack` section. As such, it can be kept to a minimal size, enough to support just the `main()` function and the FreeRTOS scheduler.

`._user_heap_stack` is only stored on **RAM** memory.

2.4.6 Further sections

The programmer might discover that there are other sections stored in the executable file, but the amount of memory they take up is negligible, and they are usually required for the MCU to operate.

Examples might be the `.isr_vector` needed to point out the location of ISRs, or debug sections containing necessary debugging information.

The initialisation of all sections is done by the **linker** and is configured by a **linker script**. In our case, that script is `STM32F103C8_FLASH.ld`. It stores the locations and memory sizes of each available memory type, as well as where each section will be stored, and what it will contain. Each MCU type comes with a different linker script.

2.5 Memory allocation in FreeRTOS

The stack and the heap are also used in a **FreeRTOS** application, but, in order for the Operating System to have full control of the tasks, their stack and heap is allocated explicitly in FreeRTOS.

2.5.1 The FreeRTOS heap

Reference: FreeRTOS — Memory Management

The FreeRTOS **heap** takes the form of a global variable stored in the `.bss` section:

```
#define configTOTAL_HEAP_SIZE      ( ( size_t ) ( 12 * 1024 ) )
```

```
20000000 l    d  .data  00000000 .data
20000000 l     O .data  00000004 uxCriticalNesting
20000010 l     O .data  000000f0 impure_data
20000008 g     O .data  00000004 SystemCoreClock
20000000 g       .data  00000000 _sdata
2000067c g     O .data  00000001 __fdlib_version
20000004 g     O .data  00000001 blinkingEnabled
20000508 g     O .data  00000004 __malloc_sbrk_base
20000100 g     O .data  00000408 __malloc_av_
2000000c g     O .data  00000004 _impure_ptr
20000680 g       .data  00000000 _edata
2000050c g     O .data  00000004 __malloc_trim_threshold
20000510 g     O .data  0000016c __global_locale
```

(a) Variables in the `.data` section[6]

```
0800bc10 l    d  .rodata        00000000 .rodata
0800bc30 l     O .rodata        00000007 CHANNEL_OFFSET_TAB
0800bca8 l     O .rodata        00000007 nRF24_ADDR_REGS
0800bcaf l     O .rodata        00000006 nRF24_RX_PW_PIPE
0800bf5b l     O .rodata        00000005 nRF24_ADDR_Rx.10134
0800bf60 l     O .rodata        00000005 nRF24_ADDR_Tx.10133
0800c263 l     O .rodata        00000010 blanks.7236
0800c273 l     O .rodata        00000010 zeroes.7237
0800c4c8 l     O .rodata        00000080 npio2_hw
0800c548 l     O .rodata        00000108 two_over_pi
0800c650 l     O .rodata        00000040 PIo2
0800c221 g     O .rodata        00000008 APBPrescTable
0800c3c0 g     O .rodata        000000c8 __mprec_tens
0800c398 g     O .rodata        00000028 __mprec_bigtens
0800c283 g     O .rodata        00000101 _ctype_
```

(b) Variables in the `.rodata` section[7]

Figure 6

.bss	0x20000688	13,5 KB
ucHeap	0x200006bc	12 KB
cDMA_RX_Buffer	0x2000381c	550 B
xGPSData	0x20003af0	160 B
pxReadyTasksLists	0x200036dc	100 B
xSensorData	0x20003bb8	80 B
dma	0x20003c18	68 B
tim	0x200037dc	64 B
__malloc_current_mallinfo	0x20003a64	40 B
dma_usart_rx	0x20003aa0	40 B
dma_usart_tx	0x20003ac8	40 B
nRF24_payload	0x20003a42	32 B
xDelayedTaskList1	0x2000375c	20 B
xDelayedTaskList2	0x20003770	20 B
xPendingReadyList	0x20003790	20 B
xSuspendedTaskList	0x200037a8	20 B

(a) Variables in the `.bss` section[8]

```
/* Highest address of the user mode stack */
_estack = 0x20005000;    /* end of RAM */
/* Generate a link error if heap and stack
   don't fit into RAM */
_Min_Heap_Size = 0x300;      /* required amount
   of heap  */
_Min_Stack_Size = 0x450; /* required amount of
   stack */

/* Specify the memory areas */
MEMORY
{
RAM (xrw)      : ORIGIN = 0x20000000, LENGTH =
   20K
FLASH (rx)     : ORIGIN = 0x08000000, LENGTH =
   64K
}
```

(b) Part of the configuration in the linker script

Figure 7

```
static uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
```

The size of the heap is always specified in **bytes** and can be changed in `FreeRTOSConfig.h`. This is essentially where all the data of FreeRTOS tasks is stored.

