

Комфортное программирование на C++ в условиях FreeRTOS на STM32

Алексей Булатов, IntelliVision

О чём будет лекция

- В чём специфика микроконтроллеров
- Операционные системы реального времени (RTOS)
- Многозадачность на МК на бытовом примере
- Стандартная библиотека C/C++, её связь с системой
- Что нужно FreeRTOS, чтобы работать на МК

Поговорим про микроконтроллеры

Что такое микроконтроллеры?

Микроконтрóллер (англ. *Micro Controller Unit, MCU*) — микросхема, предназначенная для управления электронными устройствами.

- маломощный компьютер в одном чипе
- машина времени в прошлое вычислительной техники
- ад для классических разработчиков



очень много разных. Мне нравится STM32,
говорим про него

STM32 глазами разработчика

- 32-битный **ARM** Cortex-M3, -M4, -M7F. Серия F3 ~ до 180 MHz, до 2056 kB flash, до 384 kB RAM
- Компилируется при помощи gcc-arm-none-eabi
- Есть Common Microcontroller Software Interface Standard (**CMSIS**) — сверхлёгкий API с регистрами и номерами битов
- Есть **STM HAL** — мощный универсальный C-API, код полностью открыт
- Есть **CubeMX** — утилита для генерации кода с STM HAL

Особенности разработки под МК

- Ограниченные ресурсы
- Зависимость от состояния внешнего мира
- Трудно отлаживать
- Трудно диагностировать
- Нет ОС с API, разделением памяти и привилегий
- Большинство ошибок ломают всё

Особенности разработки под МК

Нет ОС с API, разделением памяти и привилегий

- Нужно писать обработчики прерываний самому
- Единое адресное пространство, можно писать и читать по любому валидному адресу. Никакой виртуальной памяти, MMU, защиты
- Некуда докладывать о проблемах, никто не соберёт core dump
- В лучшем случае watchdog timer перезапустит код
- Отладчик хорошо отлаживает только простой код

Как выживать под МК?

- Печатать лог в последовательный порт
- Мигать светодиодом
- Писать хороший код
 - Велосипедам — нет, библиотекам — да
 - C++ безопаснее, чем C
 - ООП поможет укротить сложность

Особенности разработки под МК

Нет ОС с API, разделением памяти и привилегий

Ну а как же операционные системы реального времени?



Немного про операционные системы реального времени

Что такое RTOS?

— Real Time Operating System



1) Real Time — да

2) Operating System — с натяжкой

Главная задача RTOS — выполнять несколько задач
одновременно

Что такое RTOS?

Обычная ОС умеет:

- Многозадачность
- Много пользователей
- Абстракция над устройствами, драйверы
- API прикладных программ
- Изоляция выполнения программ
- Инфраструктура ввода-вывода
- ...

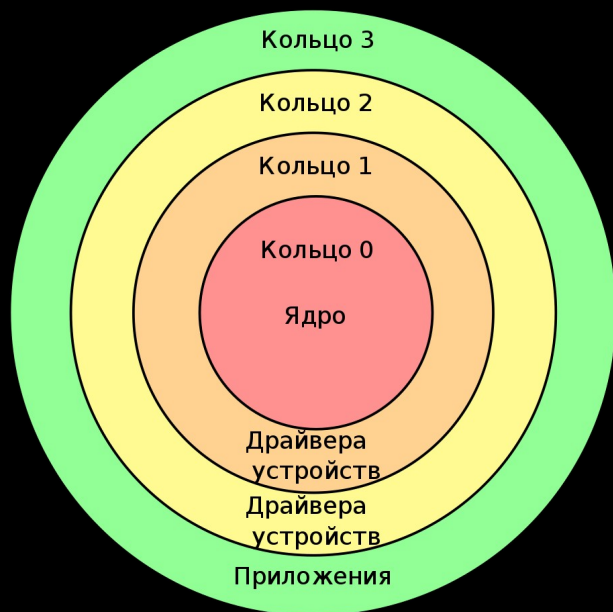
FreeRTOS умеет:

- Переключать треды
- Синхронизовать треды
- И всё. Это просто функции, которые вы линкуете со своим проектом

