УТВЕРЖДЕН

RU.27423071.00001-01 33 01-ЛУ

ОПЕРАЦИОННАЯ СИСТЕМА РЕАЛЬНОГО ВРЕМЕНИ ДЛЯ МУЛЬТИАГЕНТНЫХ КОГЕРЕНТНЫХ СИСТЕМ

Руководство программиста

RU.27423071.00001-01 33 01

Листов 63

Инв. № подл.	Подп. и дата	Взам. инв. №	Инв. № дубл.	Подп. и дата

2016

АННОТАЦИЯ

Настоящее руководство программиста распространяется на операционную систему реального времени для мультиагентных когерентных систем (далее – ОСРВ МАКС).

Документ содержит общие сведения об ОСРВ МАКС, сведения о её функциональном назначении и области применения, значения основных характеристик, описание методов обращения к ОСРВ МАКС, инструкции по разработке пользовательских приложений для функционирования в среде ОСРВ МАКС, описание входных и выходных данных и формируемых сообщений. В Приложении к настоящему документу приведены инструкции по совместной компиляции ОСРВ МАКС и пользовательского приложения и загрузке полученного в результате компиляции образа в целевое устройство.

Документ предназначен для специалистов, осуществляющих эксплуатацию ОСРВ МАКС и разработку пользовательских приложений для функционирования в среде ОСРВ МАКС.

Документ соответствует требованиям ГОСТ 19.105-78, ГОСТ 19.106-78 и ГОСТ 19.504-79.

СОДЕРЖАНИЕ

1. НАЗНАЧЕНИЕ И УСЛОВИЯ ПРИМЕНЕНИЯ

1.1. **Назначение программы**

1.1.1. ОСРВ МАКС предназначена для предоставления интерфейса доступа пользовательских приложений к ресурсам систем реального времени и иных встраиваемых систем, функционирующих под управлением поддерживаемых ОСРВ МАКС микроконтроллеров.

1.1.2. ОСРВ МАКС может использоваться при создании мультиагентных когерентных систем реального времени, включающих в себя несколько аппаратных средств – «узлов», отвечающих требованиям к среде функционирования ОСРВ МАКС.

1.2. **Условия применения**

1.2.1. ОСРВ МАКС предназначена для использования в составе систем реального времени или иных встраиваемых систем под управлением микроконтроллеров Cortex-M3, Cortex-M4, Cortex-M4F.

1.2.2. ОСРВ МАКС обеспечивает функционирование пользовательских приложений, созданных с помощью сред разработки Keil µVision и IAR Embedded Workbench for ARM (EWARM 7.5) при использовании API, входящего в состав ОСРВ МАКС.

1.2.3. ОСРВ МАКС обеспечивает функционирование пользовательских приложений, исходный код которых написан на языке программирования C++ (допускается использование элементов кода на языках программирования C, ассемблер) в соответствии с правилами и указаниями настоящего документа.

2. ХАРАКТЕРИСТИКА ПРОГРАММЫ

2.1. **Основные характеристики**

2.1.1. Основные характеристики программы перечислены в таблице 1.

Т а б л и ц а 1

Характеристика	Значение характеристики
Архитектура	модульная архитектура с ядром и необязательными компонентами
Реализуемые функциональные возможности	многозадачность, планирование, переключение контекста, взаимодействие / синхронизация задач, управление разделяемой и динамической памятью, управление прерываниями
Инструменты обеспечения предсказуемой производительности реального времени	приоритетное планирование, механизм предотвращения инверсии приоритетов
Модель	все задачи выполняются в одном адресном пространстве без какой-либо защиты
Максимальное число задач	ограничено размером доступной памяти
Механизмы синхронизации / взаимодействия	семафоры, мьютексы, события, очереди сообщений
Политики планирования	планирование на основании приоритетов, циклическое между потоками на одном приоритетном уровне, если системный квант времени установлен в 0, поток выполняется до завершения
Управление прерываниями	вложенные прерывания с приоритетами, обработчики прерываний выполняются в отдельном контексте, IST выступает как обычный поток приложения и имеет свой собственный стек, Из ISR можно подать сигнал в IST только с помощью события.
Инструменты управления временем	часы (clock), интервальный таймер

2.2. **Поставка и компиляция**

2.2.1. ОСРВ МАКС поставляется в виде исходных текстов и компилируется вместе с кодом пользовательского приложения. Инструкции по компиляции приведены в Приложении к настоящему руководству.