To explicitly allocate space in the heap, the programmer must call `pvPortMalloc()` and `pvPortFree()`. Note that the classic `malloc()` and `free()` **should not** be used, as they allocate space on the minimal remaining RAM space, and may not be thread-safe.

FreeRTOS provides algorithms that efficiently manage the heap. `heap_4.c` is a good choice for beginners, while `heap_1.c` is the fastest option, but it doesn't allow calling `pvPortFree()`.

2.5.2 The FreeRTOS stack

Each **task** in FreeRTOS has **its own stack**, where local variables of all its functions are stored. The size of this stack **must be known** beforehand[9]. For example:

```
xTaskCreate(vUARTTask, "UART", 300, NULL, 3, NULL);
xTaskCreate(vRefreshWWDGTask, "RefreshWWDG", 100, NULL, 6, NULL);
xTaskCreate(vBlinkyTask, "Blinking", 70, NULL, 3, NULL);
```

In these tasks, 300, 100 and 70 is the size of the corresponding stack in bytes. This size should be equal or larger than the maximum amount of stack that the task will ever require. Since that can't be accurately known beforehand, call graph analysis (which we'll describe below) or good guesswork are needed to specify the stack size.

An incorrect stack size may cause **stack overflow** which can result in nasty, hard-to-detect bugs. To prevent this, the user can query `uxTaskGetStackHighWaterMark()`, or use stack overflow checking.

Internally, FreeRTOS calls `vPortMalloc()` to initialise the tasks. So, in a sense, these stacks live in the heap which lives in `.bss`!

Further reading: FreeRTOS — How big should the stack be?

2.5.3 Static memory allocation

FreeRTOS also provides the option of allocating all the tasks and resources directly in `.bss` instead of its heap. In that way, the memory usage of the application is easier to analyse at compile-time.

```
#define STACK_SIZE 200 // This is in words, not bytes
StaticTask_t xTaskBuffer;
StackType_t xStack[ STACK_SIZE ];

xTaskCreateStatic(vTaskCode, "NAME", STACK_SIZE, null, 3, xStack, &xTaskBuffer);
```

Further reading: FreeRTOS — Static Vs Dynamic Memory Allocation

2.6 Memory protection and memory management

Many modern MCUs provide capabilities for memory protection and memory management. These units might make memory analysis a bit trickier, but will not be used, as they do not provide many advantages for our mission.

3 Memory analysis

Further reading: Analyzing the Linker Map file with a little help from the ELF and the DWARF

This section introduces the tools that the programmer needs in order to analyse how much memory is used by each section, and which functions or variables make the most use out of it and can be optimised.

Many of the tools used here are part of the **GNU development tools** suite, which includes compilers, linkers, disassemblers etc. In the ARM variety, these tools get the `arm-none-eabi-` prefix. For example, instead of calling `gcc`, one should call `arm-none-eabi-gcc`[10].

The most useful tool for this analysis can perhaps be **Atollic's build & stack analyzer**. However, we will also present other tools that can fetch detailed memory usage data.

In order to get this data, we will need access to the **executable binary file** of our application. The entire build output of the app resides in the `Debug` folder, and the executable is stored under the `.elf` extension (Executable and Linkable Format (ELF)). The `.elf` is the final file generated by the *linker*. There is further information in the `.map` file, which contains every symbol's size and location in memory, although it's not practical to read for humans.