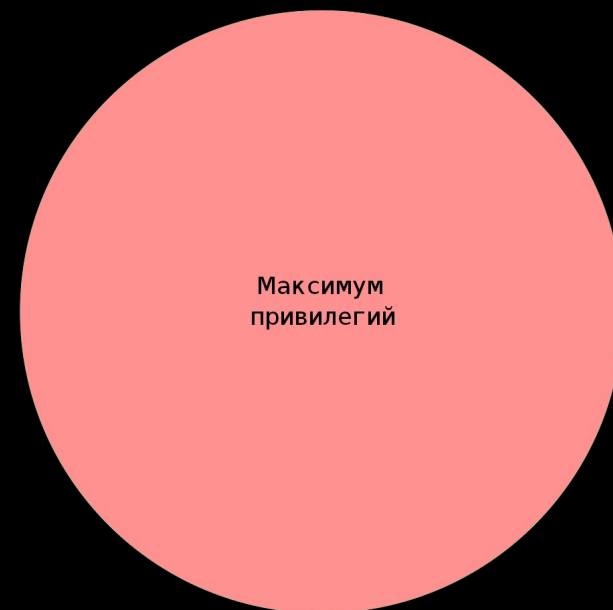
```
BaseType_t xTaskCreate(    TaskFunction_t pvTaskCode,  
                           const char * const pcName,  
                           const configSTACK_DEPTH_TYPE uxStackDepth,  
                           void *pvParameters,  
                           UBaseType_t uxPriority,  
                           TaskHandle_t *pxCreatedTask  
                           );
```

Безопасный режим

Ваш PC



Ваш МК



Пример устройства с многозадачностью

Пример: блок управления воротами



Что будет делать МК:

- Управлять приводом раздвижных ворот
- Следить за ИК-сенсором наличия препятствия
- Общаться с радиомодулем (например, LoRa)
- Считывать NFC-карточки доступа
- Слушать ИК-пульт
- Проигрывать звуковые оповещения

Как организовать многозадачность?

Main loop

```
while(true)
{
    serve_gate_drive();

    serve_sensor();

    serve_NFC();

    serve_radio();

    serve_IR_RC();

    load_next_sound_fragment();
}
```

RTOS

```
xTaskCreate(cycle_gate_drive, "gate", 128);
xTaskCreate(cycle_sensor, "sensor", 128);
xTaskCreate(cycle_NFC, "NFC", 1024);
xTaskCreate(cycle_radio, "Radio", 1024);
xTaskCreate(cycle_IR_RC, "IR_RC", 512);
xTaskCreate(cycle_load_next_sound_fragment,
"Sound", 512);

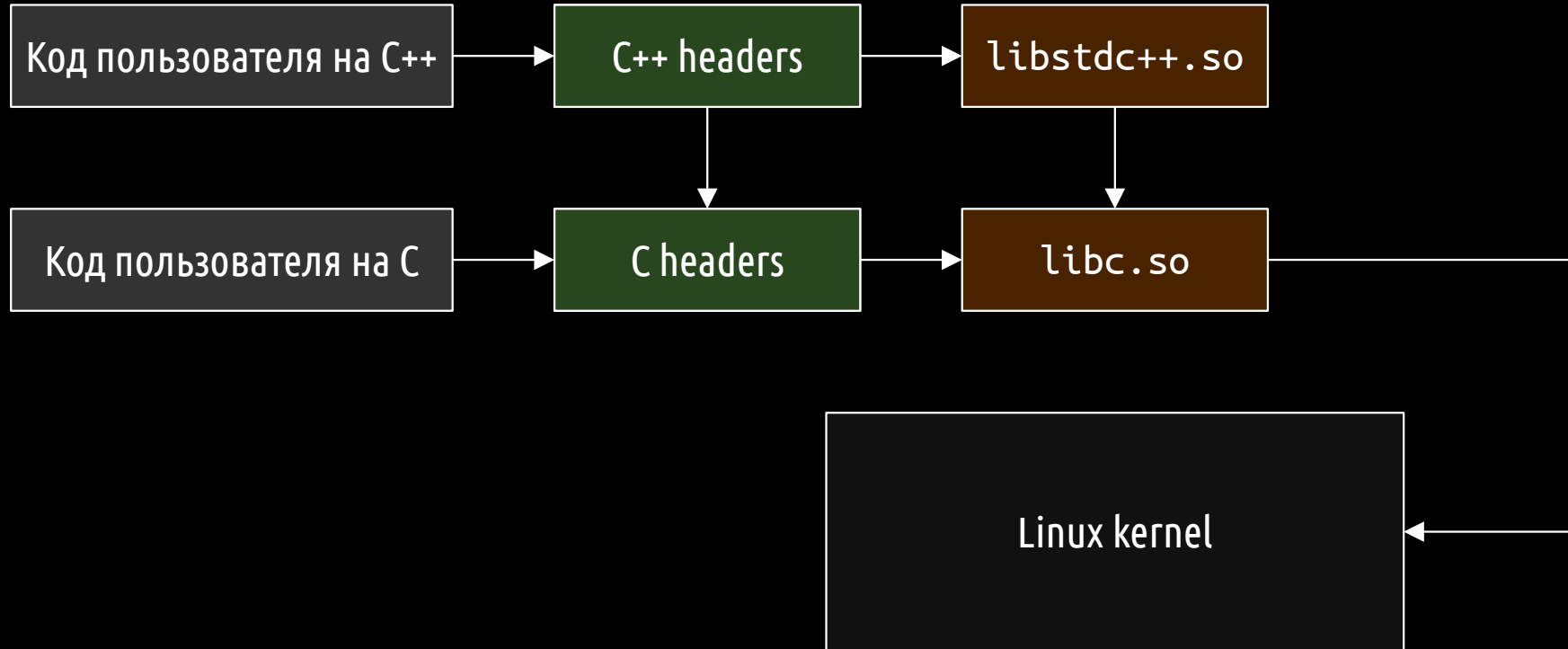
vTaskStartScheduler();
```