2.3. **Процессы и потоки (задачи)**

2.3.1. В текущей реализации целевой платформой ОСРВ МАКС является микроконтроллер ARM семейства Cortex M3/M4. Контроллеры данного семейства не содержат в своём составе блока MMU, обеспечивающего виртуализацию памяти, поэтому понятия «процесс» в чистом виде в ОСРВ МАКС быть не может. Как следствие, в рамках одного узла допускаются только потоки (в терминологии ОСРВ МАКС - задачи, Tasks).

2.3.2. При функционировании ОСРВ МАКС в составе мультиагентной когерентной системы реального времени каждый узел такой системы может быть рассмотрен в виде отдельного процесса, так как он исполняется на выделенном микроконтроллере, что изолирует его адресное пространство и прочие ресурсы от других узлов. ОСРВ МАКС же предоставляет узлам средства межпроцессной взаимосвязи.

2.4. **Режимы доступа для задач**

2.4.1. Контроллеры семейства Cortex M3/M4 поддерживают выполнение кода в двух режимах: привилегированном и непривилегированном. Непривилегированный режим отличается тем, что в нем запрещены определенные (потенциально опасные) операции – например, изменение регистров NVIC или доступ к некоторым областям памяти (при соответствующей настройке MPU).

2.4.2. Разделение кода на привилегированный и непривилегированный может использоваться при создании встроенных систем повышенной надежности. Такое разделение может позволить системе продолжить функционировать даже после того как непривилегированная задача дала сбой.

2.4.3. Запуск задачи в привилегированном либо непривилегированном режиме задается параметрами метода Task::Add() (см. 3.2.2.2), а также символом препроцессора MAKS_PROFILING_ENABLED при компиляции (см. 4.4).

3. ОПИСАНИЕ API

3.1. **Общие сведения**

3.1.1. ОСРВ МАКС поставляется совместно с API – интерфейсом программирования пользовательских приложений. Описание API приведено ниже.

3.1.2. API ОСРВ МАКС включает в себя проект для использования в среде разработки Keil µVsion 5 (maksRTOS.uvprojx) и рабочее пространство для использования в среде разработки EWARM 7.5 (maksRTOS_workspace.eww). Указанные файлы расположены в каталоге Compilers на оптическом носителе данных ОСРВ МАКС. Проект Keil µVsion 5 и рабочее пространство EWARM 7.5 содержат в себе файлы, необходимые для разработки пользовательского приложения и совместной компиляции ОСРВ МАКС и пользовательского приложения. Описание процесса компиляции и загрузки ОСРВ МАКС и пользовательского приложения приведено в приложении к настоящему документу.

3.1.3. В дальнейшем при описании правил и процедур предполагается, что разработка пользовательского приложения производится в рамках входящего в API проекта (рабочего пространства).

3.1.4. Код, выполняемый на микроконтроллере, состоит из микроядра ОС, драйверов устройств и приложения пользователя, использующего сервисы, предоставляемые ОС и драйверами, для выполнения прикладных задач. Данный раздел описывает API для использования сервисов микроядра.

3.1.5. Приложение пользователя представляет собой совокупность задач, исполняющихся псевдопараллельно под управлением планировщика. Создание задач и их запуск возложены на разработчика приложения. Изначально в списках планировщика есть лишь одна задача – IDLE Task, имеющая низший возможный приоритет. Она получает управление лишь в том случае, когда остальные задачи неактивны. В эти моменты IDLE Task может выполнять некоторые системные функции, например, переводить контроллер в спящий режим и т.п.

