Лабораторная работа №7. Многопоточное программирование

Освоение принципов многопоточного программирования и синхронизации работы нескольких потоков для организации управления периферийными подсистемами микроконтроллера.

Учебное время 8 часов

Цель работы: Привитие практических навыков по работе с библиотекой cmsis\_os.

* 1. Содержание работы

1. Ознакомиться с документацией по режимам работы: контроллера ILI9325 TFT LCD; микроконтроллера STM32F303x. Изучить функции библиотеки cmsis\_os
2. Выполнить работу согласно варианту.

**Краткие сведения о RTOS Keil RTX.**

RTOS (Real-Time Operating System) Keil RTX - это многозадачная операционная система реального времени, встроенная в среду разработки Keil, которая обеспечивает вытесняющую многозадачность. Она упрощает разработку сложных систем, улучшая управление проектами и способствуя повторному использованию кода. Основным компромиссом использования ОСРВ является увеличение объёма используемой памяти и возможное удлинение времени реакции на прерывания. Однако современные микроконтроллеры с объёмом ОЗУ от 32 КБ и выше предоставляют достаточные ресурсы для работы ОСРВ, типичный размер которой составляет всего 5 КБ.

Keil RTX реализует стандарт CMSIS-RTOS, что обеспечивает совместимость с другими ОСРВ, соответствующими этому стандарту. Это позволяет разработчикам легко переносить код между различными платформами и упрощает интеграцию ОСРВ в существующие проекты.

На сегодняшний день доступны две версии RTX: RTOS v1 (Keil RTX 4) и RTOS v2 (Keil RTX 5). В данной работе будет использоваться первая версия ОС, к особенностям которой можно отнести:

1. Многозадачность (Multitasking)

2. Наличие приоритетов задач

3. Синхронизация задач

4. Таймеры и задержки

5. Управление памятью (динамическое выделение памяти, разделение памяти между задачами)

6. Работа с прерываниями

7. Поддержка различных типов синхронизации (IPC), в том числе очередей сообщений, почтовых ящиков и сигналов событий

8. Поддержка энергоэффективности

**Архитектура RTX.**

RTX состоит из следующих основных компонентов:

* **Потоки (tasks):** это базовые элементы выполнения. Каждый поток выполняется в своём контексте с отдельным стеком.
* **Планировщик задач (scheduler):** управляет распределением процессорного времени между потоками, используя механизм вытесняющей многозадачности. В Keil RTX 4 используется планировщик Round Robin – (циклический опрос) - каждому потоку выделяется фиксированное время на выполнение.
* **Ядро ОСРВ:** обрабатывает события, управляет таймерами и взаимодействием между потоками.
* **Интерфейс взаимодействия с оборудованием:** использует системный таймер (SysTick) для управления временем выполнения задач.

**Многопоточность и сихронизация потоков.**

В отличие от функций, выполнение которых завершается после выполнения операций, потоки в ОСРВ (операционных системах реального времени) работают непрерывно. Их выполнение продолжается до завершения всей системы, благодаря наличию бесконечного цикла while(1) в их коде. Управление этими потоками осуществляется специальным компонентом ОСРВ — планировщиком задач. Планировщик использует системный таймер (обычно SysTick) для создания прерываний, которые распределяют процессорное время между потоками. Чередование потоков создаёт иллюзию одновременного выполнения нескольких задач, хотя фактически они выполняются последовательно.

**Управление потоками**

Каждому потоку выделяется определённый интервал времени, который можно настроить. Планировщик использует вытесняющую многозадачность, что позволяет более высоким приоритетам прерывать выполнение задач с низким приоритетом. Потоки могут находиться в одном из трёх состояний:

* **Running** - поток выполняется.
* **Ready** - поток готов к выполнению, ожидает выделения процессорного времени.
* **Wait** - поток заблокирован, ожидает наступления события (например, сигнала от другого потока, завершения операции ввода/вывода или истечения таймера).

**Создание потока:**

Для создания потока в Keil RTX используется функция osThreadCreate(). Каждому потоку необходимо назначить уникальную функцию, которая будет выполняться в рамках этого потока.