Что поможет открывать ворота?

- Стандартная библиотека С и С++
- FreeRTOS
- Чтобы они работали вместе

Стандартная библиотека C/C++ на низком уровне

Стандартная библиотека: Linux PC



Стандартная библиотека: Linux PC

main.cpp

```
...  
printf("hello\n");  
...
```

libc.so

stdio.h

```
printf("hello\n")
```

sys/unistd.h

```
write(1, "hello\n", 6)
```

sys/unistd.h

```
syscall(__NR_write, 1, "hello\n", 6)
```

syscall:

endbr64

mov rax,rdi

mov rdi,rsi

mov rsi,rdx

mov rdx,rcx

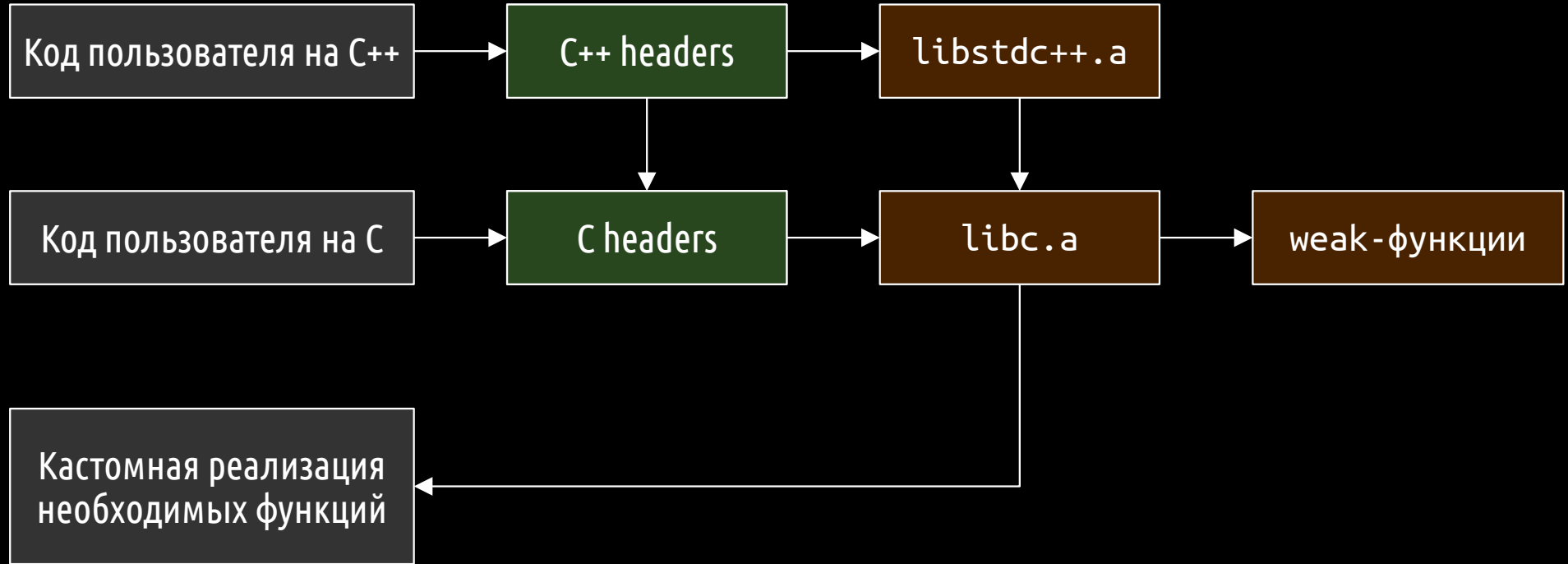
mov r10,r8

mov r8,r9

mov r9,QWORD PTR [rsp+0x8]

syscall

Стандартная библиотека: bare metal



Стандартная библиотека: bare metal

main.cpp

```
...  
printf("hello\n");  
...
```

Lite libc implementation, i.e. Newlib

stdio.h

```
printf("hello\n")
```

sys/unistd.h

```
write(1, "hello\n", 6)
```

sys/unistd.h

```
_write(1, "hello\n", 6)
```

```
int _write(int file, char *ptr, int len)  
{  
    switch (file)  
    {  
        case STDOUT_FILENO: /* stdout */  
        case STDERR_FILENO: /* stderr */  
            HAL_UART_Transmit(&huart2, (uint8_t*) ptr,  
                             len, HAL_MAX_DELAY);  
            break;  
        default:  
            errno = EBADF;  
            return -1;  
    }  
    return len;  
}
```

Какие функции нам определить?

- `void* _sbrk(int incr)` — увеличивает размер сегмента данных. Вызывается при `malloc()`
- `int _write(int file, char *ptr, int len)` — запись в файл или в поток вывода