3.2. **Классы для организации выполнения программы**

Многие из описанных ниже методов возвращают значение типа Result, определяемого следующим образом:

```
enum Result
{
      // операция завершена успешно
      ResultOk,
```

```
            // операция завершена по истечению таймаута
            ResultTimeout,

            // вызов данной операции из прерывания запрещён
            ResultErrorInterruptNotSupported,

            // вызов данной операции из прерывания запрещён, т.к. его приоритет больше
            // (числовое значение меньше) чем MAX_SYSCALL_INTERRUPT_PRIORITY
            ResultErrorSysCallNotAllowed,

            // выполнение операции не поддерживается
            ResultErrorNotSupported,

            // аргументы операции некорректны
            ResultErrorInvalidArgs,

            // внутреннее состояние объекта или состояние системы некорректно для
выполнения операции,
            // например:
            // - при вызове Mutex::Lock(), текущая задача уже является его владельцем
            //  - при вызове Semaphore::Signal(), текущий счётчик уже равен
максимальному значению
            ResultErrorInvalidState
      };
```

3.2.1. Приложение (класс Application)

Класс Application является базовым. Реальный класс, представляющий приложение пользователя, должен быть производным от него. Следующие виртуальные методы класса Application должны быть обязательно определены в классе приложения пользователя:

– Initialize()

Стандартная схема использования класса приложения: создать его экземпляр и выполнить метод Run():

```
int main()
{
   TouchTestApp app;
   app.Run();
   return 0;   // сюда мы, на самом деле, никогда не попадем
}
```

Метод Run() выполняет необходимую инициализацию ОС, а также в нужный момент вызывает метод Initialize().

3.2.1.1. Метод Initialize()

Прототип метода:

```
virtual void Initialize() = 0;
```

Каждый класс приложения должен реализовывать данный метод, внутри которого производится начальная инициализация приложения. В том числе, там удобнее всего добавлять задачи для программы.

Пример 1:

```
class TouchTestApp : public Application
{
private:
   virtual void Initialize();
};
...
void TouchTestApp::Initialize()
{
   Touch_Initialize();

   Task::Add(new TouchTask());
}
```

Пример 2:

```
void DemoApp3::Initialize()
{
   RCC->AHB1ENR |= RCC_AHB1ENR_CRCEN;  // **
   GUI_Init();
   Touch_Initialize();

   DemoApp3TaskGfx* taskGfx = new DemoApp3TaskGfx("demo3_gfx");
   DemoApp3TaskTouch* taskTouch =
                     new DemoApp3TaskTouch("demo3_touch", taskGfx);
   Task::Add(taskGfx, Task::PriorityNormal, 0x100);
   Task::Add(taskTouch, Task::PriorityNormal, 0x100);
}
```

Обратите внимание, что метод Initialize() выполняется с привилегированным доступом, благодаря чему здесь возможна инициализация системных регистров (строка **), невозможная, например, в методе `DemoApp3TaskGfx.Execute()`.

3.2.2. Задача (класс Task)

Данный класс является базовым для реализации задачи. Любая задача должна быть унаследована от него. В унаследованном классе следует переопределить виртуальный метод Execute().

3.2.2.1. Конструктор класса

Прототип конструктора:

```
Task(const char* name);
```

name – имя задачи (произвольная строка).

При необходимости, конструктор можно переопределить для специфической инициализации разрабатываемого класса и/или добавить конструктор с несколькими параметрами.