3.1 `size`

The `size` command can be used to obtain a very rough understanding of the memory allocation of the entire program:

```
$ arm-none-eabi-size ./FreertosMockup.elf
   text    data     bss     dec     hex filename
  50864    1676   15696   68236   10a8c FreertosMockup.elf
```

Note that the above names do not correspond to sections (as seen in subsection 2.4). `text` refers to all the data only in the FLASH[11], `bss` for the data only in the RAM, and `data` for both. `dec` is the sum of all these three, and `hex` is that sum in hexadecimal format.

To calculate the amount of memory spent on each peripheral, use the following

formulas:

$$\text{FLASH} = \text{text} + \text{data}$$
$$\text{RAM} = \text{bss} + \text{data}$$

3.2 nm

GNU's `nm` provides a list of all **symbols and their sizes** stored in the executable. Note that only global objects are stored, while stack variables will not be shown.

```
$ arm-none-eabi-nm -CS --size-sort ./FreertosMockup.elf
 # ...
0800017c 0000027a T __subdf3
08009ce0 000002c8 T pow
08001060 000002dc T __udivmoddi4
08009968 00000304 T _realloc_r
080050a0 0000039c T cGetGPSData
0800a990 000003e8 T __ieee754_rem_pio2
20000270 00000408 D __malloc_av_
080061d8 0000040c T minmea_scan
08008bac 00000418 T _malloc_r
0800b08c 000006a8 T __kernel_rem_pio2
08009fa8 000009e4 T __ieee754_pow
08007fd8 00000ba8 T _dtoa_r
08006e8c 00000fa0 T _svfprintf_r
200006bc 00003000 b ucHeap
```

In the above example, we can see that the `ucHeap` variable, located in the RAM (starting from `0x2000000`) takes up the most space, with $(3000)_{16} = (12288)_{10}$ bytes. [12] We can also see that *floating point* functions take up a lot of space, so the program can be optimised by working only with integers.

3.3 readelf

GNU's `readelf` is a tool that extends the functionality of `nm`. It can be used to list all sections in the `.elf` file:

```
$ arm-none-eabi-readelf -S ./FreertosMockup.elf
There are 23 section headers, starting at offset 0x86b74:

Section Headers:
[Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
[ 0]                   NULL            00000000 000000 000000 00      0   0  0
[ 1] .isr_vector       PROGBITS        08000000 010000 00010c 00   A  0   0  1
[ 2] .text             PROGBITS        08000110 010110 00bafc 00  AX  0   0 16
[ 3] .rodata           PROGBITS        0800bc10 01bc10 000aa0 00   A  0   0  8
[ 4] .ARM              ARM_EXIDX       0800c6b0 01c6b0 000008 00  AL  2   0  4
[ 5] .init_array       INIT_ARRAY      0800c6b8 01c6b8 000004 04  WA  0   0  4
[ 6] .fini_array       FINI_ARRAY      0800c6bc 01c6bc 000004 04  WA  0   0  4
[ 7] .data             PROGBITS        20000000 020000 000684 00  WA  0   0  4
[ 8] .bss              NOBITS          20000688 020688 003600 00  WA  0   0  8
[ 9] ._user_heap_stack NOBITS          20003c88 023c88 000750 00  WA  0   0  1
[10] .ARM.attributes   ARM_ATTRIBUTES  00000000 020684 000029 00      0   0  1
[11] .debug_info       PROGBITS        00000000 0206ad 030367 00      0   0  1
[12] .debug_abbrev     PROGBITS        00000000 050a14 006d0d 00      0   0  1
[13] .debug_loc        PROGBITS        00000000 057721 00d505 00      0   0  1
[14] .debug_aranges    PROGBITS        00000000 064c28 001200 00      0   0  8
[15] .debug_ranges     PROGBITS        00000000 065e28 001cd0 00      0   0  8
[16] .debug_line       PROGBITS        00000000 067af8 00bef0 00      0   0  1
[17] .debug_str        PROGBITS        00000000 0739e8 006d75 01  MS  0   0  1
[18] .comment          PROGBITS        00000000 07a75d 00007c 01  MS  0   0  1
[19] .debug_frame      PROGBITS        00000000 07a7dc 0042c8 00      0   0  4
[20] .symtab           SYMTAB          00000000 07eaa4 005970 10     21 993  4
```

```
[21] .strtab           STRTAB          00000000 084414 002670 00      0   0  1
[22] .shstrtab         STRTAB          00000000 086a84 0000ee 00      0   0  1
```