- `int _read(int file, char *ptr, int len)`
`int _stat(const char *filepath, struct stat *st)`
`int _lseek(int file, int ptr, int dir)`
`int _close(int file)` — если у вас есть файловая система
- `int _gettimeofday(struct timeval *tv, struct timezone *tz)` — получение времени

Запускаем FreeRTOS

Настройка параметров системы

FreeRTOSConfig.h — compile-time configuration

```
96  #define configENABLE_FPU                0
97  #define configENABLE_MPU                0
98
99  #define configUSE_PREEMPTION              1
100 #define configSUPPORT_STATIC_ALLOCATION    1
101 #define configSUPPORT_DYNAMIC_ALLOCATION  1
102 #define configUSE_IDLE_HOOK               0
103 #define configUSE_TICK_HOOK               0
104 #define configCPU_CLOCK_HZ                ( SystemCoreClock )
105 #define configTICK_RATE_HZ                ((TickType_t)1000)
106 #define configMAX_PRIORITIES              ( 7 )
107 #define configMINIMAL_STACK_SIZE          ((uint16_t)128)
108 #define configTOTAL_HEAP_SIZE             ((size_t)15360)
109 #define configMAX_TASK_NAME_LEN           ( 16 )
110 #define configUSE_16_BIT_TICKS            0
111 #define configUSE_MUTEXES                  1
112 #define configQUEUE_REGISTRY_SIZE         8
113 #define configUSE_PORT_OPTIMISED_TASK_SELECTION 1
114 /* USER CODE BEGIN MESSAGE_BUFFER_LENGTH_TYPE */
```

Прерывания Cortex-M

Прерывание — это когда что-то случилось. Бывают аппаратные (возникают сами) и программные (можно инициировать из кода).