Пример:

```
   Task::Add(new ResEmiterTask("emit"), Task::PriorityRealtime, 0x50);
```

3.2.2.2. Метод Add()

Прототипы:

```
   static Result Add(Task* task, Task::Priority priority = Task::PriorityNormal,
size_tstackSize = Task::MIN_STACK_SIZE);
   static Result Add(Task* task, Task::Priority priority = Task::PriorityNormal,
Task::Mode mode = Task::ModeUnprivileged, size_t stackSize = Task::MIN_STACK_SIZE);
```

Используется для добавления задачи в планировщик. Аргументы:

- task – указатель на объект «задача», добавляемый в планировщик;
- priority – приоритет задачи (см. раздел 3.2.3.1);
- mode – запуск задачи в привилегированном или непривилегированном режиме;
- stackSize – размер стека задачи в 32-битных словах.

Если планировщик работает в режиме вытесняющей многозадачности (см. 3.2.3.2), при выполнении этого метода происходит немедленное переключение контекста (возможно, на добавленную задачу, если нет более приоритетных).

В том варианте метода, где отсутствует параметр mode, подразумевается непривилегированный режим[1)].

Пример:

```
void MutexTestApp::Initialize()
{
   mutex = new Mutex();
   Task::Add(new PI1Task(), Task::PriorityBelowNormal, 0x50);
   Task::Add(new PI2Task(), Task::PriorityNormal, 0x50);
   Task::Add(new PI3Task(), Task::PriorityHigh, 0x50);
}
```

[1)] Однако, если символ препроцессора MAKS_PROFILING_ENABLED определен как 1, умолчанием становится привилегированный режим. Это связано с текущей реализацией профилирования, работающей только в привилегированном режиме.

3.2.2.3. Метод Execute()

Прототип:

```
virtual void Execute() = 0;
```

Реализация рабочих функций задачи. В отличие от операционных систем, базирующихся на процедурно-ориентированном подходе, не требуется никаких аргументов. Это связано с тем, что метод класса может использовать сколь угодно много переменных-членов, которые инициализируются в конструкторе задачи.

Пример:

```
void Task1::Execute()
{
   puts("Task1 is running");
   while (true)
   {
         puts("Task2 is running");
         Task* task = new Task2();
         Add(task, Task::PriorityNormal, 0x50);
         Delay(700);
         puts("Task2 delete");
         task->Delete();
   }
}
```

3.2.2.4. Метод Delete()

Прототип:

```
Result Delete();
```

Удаляет задачу из планировщика. Если задача удаляет сама себя, происходит немедленное переключение контекста.

Пример: см. пример к методу Execute() выше.

3.2.2.5. Метод Delay()

Прототип:

```
static Result Delay(uint32_t timeoutMs);
```

Выполняет задержку на величину timeoutMs, блокируя задачу.

Пример: см. пример к методу Execute() выше.

3.2.2.6. Метод CpuDelay()

Прототип:

```
static void CpuDelay(uint32_t timeoutMs);
```

Выполняет задержку на величину timeoutMs, не блокируя задачу, но расходуя процессорное время. Может быть использована там, где переключение контекста не нужно или невозможно, например, при кооперативной многозадачности (см. 3.2.3.2) или отладке.

3.2.2.7. Метод Yield()

Прототип:

```
static void Yield();
```

Используется для передачи управления следующей задаче. В первую очередь, данная функция важна при кооперативной многозадачности (см. 3.2.3.2), так как в этом случае нет переключений по квантам времени. Но и при вытесняющей многозадачности, если известно, что задача пока больше не нужна, она может отдать остаток кванта времени другим задачам, вызвав данную функцию.

Пример:

```
void CooperativeTask::Execute()
{
  while (true) {
    printf("Stop running\n");
    Yield();   // перейдем к следующей строке, когда другая задача выполнит Yield()
    printf("Running again\n");
  }
}
```

3.2.2.8. Метод SetPriority()

Прототип:

```
Result SetPriority(Priority value);
```

Устанавливает приоритет задачи. При вытесняющей многозадачности (см. 3.2.3.2) следствием может быть немедленное переключение контекста в данной точке, не дожидаясь окончания текущего кванта времени.

3.2.2.9. Метод GetName()

Прототип:

```
const char* GetName() const;
```

Возвращает имя задачи.