Пример создания потока:

// Функция потока

void myTask(void const \*argument) {

while(1) {

// Код задачи

}

}

int main(void) {

osThreadDef(task1, myTask, osPriorityNormal, 0, 128); // Определение потока

osThreadCreate(osThread(task1), NULL); // Создание потока

osKernelStart(); // Запуск ядра RTOS

while(1);

}

Здесь:

* osThreadDef(task1, myTask, osPriorityNormal, 0, 128) — создание определения потока с именем task1, функцией myTask, приоритетом osPriorityNormal, без аргументов (0) и размером стека 128.
* osThreadCreate(osThread(task1), NULL) — создание потока на основе определения.

Для завершения потока можно использовать функцию osThreadTerminate().

**Мьютексы (Mutexes)**

Мьютексы в Keil RTX используются для синхронизации доступа к общим ресурсам, чтобы избежать конфликтов при параллельном доступе. Мьютекс блокирует ресурс, пока он используется одним потоком, и освобождает его для других потоков после завершения работы с ним.

Мьютекс создается с помощью функции osMutexCreate(). Для захвата мьютекса используется osMutexWait(), а для освобождения — osMutexRelease().

Пример создания и использования мьютекса:

osMutexId mutexId; // Идентификатор мьютекса

// Функция потока 1

void task1(void const \*argument) {

while(1) {

osMutexWait(mutexId, osWaitForever); // Захват мьютекса

// Работа с общим ресурсом

osMutexRelease(mutexId); // Освобождение мьютекса

}

}

// Функция потока 2

void task2(void const \*argument) {

while(1) {

osMutexWait(mutexId, osWaitForever); // Захват мьютекса

// Работа с общим ресурсом

osMutexRelease(mutexId); // Освобождение мьютекса

}

}

int main(void) {

osMutexDef(mutex1);

mutexId = osMutexCreate(osMutex(mutex1)); // Создание мьютекса

osThreadDef(task1, task1, osPriorityNormal, 0, 128);

osThreadCreate(osThread(task1), NULL);

osThreadDef(task2, task2, osPriorityNormal, 0, 128);

osThreadCreate(osThread(task2), NULL);

osKernelStart(); // Запуск RTOS

while(1);

}

Здесь:

* osMutexDef(mutex1) — создание определения мьютекса.
* osMutexCreate(osMutex(mutex1)) — создание мьютекса.
* osMutexWait(mutexId, osWaitForever) — захват мьютекса. Если он уже занят, задача будет ожидать освобождения.
* osMutexRelease(mutexId) — освобождение мьютекса.

**Семафоры (Semaphores)**

Семафоры используются для синхронизации задач, особенно в случаях, когда необходимо ограничить доступ к ресурсу. Семафор может быть бинарным или счётным, в зависимости от того, сколько задач могут одновременно получить доступ к ресурсу.

Семафор создается с помощью osSemaphoreCreate(). Для захвата семафора используется osSemaphoreWait(), а для его освобождения — osSemaphoreRelease().

Пример создания и использования семафора:

osSemaphoreId semaphoreId; // Идентификатор семафора

// Функция потока

void task1(void const \*argument) {

while(1) {

osSemaphoreWait(semaphoreId, osWaitForever); // Захват семафора

// Работа с ресурсом

osSemaphoreRelease(semaphoreId); // Освобождение семафора

}

}

int main(void) {

osSemaphoreDef(sem1);

semaphoreId = osSemaphoreCreate(osSemaphore(sem1), 1); // Создание семафора 1

osThreadDef(task1, task1, osPriorityNormal, 0, 128);

osThreadCreate(osThread(task1), NULL);

osKernelStart(); // Запуск RTOS

while(1);

}

Здесь:

* osSemaphoreDef(sem1) — создание определения семафора.
* osSemaphoreCreate(osSemaphore(sem1), 1) — создание семафора с начальным значением 1.
* osSemaphoreWait(semaphoreId, osWaitForever) — захват семафора. Поток будет ожидать, если семафор не доступен.
* osSemaphoreRelease(semaphoreId) — освобождение семафора.

**Очереди сообщений (Message Queues)**

Очереди сообщений позволяют обмениваться данными между потоками. Задачи могут отправлять и получать сообщения через очередь, что полезно для передачи данных или событий между потоками. Важно учесть, что все операции внутри очереди окружены другими примитивами синхронизации, ввиду чего очередь сообщений является безопасным способом организации межпроцессорного взаимодействия.