Для FreeRTOS необходимы эти три:

- **SysTick** — аппаратное прерывание системного таймера
- **Supervisor Call (SVC)** — программное прерывания, обрабатываемое немедленно
- **Pendable Service Call (PendSV)** — отложенное программное прерывание

startup_stm32f407xx.s

```
141 g_pfnVectors:
142 .word _estack
143 .word Reset_Handler
144 .word NMI_Handler
145 .word HardFault_Handler
146 .word MemManage_Handler
147 .word BusFault_Handler
148 .word UsageFault_Handler
149 .word 0
150 .word 0
151 .word 0
152 .word 0
153 .word SVC_Handler
154 .word DebugMon_Handler
155 .word 0
156 .word PendSV_Handler
157 .word SysTick_Handler
158
```

Прерывания Cortex-M в работе FreeRTOS

- `PendSV_Handler` — внутри него, в зависимости от ситуации, происходит переключение контекста
- `SysTick_Handler` — поднимает флаг `PendSV` с определенной частотой во времени

Пользователь может вызывать функции ОС, которые внутри тоже поднимают `PendSV`

Работа с FreeRTOS

```
13 void vGreenBlinkTask( void *pvParameters ) {
14     >> for( ;; ) {
15         >>     HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
16         >>     vTaskDelay(700);
17     >> }
18 }
19
20 void vRedBlinkTask( void *pvParameters ) {
21     >> for( ;; ) {
22         >>     HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_6);
23         >>     vTaskDelay(1000);
24     >> }
25 }
26
27 int main() {
28     >> InitSystem();
29
30     >> xTaskCreate(&vGreenBlinkTask, "GreenBlink", 128, nullptr, 1, NULL);
31     >> xTaskCreate(&vRedBlinkTask, "RedBlink", 128, NULL, 1, NULL);
32
33     >> vTaskStartScheduler();
34
35     >> for( ;; ) { }
36 }
```

Работа с FreeRTOS

- Треды будут прерываться, когда решит RTOS
- Для синхронизации доступны
 - Мьютексы
 - Семафоры
 - Очереди
 - Критические секции
- Прimitives можно использовать из прерываний, есть специальные версии функций *_FROM_ISR

Стандартная библиотека в условиях FreeRTOS

Стандартная библиотека в условиях FreeRTOS

А какие могут быть проблемы?

Reentrancy

— по-русски «реентрабельность».

В функцию могут входить одновременно несколько потоков исполнения

Reentrancy не гарантирует отсутствие гонки данных!!

Пример: `fwrite()` реентрабельна на РС, но что окажется в файле, зависит только от вас

Reentrancy

На PC:

- Системные вызовы реентрабельны
- Функции стандартной библиотеки C в основном реентрабельны
- Потокбезопасность функций C++ описана в стандарте

Когда у вас RTOS на микроконтроллере:

- Newlib должен как-то узнать о тредах RTOS

Reentrancy

Newlib должен как-то узнать о тредах RTOS

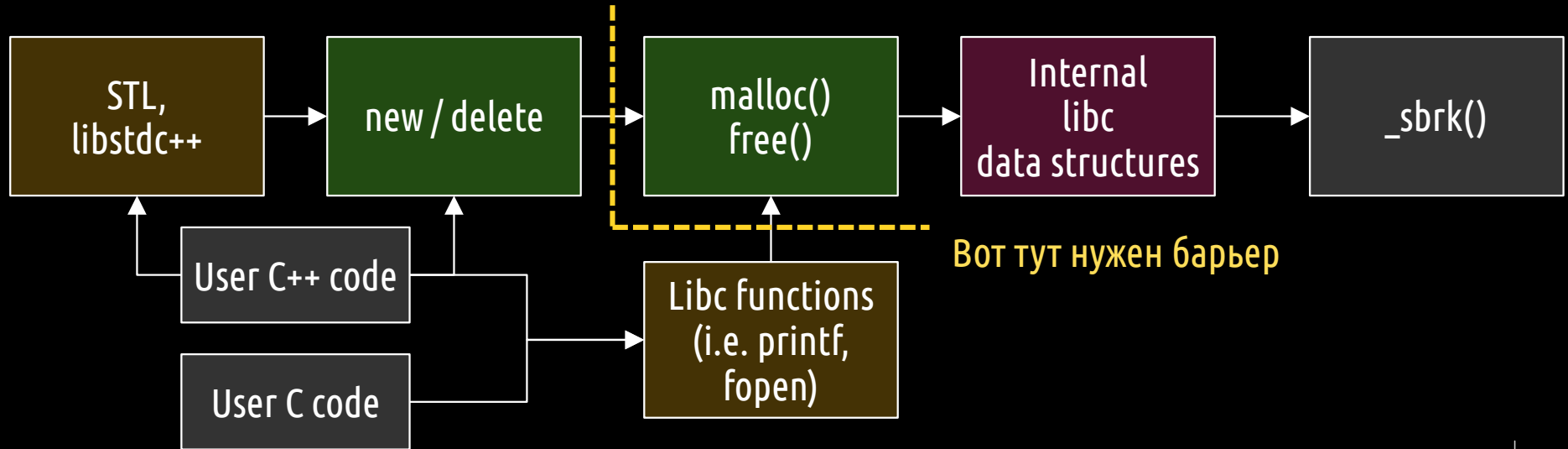
Реализуем функции из `unistd.h` безопасным образом.
Просто добавим блокировки