3.2.2.10. Метод GetPriority()

Прототип:

```
Priority GetPriority() const;
```

Возвращает текущий приоритет задачи.

3.2.2.11. Метод GetCurrent()

Прототип:

```
static Task* GetCurrent();
```

Возвращает указатель на текущую выполняемую задачу.

3.2.2.12. Метод GetState()

Прототип:

```
State GetState() const;
```

Возвращает текущее состояние задачи. См. 3.2.3.1.

3.2.3. Планировщик задач (класс Scheduler)

3.2.3.1. Приоритеты и состояния задач

В текущей версии ОСРВ МАКС допускаются следующие приоритеты для задач (перечисление Task::Priority):

```
enum Priority
{
	PriorityIdle = 0,
	PriorityLow = 1,
	PriorityBelowNormal = 2,
	PriorityNormal = 3,
	PriorityAboveNormal = 4,
	PriorityHigh = 5,
	PriorityRealtime = 6
};
```

Задача может находиться в следующих состояниях (перечисление Task::State):

```
enum State
{
	/// Задача готова к выполнению
	StateReady,
	/// Задача выполняется
	StateRunning,
	/// Задача заблокирована, ждёт события синхронизации или
	/// наступления таймаута
	StateBlocked,
	/// Задача ещё не добавлена в планировщик
	StateInactive
};
```

3.2.3.2. Вытесняющая и кооперативная многозадачность

Планировщик ОС поддерживает два типа многозадачности: вытесняющую и кооперативную. При вытесняющей на исполнение подаются только задачи с максимальным приоритетом. Допустим, в системе имеется 4 задачи со следующими приоритетами:

Имя задачи	**Приоритет**
Task1	`PriorityNormal`
Task2	`PriorityLow`
Task3	`PriorityNormal`
Task4	`PriorityLow`

В таком случае, исполняться будут только задачи Task1 и Task3, остальные же не будут получать квантов времени на исполнение. Так будет продолжаться, пока задачи Task1 и Task3 не перейдут в заблокированное состояние (ожидая объект синхронизации, исполняя задержку по таймеру и т.п.). Именно тогда начнётся исполнение задач Task2 и Task4. Как только любая из задач Task1 или Task3 выйдет из заблокированного состояния, задачи Task2 и Task4 снова перестанут получать кванты времени[1)].

В первую очередь, такой подход удобен для реализации отложенной обработки прерываний. Задачи, производящие такую обработку, имеют высокий приоритет, но почти всё время находятся в заблокированном состоянии. Обработчик прерывания снимает блокировку, после чего завершается (чтобы не маскировать другие прерывания). Задача отложенной обработки выполняет какие-либо действия, после чего снова блокируется, давая возможность работы основным задачам.

Кооперативная многозадачность отличается от вытесняющей тем, что планировщик самостоятельно не может прервать выполнение текущей задачи, даже если появилась готовая к выполнению задача с более высоким приоритетом. Каждая задача должна самостоятельно передать управление планировщику (явным вызовом). Таким образом, высокоприоритетная задача будет ожидать, пока низкоприоритетная завершит свою работу и отдаст управление планировщику.

Тип многозадачности жестко задан в конструкторе планировщика, для его смены следует модифицировать код (см. 4.2).

3.2.3.3. Метод GetInstance()

Прототип:

```
static Scheduler& GetInstance();
```

[1)] «Как только» здесь означает «не дожидаясь завершения кванта времени». Иногда такую многозадачность называют гибридной, в отличие от «чистой» вытесняющей, где переключение всегда происходит только по завершении кванта времени.

Возвращает ссылку на объект планировщика ОС для того, чтобы использовать его функционал. Примеры использования приведены ниже.

3.2.3.4. Метод GetTickCount()

Прототип:

```
uint32_t GetTickCount() const;
```

Возвращает время работы системы в системных тиках. В текущей реализации один системный тик равен одной миллисекунде (см. также 4.1).