It can also be used to list symbol sizes, bearing the same results as subsection 3.2: [13]

```
$ readelf -Ws ./FreertosMockup.elf | sort -n -k3
 # ...
1254: 20000270  1032 OBJECT  GLOBAL DEFAULT    7 __malloc_av_
1014: 080061d9  1036 FUNC    GLOBAL DEFAULT    2 minmea_scan
1288: 08008bad  1048 FUNC    GLOBAL DEFAULT    2 _malloc_r
1162: 0800b08d  1704 FUNC    GLOBAL DEFAULT    2 __kernel_rem_pio2
1235: 08009fa9  2532 FUNC    GLOBAL DEFAULT    2 __ieee754_pow
1287: 08007fd9  2984 FUNC    GLOBAL DEFAULT    2 _dtoa_r
1078: 08006e8d  4000 FUNC    GLOBAL DEFAULT    2 _svfprintf_r
 247: 200006bc 12288 OBJECT  LOCAL  DEFAULT    8 ucHeap
```

3.4 `objdump`

GNU's `objdump` is a disassembly tool. By calling `arm-none-eabi-objdump -S ./FreertosMockup.elf`, one can see the full assembly code of the application. The `-t` parameter can be used to list symbol sizes of the `.text` section only, and the `--dwarf`[14] parameter prints detailed debugging info, including sizes of the local function stacks.

3.5 `linkermapviz`

Source Code: https://github.com/PromyLOPh/linkermapviz[15]

`linkermapviz` is a tiny yet powerful tool that shows a visualisation of the memory consumption of all the sections in the executable file. It provides a nice overview of the allocation of memory while categorising the symbols based on their files and showing the largest consumers.

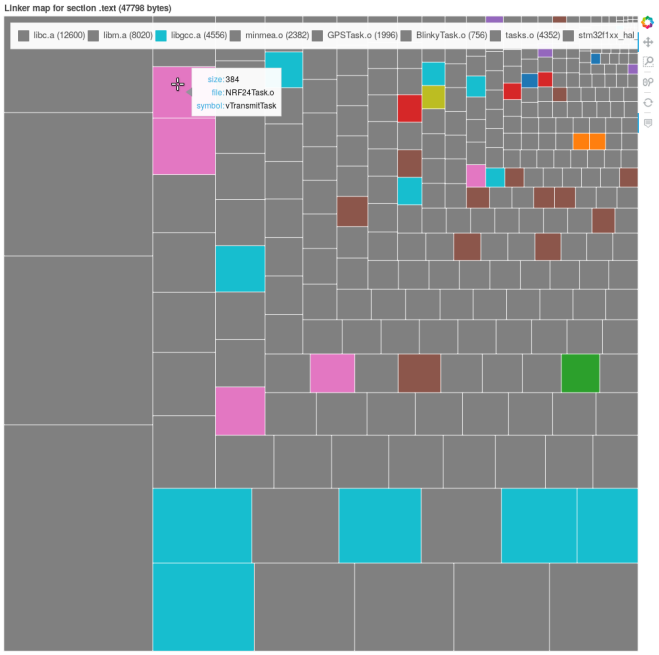


Figure 8: Visualisation of the `.text` section of the freertos-mockup

Such an overview of allocated memory will be useful for a better analysis and presentation of our embedded software.

3.6 radare2 (`r2`)

radare2 along with its GUI, Cutter, is a very powerful disassembly and reverse engineering tool for binaries. While its features aren't useful for us in this context, they can be valuable for some advanced debugging situations.