Но достаточно ли этого?

Reentrancy

Newlib должен как-то узнать о тредах RTOS

А что с выделением памяти?



Reentrancy: malloc & free

Воспользуемся оборачиванием функций во время линковки в gcc!

```
LD_FLAGS += -Wl,-wrap=malloc -Wl,-wrap=free
```

Reentrancy: malloc & free

GCC позволяет просто переопределить любую функцию

1) При линковке добавим ключи:

```
LDFLAGS += -Wl,--wrap=malloc -Wl,--wrap=free
```

2) Теперь есть 4 функции:

```
void* __real_malloc(size_t);  
void __real_free (void *);
```

) Оригинальные
функции

```
void* __wrap_malloc(size_t size) { ... }  
void __wrap_free(void *pv) { ... }
```

) Функции на замену

Reentrancy: malloc & free

```
24 void* __real_malloc(size_t);  
25 void __real_free(void *);  
26  
27 void* __wrap_malloc(size_t size)  
28 {  
29     void *pvReturn;  
30     vTaskSuspendAll();  
31     pvReturn = __real_malloc(size);  
32     (void)xTaskResumeAll();  
33     return pvReturn;  
34 }  
35  
36 void __wrap_free(void *pv)  
37 {  
38     vTaskSuspendAll();  
39     __real_free(pv);  
40     (void)xTaskResumeAll();  
41 }  
42
```

← Минимальный пример
безопасной обёртки для
FreeRTOS

Reentrancy*

Вообще говоря,
`::operator new()` !=
`malloc()` + constructor call

`::operator delete()` !=
`free()` + constructor call

Поэтому добавим глобальное
определение →

```
57 void* operator new(std::size_t n)
58 {
59     return malloc(n);
60 }
61
62 void operator delete(void* p) throw()
63 {
64     free(p);
65 }
66
67 void* operator new[](std::size_t n)
68 {
69     return malloc(n);
70 }
71
72 void operator delete[](void* p) throw()
73 {
74     free(p);
75 }
```

Reentrancy: newlib

Существует более канонический способ защитить newlib:

- Для функций из `unistd.h` есть реентрабельные вариации, например:

вместо `_sbrk(int incr)`
→ `_sbrk_r(struct _reent *r, int incr)`

вместо `_write(int file, char *ptr, int len)`
→ `_write_r(struct _reent *r, int file, char *ptr, int len)`

- Есть специальные блокировщики для `malloc` и `free`:

`void _malloc_lock(struct _reent *r)`
`void _malloc_unlock(struct _reent *r)`



Заключение

Что можно сказать

- Программирование для MCU менее комфортно, чем программирование для PC
- Чтобы использовать стандартную библиотеку C и C++, необходимо реализовать примитивы из `sys/unistd.h`
- RTOS решает проблему Real Time, но создаёт много проблем из-за отсутствия Operating System
- При использовании RTOS нужны потокобезопасные реализации функций `sys/unistd.h`
- `malloc()` и `free()` должны быть защищены дополнительно

Задание



<https://github.com/DAlexis/neimark-2024-stm>