Очередь сообщений создается с помощью функции osMessageCreate(). Для отправки сообщения используется osMessagePut(), а для получения — osMessageGet().

Пример создания и использования очереди сообщений:

osMessageQId queueId; // Идентификатор очереди

// Функция потока 1 (отправка сообщений)

void task1(void const \*argument) {

uint32\_t message = 0;

while(1) {

osMessagePut(queueId, message++, osWaitForever); // Отправка сообщения

osDelay(100); // Задержка

}

}

// Функция потока 2 (приём сообщений)

void task2(void const \*argument) {

osEvent event;

while(1) {

event = osMessageGet(queueId, osWaitForever); // Получение сообщения

if (event.status == osEventMessage) {

// Обработка полученного сообщения

}

}

}

int main(void) {

osMessageQDef(queue1, 10, uint32\_t); // Определение очереди на 10 сообщений

queueId = osMessageCreate(osMessageQ(queue1), NULL); // Создание очереди

osThreadDef(task1, task1, osPriorityNormal, 0, 128);

osThreadCreate(osThread(task1), NULL);

osThreadDef(task2, task2, osPriorityNormal, 0, 128);

osThreadCreate(osThread(task2), NULL);

osKernelStart(); // Запуск RTOS

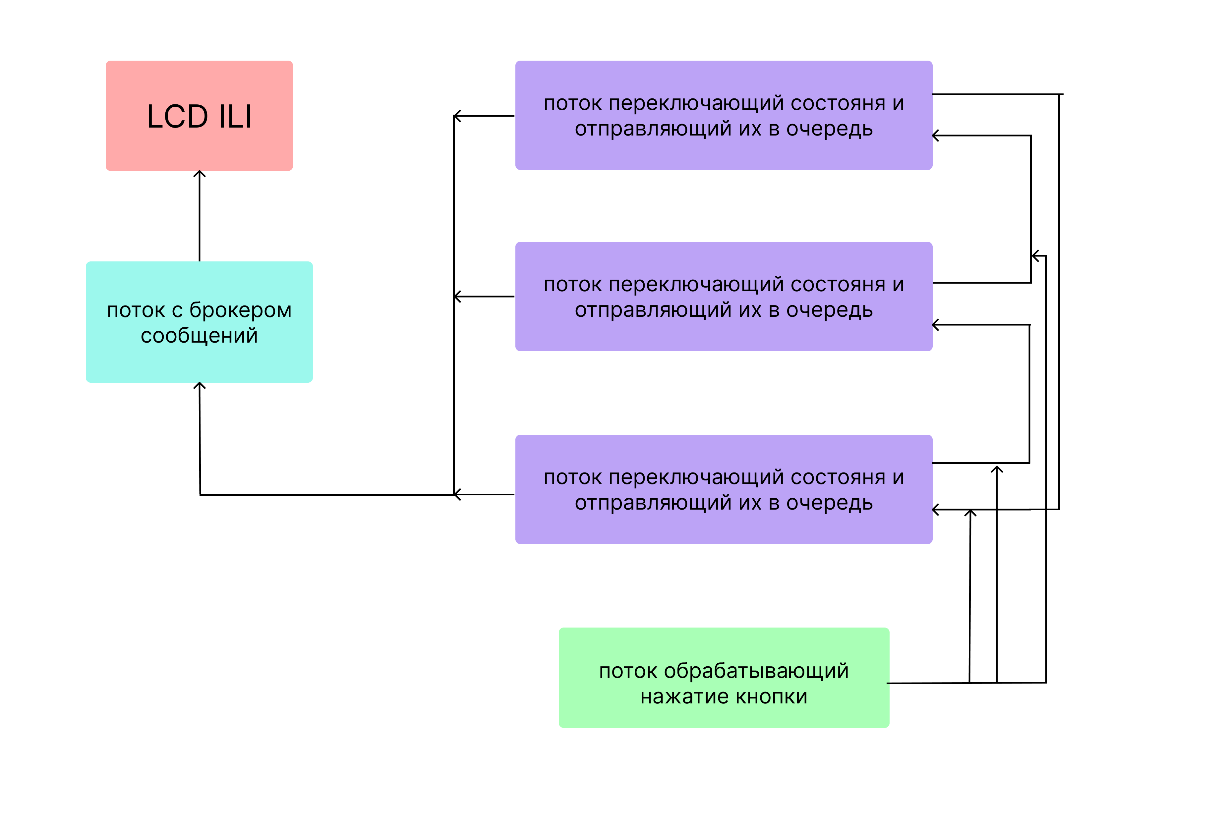
while(1);