3.7 `.su` files

Further reading: GNU Static Stack Usage Analysis

The `gcc` compiler offers an `-fstack-usage`, which generates `.su` files for every source file in the application. These files contain invaluable information about the **size** of the **local stack of each function** in our program:

```
$ cat `find -iname '*.su'` | sort -n -k2 | column -t
# ...
uart.c:128:6:UART_SendBufHexFancy        56   dynamic
# ...
minmea.c:411:6:minmea_parse_gga          80   static
minmea.c:439:6:minmea_parse_gsa          80   static
TraceTask.c:10:6:vSetupTrace             80   static
minmea.c:516:6:minmea_parse_gsv          88   static
uart.c:6:6:UART_Init                     104  static
GPSTask.c:195:8:cGetGPSData              112  static
BlinkyTask.c:70:6:vSetupBlinky           136  static
GPSTask.c:38:6:vSetupGPS                 136  static
```

```
UARTTask.c:43:6:osQueueUARTMessage       144  static
NRF24Task.c:198:6:vTaskInfoTransmitTask  480  static
```

In the above output, we can see that `vTaskInfoTransmitTask()` and `osQueueUARTMessage()` are the largest memory-eaters of our application, with 480 and 144 bytes of space respectively, and could perhaps be optimized to save space.

Some functions, like `UART_SendBufHexFancy()`, may involve *dynamic* memory allocation, and their stack size cannot be accurate.

Also note that the above only shows the **local** cost of the function. However, each function calls other functions, and *their* stack sizes should be included in our calculation as well! This can be done using a script such as `avstack.pl`:

```
$ perl avstack.pl `find -name '*.o' -not -name '*startup*'`
  Func                                Cost    Frame   Height
-------------------------------------------------------------------------
> vTaskInfoTransmitTask               680      484        8
> vGPSMessageRXTask                   452       12        9
  prvGPSDMAMessageRX                  440       28        8
  cGetGPSData                         412      116        7
> vGPSTask                            348       52        7
> vReceiveTask                        332       36        7
> main                                332       36        9
> vCheckTask                          316       20        7
# ...
```

Here we can see that, for example, `vGPSMessageRXTask()` is also a large consumer of memory, due to its calls to other functions.

There is also an expensive proprietary tool that performs static stack analysis.

3.8 Atollic Build Analyzer

Perhaps the easiest to use and yet most powerful tool to analyse the allocated memory is **Atollic's own build analyzer**.

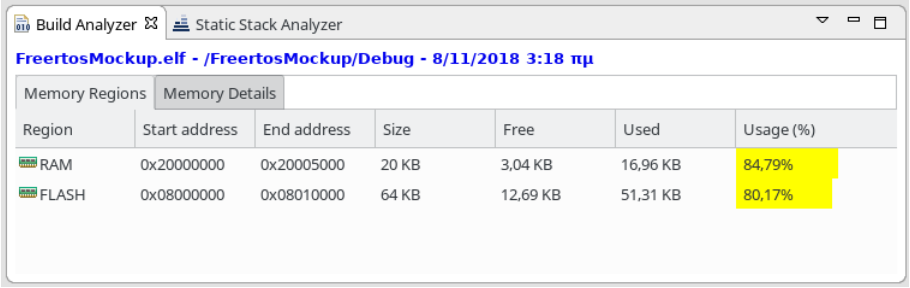
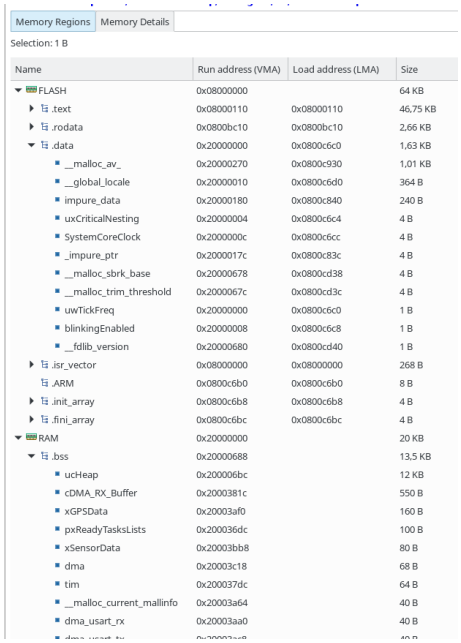


Figure 9: Memory Regions of the freertos-mockup

3.8.1 Build Analyzer > Memory Details



On the **Memory Details** tab of the analyser, we can observe the amount of memory spent by each symbol on each section of the executable. This is extremely useful as an overview of which symbols take up the largest amount of memory. This data corresponds to the exact same data received by the terminal commands