Пример:

```
GUI_TIMER_TIME GUI_X_GetTime(void)
{
   return maks::Scheduler::GetInstance().GetTickCount();
}
```

3.3. **Классы для синхронизации задач**

3.3.1. Семафоры (класс Semaphore)

3.3.1.1. Конструктор класса

Прототип:

```
Semaphore(size_t startCount, size_t maxCount);
```

Создаёт семафор

- startCount – начальное значение счётчика семафора;
- maxCount – максимально возможное количество, до которого может увеличиться значение счетчика семафора.

3.3.1.2. Метод Wait()

Прототип:

```
Result Wait(uint32_t timeoutMs = INFINITE_TIMEOUT);
```

Блокирует задачу до тех пор, пока значение счётчика семафора не станет отличным от нуля. После чего уменьшает значение счётчика и разблокирует задачу.

timeoutMs - Таймаут, при истечении которого задача разблокируется.

Если значение таймаута равно нулю, то метод вернёт управление мгновенно, без попытки блокировки, поэтому с данным значением параметра метод может вызываться также и из обработчиков прерываний.

Пример:

```
Result res = semaphore->Wait(200);
if (res == ResultOk)
{
```

```
      mutex->Lock();
      printf("After      Wait:   %s,   count   =   %d\r\n",   GetName(),   semaphore-
>GetCurrentCount());
      mutex->Unlock();
   }
   else if (res == ResultTimeout)
   {
      mutex->Lock();
      printf("Timeout: %s\r\n", GetName());
      mutex->Unlock();
   }
   else
   {
      mutex->Lock();
      printf("Something went wrong: %s\r\n", GetName());
      mutex->Unlock();
   }
```

3.3.1.3. Метод Signal()

Прототип:

```
Result Signal();
```

Увеличивает значение счётчика семафора на 1. При этом возможна разблокировка одной из задач, выполнявших для данного семафора метод Wait().

Пример:

```
void ResEmiterTask::Execute()
{
   while (true)
   {
         mutex->Lock();
         printf("Before Signal, count = %d\r\n", semaphore->GetCurrentCount());
         mutex->Unlock();

         semaphore->Signal();

         mutex->Lock();
         printf("After  Signal, count = %d\r\n", semaphore->GetCurrentCount());
         mutex->Unlock();

         Delay(1000);
   }
}
```

3.3.1.4. Метод GetCurrentCount()

Прототип:

```
size_t GetCurrentCount() const;
```

Возвращает текущее значение счётчика семафора

Пример приведён в описании метода Signal().

3.3.1.5. Метод GetMaxCount()

Прототип:

```
size_t GetMaxCount() const;
```

Возвращает заданное при создании семафора максимально допустимое значение счётчика.

3.3.2. Мьютексы (класс Mutex)

Мьютекс (MUTualEXclusive) – это объект синхронизации, которым в каждый момент времени может владеть только один владелец. Все остальные должны ждать, когда мьютекс освободится. Мьютексы нерекурсивны: при попытке захвата мьютекса, который уже захвачен данной задачей, будет выдана ошибка ResultErrorInvalidState.

3.3.2.1. Метод Lock()

Прототип:

```
Result Lock(uint32_t timeoutMs = INFINITE_TIMEOUT);
```

Захватывает мьютекс.

timeoutMs - Величина таймаута, при превышении которой вернётся код ошибки. 0 означает мгновенную попытку захвата без блокировки задачи.

Пример:

```
voidSyncedPrintTask::Execute()
{
   while (true)
   {
         Result res = mutex->Lock(200);
         if (res == ResultOk)
         {
               printf("Begin %s\r\n", GetName());
               Delay(500);
               printf("%s\r\n", _message);
               Delay(500);
               printf("End %s\r\n", GetName());
               mutex->Unlock();
         }
         else if (res == ResultTimeout)
```

```
            {
                  printf("Timeout %s\r\n", GetName());
                  Delay(100);
            }
            else
            {
                  printf("Something went wrong: %s\r\n", GetName());
            }
      }
}
```

3.3.2.2. Метод Unlock()

Прототип:

```
Result Unlock();
```

Снимает блокировку с мьютекса. Пример использования приведён в описании метода Lock().