}

Здесь:

* osMessageQDef(queue1, 10, uint32\_t) — определение очереди с вместимостью 10 и типом данных uint32\_t.
* osMessageCreate(osMessageQ(queue1), NULL) — создание очереди.
* osMessagePut(queueId, message++, osWaitForever) — отправка сообщения в очередь.
* osMessageGet(queueId, osWaitForever) — получение сообщения из очереди.

**Предложенная схема к реализации взаимодействия светофоров**



Создадим для каждого светофора свой поток, который будет отвечать за обновление его состояния, назовём эти потоки потоками светофора. Поток светофора через брокера сообщений добавляет свое состояние в очередь сообщений. В брокере сообщений запущен поток, который получает все состояния светофоров и в зависимости от этих данных обновляет дисплей. Создадим поток, обрабатывающий нажатие на кнопки. Нажатие на кнопки может:

* Привести все светофоры в определённое состояние
* Увеличивать состояние потока на 1 раз в n мс, пока кнопка нажата, тем самым ускорим работу светофоров.

Важно рассмотреть, как должны обновляться состояния в потоке светофора. Светофор должен содержать внутри себя своё текущее состояние. Сделаем это состояние доступным для других потоков, чтобы они могли изменять его.

… !!!!!!! Паша вставь код небольшой урезанный код switch

Светофор каждую итерацию проходится по switch, и сопоставив своё состояние начинает ожидать в нём некоторое, заданное вариантом время. После завершения ожидания поток меняет своё внутреннее состояние на следующее по счёту. Важно, чтобы функция ожидания учитывала прерывания с помощью кнопок.

… !!!!!!! Паша вставь код небольшой урезанный код custom delay

Светофор может изменить своё состояние и изменить состояние другого светофора. Для гарантии, что только один поток будет иметь доступ к переменной состояния используем семафоры.

… !!!!!! Паша вставь код небольшой урезанный код логики в switch изменения состояния другого потока.

При нажатии на кнопку тоже будем обращаться к внутреннему состоянию потока светофора, чтобы изменить и при этом гарантировать доступ к переменной состояния потока используем семафор.

… !!!!!! Паша вставь код небольшой урезанный код логики изменения состояния потока в кнопке.

**Варианты**

Возможные состояния для одного сигнала светофора:

R01[(КЯ), 3 сек]

Y01[(ЖТ), 2 сек]

A01[(АМ), 4 сек]

G01[(ЗЯ), 1 сек]

O01[(ОМ), 3 сек]

R02[(КТ), 2 сек]

Y02[(ЖМ), 4 сек]

A02[(АЯ), 1 сек]

G02[(ЗМ), 3 сек]

O02[(ОТ), 2 сек]

R03[(КМ), 4 сек]

Y03[(ЖЯ), 3 сек]

A03[(АТ), 2 сек]

G03[(ЗТ), 1 сек]

O03[(ОЯ), 4 сек]

Возможные состояния для двух сигналов светофора:

R02[(КЯ, ОТ), 3 сек]

Y02[(ЖТ, АЯ), 2 сек]

A02[(АМ, ЗТ), 4 сек]

G02[(КТ, ЖМ), 1 сек]

O02[(ОМ, ЗЯ), 3 сек]

RG2[(КЯ, ЖТ), 2 сек]

YA2[(ЖМ, АЯ), 4 сек]

AG2[(АМ, ЗМ), 1 сек]

GO2[(ЗЯ, ОМ), 3 сек]

OR2[(ОТ, КТ), 2 сек]

RY2[(КМ, ЖЯ), 4 сек]

YB2[(ЖТ, АМ), 3 сек]

BG2[(АМ, ЗЯ), 2 сек]

GR2[(ЗТ, КЯ), 1 сек]

OG2[(ОМ, ЗМ), 3 сек]