3.8.2 Static Stack Analyzer

List | Call graph

Function	Depth	Max cost	Local cost	Type	Location	Info
HardFault_Handler	?	?	0	STATIC	stm32f1xx_it.c:68	Max cost uncertain. Recursive
MemManage_Handler	?	?	0	STATIC	stm32f1xx_it.c:86	Max cost uncertain. Recursive
vTaskInfoTransmitTask	12	648	480	STATIC	NRF24Task.c:198	Max cost uncertain
vGPSMessageRXTask	16	416	8	STATIC	GPSTask.c:152	Max cost uncertain
prvGPSDMAMessageRX	15	408	24	STATIC	GPSTask.c:346	Max cost uncertain
cGetGPSData	14	384	112	STATIC	GPSTask.c:195	Max cost uncertain
osQueueUARTMessage	13	272	144	STATIC	UARTTask.c:43	Max cost uncertain
minmea_parse_gsv	4	164	88	STATIC	minmea.c:516	Max cost uncertain
minmea_parse_gsa	4	156	80	STATIC	minmea.c:439	Max cost uncertain
minmea_parse_rmc	4	156	80	STATIC	minmea.c:381	Max cost uncertain
minmea_parse_gga	4	156	80	STATIC	minmea.c:411	Max cost uncertain
minmea_parse_vtg	4	140	64	STATIC	minmea.c:555	Max cost uncertain
minmea_parse_gll	4	132	56	STATIC	minmea.c:470	Max cost uncertain
minmea_parse_zda	4	116	40	STATIC	minmea.c:589	Max cost uncertain
minmea_sentence_id	4	92	16	STATIC	minmea.c:352	Max cost uncertain
minmea_tocoord	2	16	16	STATIC	minmea.h:246	Max cost uncertain
minmea_tofloat	2	16	16	STATIC	minmea.h:235	Max cost uncertain
__aeabi_f2d	1	0	0			Max cost uncertain. No stack usage information available for this function
osQueueUARTMessage	13	272	144	STATIC	UARTTask.c:43	Max cost uncertain
strtok	6	0	0			Max cost uncertain. No stack usage information available for this function
ulTaskNotifyTake	3	52	16	STATIC	tasks.c:4479	
vGPSTask	14	320	48	STATIC	GPSTask.c:110	Max cost uncertain
Reset_Handler	17	304	0			Max cost uncertain. No stack usage information available for this function
vReceiveTask	14	304	32	STATIC	NRF24Task.c:151	Max cost uncertain
vCheckTask	14	288	16	STATIC	CheckTask.c:10	Max cost uncertain
osQueueUARTMessage	13	272	144	STATIC	UARTTask.c:43	Max cost uncertain
vTaskDelay	4	88	8	STATIC	tasks.c:1305	
xTaskGetTickCount	0	0	0	STATIC	tasks.c:2255	
vTransmitTask	12	152	24	STATIC	NRF24Task.c:100	Max cost uncertain

The **Static Stack Analyzer** tab performs the same analysis as the one done in subsection 3.7, providing us however with a much more visual way to measure our functions.

For every function, we can see its **local cost** (i.e. how much the variables allocated inside it take up), and its **max cost** (includes calls to other functions). For functions with unknown memory constraints or recursive calls, the data may be inaccurate, since there is no way to predict the required memory beforehand.

The main aspect of the **call graph** in the stack analyzer is that every function calls other functions, which themselves call other functions, and so on. As such, the stack needs to hold the data for a whole bunch of nested functions.

In the above example, we can see that `vGPSMessageRXTask()` is expensive, mainly due to its calls to `osQueueUARTMessage` and `cGetGPSData`.

3.9 A case study: Memory usage optimisation on freertos-mockup

During development of the mockup, a lot of RAM and FLASH memory issues were brought up. These issues were fixed with some guessing and fine-tuning of the corresponding variables. However, in a more mission-critical environment, it is important to know how the memory operates and how space can be utilised more efficiently (pun intented). With that in mind, we will try to analyse and improve the memory utilisation of the current revision of the mockup.

As shown in Figure 9, the FLASH memory is almost getting full. While the RAM seems to be quite used as well, most of it corresponds to the FreeRTOS heap. One of our goals is to enlarge this heap as much as possible, so that tasks and `vPortMalloc()` calls have a lot of leeway and do not run out of memory.