3.3.2.3. Метод IsLocked()

Прототип:

```
bool IsLocked() const;
```

Возвращает текущее состояние мьютекса, не блокируя задачу.

3.3.3. События (класс Event)

Семафоры и мьютексы обычно используются для того, чтобы исключить конкуренцию задач при использовании тех или иных ресурсов, но иногда просто следует заблокировать задачу до тех пор, пока не возникнет какое-либо условие для её разблокировки. Типовой сценарий – произошло прерывание и функция-обработчик сигнализирует высокоприоритетной задаче отложенной обработки, что следует пробудиться и выполнить какие-то действия. События позволяют реализовать данный механизм.

3.3.3.1. Конструктор класса

Прототип:

```
Event(bool broadcast = true);
```

Параметр broadcast задает правило, по которому информируются получатели, ждущие возникновения события. true – будут разблокированы все задачи, ожидающие его, false – будет разблокирована только одна задача, которая находится первой в очереди ожидающих этого события. Пример см. в описании метода Raise().

3.3.3.2. Метод Raise()

Прототип:

```
Result Raise();
```

Задачи (или одна из задач), которые ожидают событие, будут разблокированы.

Пример:

```
Event* e = new Event();
e->Raise();
```

3.3.3.3. Метод Wait()

Прототип:

```
Result Wait(uint32_t timeoutMs= INFINITE_TIMEOUT);
```

Блокирует задачу до получения события (либо до истечения таймаута)

timeoutMs - таймаут ожидания в миллисекундах.

Пример:

```
Result res = e->Wait(600);
if (res == ResultOk)
{
   mutex->Lock();
   printf("Event has been dispatched! [task:%s]\r\n", GetName());
   mutex->Unlock();
}
else if (res == ResultTimeout)
{
   mutex->Lock();
   printf("Event receiver timeout! [task:%s]\r\n", GetName());
   mutex->Unlock();
}
else
{
   mutex->Lock();
   printf("Event receiver error %d! [task:%s]\r\n", res, GetName());
   mutex->Unlock();
}
```

3.3.4. Мьютекс-страж (класс MutexGuard)

Объект данного класса создается на основе объекта-мьютекса путем объявления локальной переменной. При этом, в точке объявления этой переменной мьютекс захватывается, а освобождается он, когда происходит выход из области действия переменной (вызывается

деструктор MutexGuard). Это позволяет не заботиться об освобождении мьютекса при сложных ветвлениях в программе, когда данное освобождение может произойти во многих точках.

Пример:

```
static Mutex m;
…
void resourceAccessRoutine() {
   MutexGuard guard(m);          // эквивалентно m.Lock()
   …
   if (someCondition)
         return;           // здесь пришлось бы вызывать m.Unlock()
   …
   if (anotherCondition)
         return;           // и здесь пришлось бы вызывать m.Unlock()
   …
}
```

3.3.4.1. Конструктор класса

Прототип:

```
explicit MutexGuard(Mutex& mutex, bool onlyUnlock = false);
```

Параметры:

- mutex – ссылка на мьютекс;
- onlyUnlock – если значение – истина, захват мьютекса в конструкторе не производится. Подразумевается, что мьютекс уже был захвачен ранее явным вызовом метода Lock().

3.3.5. Бинарный семафор (класс BinarySemaphore)

Класс представляет специальный тип семафора, у которого максимально возможное значение счетчика – единица. Ниже описан конструктор класса. В остальном класс не отличается от своего родителя Semaphore.

3.3.5.1. Конструктор класса

Прототип:

```
explicit BinarySemaphore(bool isEmpty = true);
```

isEmpty – если параметр – истина, начальное значение счетчика семафора – 0, в противном случае – 1.