Возможные состояния для трёх сигналов светофора:

R03[(КЯ, ОТ, ЖМ), 3 сек]

G03[(КТ, ОЯ, ЖТ), 2 сек]

Y03[(КЯ, ОТ, АМ), 4 сек]

A03[(КТ, ЖТ, ЗЯ), 1 сек]

O03[(КЯ, ЖМ, АТ), 3 сек]

RG3[(КМ, ОТ, ЖТ), 2 сек]

YA3[(ЖЯ, АМ, ЗТ), 4 сек]

AG3[(КЯ, ЖМ, ЗМ), 1 сек]

GO3[(КМ, ЗЯ, ОТ), 3 сек]

OR3[(КТ, ОМ, ЖТ), 2 сек]

RY3[(КЯ, ЖЯ, ОТ), 4 сек]

YB3[(ЖМ, АЯ, ЗМ), 3 сек]

BG3[(КТ, ОЯ, ЖТ), 2 сек]

GR3[(ЗТ, КМ, ОМ), 1 сек]

OG3[(ОМ, ЗМ, КЯ), 3 сек]

Светофоры должны переключаться автоматически, как описано ранее, но сигналы должны следовать строго в указанной последовательности: следующий сигнал (n) включается только после полного завершения текущего сигнала (n-1).

Вариант 1:

1R01 1Y01 2A01 2G01 3O01 3R02 4Y02 4A02 5G02 5O02 6R03 6Y03 7A03 7G03 8O03 8R01 1Y01 2A01 3G01 4O01

Вариант 2:

1R02 1Y02 2A02 2G02 3O02 3R03 4Y03 4A03 5G03 5O03 6R01 6Y01 7A01 7G01 8O01 8R02 1Y02 2A02 3G02 4O02

Вариант 3:

1R03 1Y03 2A03 2G03 3O03 3R01 4Y01 4A01 5G01 5O01 6R02 6Y02 7A02 7G02 8O02 8R03 1Y03 2A03 3G03 4O03

Вариант 4:

1R01 1Y02 2A01 2G02 3O01 3R03 4Y01 4A03 5G01 5O03 6R02 6Y03 7A02 7G03 8O02 8R01 1Y02 2A01 3G02 4O01

Вариант 5:

1R02 1Y03 2A02 2G03 3O02 3R01 4Y02 4A01 5G02 5O01 6R03 6Y01 7A03 7G01 8O03 8R02 1Y03 2A02 3G03 4O02

Вариант 6:

1R03 1Y01 2A03 2G01 3O03 3R02 4Y03 4A02 5G03 5O02 6R01 6Y02 7A01 7G02 8O01 8R03 1Y01 2A03 3G01 4O03

Вариант 7:

1R01 1Y03 2A01 2G03 3O01 3R02 4Y01 4A02 5G01 5O02 6R03 6Y02 7A03 7G02 8O03 8R01 1Y03 2A01 3G03 4O01

Вариант 8:

1R02 1Y01 2A02 2G01 3O02 3R03 4Y02 4A03 5G02 5O03 6R01 6Y03 7A01 7G03 8O01 8R02 1Y01 2A02 3G01 4O02

Вариант 9:

1R03 1Y02 2A03 2G02 3O03 3R01 4Y03 4A01 5G03 5O01 6R02 6Y01 7A02 7G01 8O02 8R03 1Y02 2A03 3G02 4O03

Вариант 10:

1R02 1Y02 2A02 2RG2 3YA2 3GO2 4OR2 4RY2 1Y02 2A02 3GO2 4RY2 1R02 2YA2 3OR2 4GO2 1Y02 2RG2 3YA2 4RY2

Вариант 11:

1RG2 1YA2 2GO2 2OR2 3RY2 3RG2 4YA2 4GO2 1OR2 2YA2 3RY2 4RG2 1GO2 2RG2 3OR2 4YA2 1RY2 2YA2 3RG2 4GO2

Вариант 12:

1RY2 1RG2 2YA2 2GO2 3OR2 3RY2 4RG2 4YA2 1GO2 2RY2 3OR2 4YA2 1RG2 2OR2 3YA2 4GO2 1RY2 2GO2 3RG2 4OR2

Вариант 13:

1YA2 1GO2 2RG2 2RY2 3YA2 3OR2 4GO2 4RY2 1RG2 2YA2 3RY2 4OR2 1GO2 2RG2 3OR2 4YA2 1RY2 2YA2 3RG2 4GO2

Вариант 14:

1OR2 1RG2 2GO2 2YA2 3RY2 3OR2 4RG2 4GO2 1YA2 2RG2 3OR2 4YA2 1RY2 2GO2 3YA2 4RG2 1GO2 2RY2 3RG2 4YA2

Вариант 15:

1RG2 1OR2 2YA2 2RY2 3GO2 3YA2 4RY2 4RG2 1YA2 2GO2 3OR2 4YA2 1RY2 2RG2 3YA2 4GO2 1RG2 2GO2 3OR2 4YA2

Вариант 16:

1RY2 1YA2 2OR2 2GO2 3RG2 3RY2 4YA2 4OR2 1GO2 2RG2 3YA2 4RY2 1YA2 2GO2 3RG2 4OR2 1RY2 2YA2 3GO2 4RG2

Вариант 17:

1GO2 1RG2 2RY2 2OR2 3YA2 3RG2 4GO2 4RY2 1YA2 2OR2 3RG2 4GO2 1RY2 2YA2 3OR2 4RG2 1GO2 2RY2 3YA2 4OR2

Вариант 18:

1OR2 1RG2 2YA2 2GO2 3RG2 3RY2 4GO2 4YA2 1RY2 2RG2 3OR2 4GO2 1YA2 2GO2 3YA2 4RG2 1RG2 2YA2 3RY2 4OR2

Вариант 19:

1R03 1YA3 2GO3 2RG3 3OR3 3YA3 1R03 2YA3 3GO3 1RG3 2OR3 3YA3 1R03 2GO3 3RG3 1YA3 2RG3 3OR3 1GO3 2YA3

Вариант 20:

1GO3 1RG3 2OR3 2YA3 3GO3 3RG3 1R03 2GO3 3YA3 1RG3 2OR3 3YA3 1GO3 2YA3 3RG3 1YA3 2RG3 3OR3 1R03 2GO3

Вариант 21:

1RG3 1YA3 2OR3 2GO3 3RG3 3YA3 1R03 2RG3 3OR3 1YA3 2GO3 3YA3 1RG3 2OR3 3YA3 1GO3 2RG3 3OR3 1YA3 2GO3

Вариант 22:

1YA3 1GO3 2RG3 2YA3 3OR3 3RG3 1R03 2OR3 3YA3 1RG3 2GO3 3YA3 1YA3 2RG3 3OR3 1GO3 2YA3 3YA3 1RG3 2OR3

Вариант 23:

1RG3 1YA3 2GO3 2OR3 3RG3 3YA3 1R03 2YA3 3RG3 1GO3 2OR3 3YA3 1YA3 2RG3 3OR3 1RG3 2GO3 3YA3 1R03 2YA3

Вариант 24:

1OR3 1RG3 2YA3 2GO3 3OR3 3YA3 1R03 2RG3 3GO3 1YA3 2OR3 3YA3 1RG3 2YA3 3RG3 1GO3 2RG3 3OR3 1YA3 2GO3

Вариант 25:

1YA3 1GO3 2RG3 2OR3 3YA3 3RG3 1R03 2GO3 3OR3 1RG3 2YA3 3YA3 1GO3 2RG3 3YA3 1YA3 2OR3 3RG3 1R03 2GO3

Вариант 26:

1GO3 1RG3 2OR3 2YA3 3RG3 3GO3 1R03 2YA3 3OR3 1RG3 2GO3 3YA3 1OR3 2RG3 3YA3 1YA3 2GO3 3RG3 1R03 2YA3

Вариант 27:

1RG3 1YA3 2GO3 2RG3 3YA3 3OR3 1R03 2OR3 3RG3 1YA3 2YA3 3GO3 1RG3 2RG3 3YA3 1GO3 2OR3 3YA3 1R03 2YA3

Вариант 37:

1YA3 1RG3 2OR3 2YA3 3GO3 3RG3 1R03 2RG3 3OR3 1YA3 2GO3 3YA3 1RG3 2OR3 3YA3 1GO3 2YA3 3RG3 1R03 2GO3