A quick analysis of the `.text` section shows quite a few C library functions, such as `_malloc_r()` and `__ieee754_rem_pio2`. The `malloc()` variety should not appear, since we are only using FreeRTOS' allocation function. However, they are necessary for C standard library functions, such as `printf()`. Ideally, these functions would be reimplemented to use FreeRTOS' ones. On the other hand, the floating functions are necessary, since we are using a software implementation of floating point arithmetic, something that shows the advantages of a separate Memory Protection Unit (MPU), which would reduce the memory usage and improve the performance of our program.

Since the stack and heap are not used for anything other than `main()` and the occasional `printf()` call, we can safely reduce the minimum heap and stack sizes, so that we can increase the FreeRTOS heap size.

Also, since `osQueueUARTMessage()` is used often, we can reduce the buffer size to a more reasonable value of 84 characters, saving 40 bytes per call. We can also reduce the stack sizes of some tasks, since the call graphs show that they are not consuming large amounts of memory.

The final list of changes is:

```
diff --git a/Inc/FreeRTOSConfig.h b/Inc/FreeRTOSConfig.h
index 054c954..43714f3 100644
--- a/Inc/FreeRTOSConfig.h
+++ b/Inc/FreeRTOSConfig.h
@@ -51,7 +51,7 @@
 #define configTICK_RATE_HZ         ( ( TickType_t ) 1000 )
 #define configMAX_PRIORITIES       ( 5 )
 #define configMINIMAL_STACK_SIZE   ( ( unsigned short ) 128 )
-#define configTOTAL_HEAP_SIZE      ( ( size_t ) ( 12 * 1024 ) )
+#define configTOTAL_HEAP_SIZE      ( ( size_t ) ( 15 * 1024 ) )
 #define configMAX_TASK_NAME_LEN    ( 16 )
 #define configUSE_16_BIT_TICKS       0
 #define configIDLE_SHOULD_YIELD      1
diff --git a/STM32F103C8_FLASH.ld b/STM32F103C8_FLASH.ld
index 017f5f4..646c1d7 100644
--- a/STM32F103C8_FLASH.ld
+++ b/STM32F103C8_FLASH.ld
@@ -34,8 +34,8 @@ ENTRY(Reset_Handler)
 /* Highest address of the user mode stack */
 _estack = 0x20005000;    /* end of RAM */
 /* Generate a link error if heap and stack don't fit into RAM */
-_Min_Heap_Size = 0x300;      /* required amount of heap  */
-_Min_Stack_Size = 0x450; /* required amount of stack */
+_Min_Heap_Size = 0x200;      /* required amount of heap  */
+_Min_Stack_Size = 0x400; /* required amount of stack */

 /* Specify the memory areas */
 MEMORY
diff --git a/Src/Tasks/UARTTask.c b/Src/Tasks/UARTTask.c
index 15448bc..101ed8a 100644
--- a/Src/Tasks/UARTTask.c
+++ b/Src/Tasks/UARTTask.c
@@ -44,10 +44,10 @@ void osQueueUARTMessage(const char * fmt, ...) {
        // TODO: Less copying around bits
        va_list arg;
-       char buffer[128];
+       char buffer[84];

        va_start(arg, format);
-       vsnprintf(buffer, 128, format, arg);
+       vsnprintf(buffer, 84, format, arg);
        va_end(arg);

diff --git a/Src/main.c b/Src/main.c
index b83ea50..8d8eb76 100644
--- a/Src/main.c
+++ b/Src/main.c
@@ -38,8 +38,8 @@ int main(void) {
        xI2CSemaphore = xSemaphoreCreateMutex();
        xDataEventGroup = xEventGroupCreate();

-       xTaskCreate(vCheckTask, "Check", 200, (void*) 1, 1, NULL);
-       xTaskCreate(vCheckTask, "Check", 200, (void*) 2, 8, NULL);
+       xTaskCreate(vCheckTask, "Check", 150, (void*) 1, 1, NULL);
+       xTaskCreate(vCheckTask, "Check", 150, (void*) 2, 8, NULL);

 #if SAT_Enable_Sensors
        xTaskCreate(vMPU9250Task, "MPU9250", 300, NULL, 4, NULL);
@@ -47,8 +47,8 @@ int main(void) {
 #endif

        xTaskCreate(vUARTTask, "UART", 300, NULL, 3, &xUARTTaskHandle);
-       xTaskCreate(vRefreshWWDGTask, "RefreshWWDG", 100, NULL, 6, NULL);
-       xTaskCreate(vBlinkyTask, "Blinking", 100, NULL, 3, NULL);
+       xTaskCreate(vRefreshWWDGTask, "RefreshWWDG", 60, NULL, 6, NULL);
+       xTaskCreate(vBlinkyTask, "Blinking", 60, NULL, 3, NULL);

 #if SAT_Enable_NRF24
        xTaskCreate(vTransmitTask, "NRF_TX", 250, NULL, 1, NULL);
```

4 Incomplete Tasks

Memory management in embedded systems can be tricky, especially when there is no general-purpose operating system that can take the abstraction away. Once again, the developer is needed to delve deep within the software's logic to resolve any issues that may occur.

Concerning memory management, some incomplete tasks are:

- **More studying** It would be useful to find resources explaining how to improve the distribution of memory on our application, such as the following:
  - The Lost Art of Structure Packing (credits to G. Pavlakis)
  - Justifiably taboo: Avoiding malloc()/free() APIs in military/aerospace embedded code
  - What are some best practices for reducing memory usage in C?
  - Possibly the worst sin of calling malloc is that it might take a very long time to complete.
  - Tips and Tricks – 7 Tips for Memory Management
- **Definition of good practices**
  Memory allocation can become very flexible and dangerous, or very safe but restrictive. The choice depends on the kind of the mission and the decisions taken by the development team. A good start for a baseline of good practices would be an embedded C standard, such as MISRA-C.
- **Expression of opinion against the heap**
  `malloc()` or `vPortMalloc()` can be a very powerful and simultaneously dangerous tool. Should it be used on our CubeSat, considering that our microcontrollers perform mission-critical operations?
  It is very hard to analyse the usage of the heap, compared to global variables

or the stack, whose sizes are known during compile-time to some extent. Risk analysis can be performed on using the heap, and questions such as the following should be answered:

- Is usage of the heap allowed in MISRA-C?
- Are other mission-critical and aërospace missions making use of the heap? What about embedded development in general? (Hint: There may be no single answer.)
- Is there a standard that defines safe ways to use it? Is documenting every call to it enough?

Notes

1. We can't store these variables statically, since a function calling itself would return incorrect results. Also, we would allocate more memory than needed.

2. See Cortex-M3 Devices Generic User Guide — 2.1.2. Stacks

3. Generated using *radare2*, with the `aaa`, `iS*` and `S=` commands.

4. Created with the `VV` command of *radare2*.

5. Created with Cutter.

6. Returned using the command `arm-none-eabi-objdump -t -j .data ./FreertosMockup.elf`.

7. Returned using the command `arm-none-eabi-objdump -t -j .rodata ./FreertosMockup.elf`.

8. As shown on Atollic's build analyzer.

9. General purpose OSs don't have this limitation, since they can expand the stack if the program requires more than is available.

10. EABI refers to the embedded-application binary interface, and *none* corresponds to the *platform* used (this can be set to `atollic`).

11. As an example, `text` contains `.text`, `.rodata` and `.isr_vector`.

12. The `--radix=d` option can be used for `nm` to use decimal instead of hexadecimal notation for sizes.

13. On a C++ application, the `c++filt` command should be called on the output to demangle the symbol names.

14. For more information about DWARF, see The DWARF debugging format.

15. Use the following fork that has some bugs fixed, until the pull requests are accepted: `https://github.com/kongr45gpen/linkermapviz`

Acronyms

EABI Embedded Application Binary Interface. 19

ELF Executable and Linkable Format. 5, 10

ISR Interrupt Service Routine. 7

MCU MicroController Unit. 7, 10

MPU Memory Protection Unit. 16

OS Operating System. 19

RAM Random Access Memory. 2

ROM Read-Only Memory. 2

Contents