3.4. **Классы для взаимодействия между задачами в пределах узла**

3.4.1. Очередь сообщений (класс MessageQueue)

Очереди сообщений наиболее эффективны при конвейерной обработке данных. Классический пример – одна задача принимает данные из последовательного порта, производит их декодирование и предобработку, а затем пересылает упакованные сообщения во внутреннем формате для обработки в другой задаче. Чтобы не блокировать свою работу, посылающая задача просто ставит обработанное сообщение в очередь и приступает к дальнейшей работе с последовательным портом.

Для унификации обрабатываемых данных, класс очереди сообщений выполнен в виде шаблона.

3.4.1.1. Конструктор класса

Прототип:

```
MessageQueue(size_t maxSize);
```

maxSize - максимальный размер очереди в элементах данных

Пример:

```
voidMessageQueueTestApp::Initialize()
{
   mQueue = new MessageQueue<short>(5);

   Task::Add(new MessageSenderTask("send"), Task::PriorityNormal, 0x50);
   Task::Add(new MessageReceiverTask("receive"), Task::PriorityNormal, 0x50);
}
```

3.4.1.2. Метод Push()

Прототип:

```
Result Push(const T& message, uint32_t timeoutMs= INFINITE_TIMEOUT);
```

Помещает сообщение в конец очереди.

- message – тело сообщения;
- timeoutMs – если очередь переполнена, задача будет заблокирована до появления свободного места, либо до истечения таймаута.

Пример:

```
voidMessageSenderTask::Execute()
{
   short x = 0;
   while (true)
   {
         Result res = mQueue->Push (++x, 300);
         if (res == ResultOk)
         {
```

```
            printf("Send: %d\r\n", x);
            Delay(200);
        }
        else if (res == ResultTimeout)
        {
            printf("Timeout send: %d \r\n", x);
        }
        else
        {
            printf("Send: something went wrong: %s\r\n", GetName());
        }
    }
}
```

3.4.1.3. Метод PushFront()

Прототип:

```
Result PushFront(const T& message, uint32_t timeoutMs= INFINITE_TIMEOUT);
```

Полностью идентична функции Push, но помещает сообщение в начало очереди.

3.4.1.4. Метод Pop()

Прототип:

```
Result Pop(T& message, uint32_t timeoutMs= INFINITE_TIMEOUT);
```

Извлекает сообщение из начала очереди.

- message – ссылка на переменную для извлечения;
- timeoutMs – таймаут. Если за указанное время в очереди не появится ни одного сообщения, будет возвращена ошибка.

Пример:

```
voidMessageReceiverTask::Execute()
{
    while (true)
    {
        short x;
        Result res = mQueue->Pop(x);
        if (res == ResultOk)
        {
            printf("Receive: %d [count:%d]\r\n", x, mQueue->Count());
            Delay(1000);
        }
        else if (res == ResultTimeout)
        {
            printf("Timeout receive: [count:%d]\r\n", mQueue->Count());
```

```
            }
            else
            {
                  printf("Receive: something went wrong: %s\r\n", GetName());
            }
      }
}
```

3.4.1.5. Метод Peek()

Прототип:

```
Result Peek(T& message, uint32_t timeoutMs= INFINITE_TIMEOUT);
```

Полностью идентична функции Pop(), но не удаляет сообщение из очереди.

3.4.1.6. Метод Count()

Прототип:

```
size_t Count() const;
```

Возвращает текущее количество элементов, помещённых в очередь.

Пример:

```
printf("Receive: %d [count:%d]\r\n", x, mQueue->Count());
```

3.4.1.7. Метод GetMaxSize()

Прототип:

```
size_tGetMaxSize() const;
```

Возвращает заданный при создании максимально допустимый размер очереди.

3.5. **Прочее**

3.5.1. Профилирование

Профилирование кода приложения может быть выполнено при помощи описанных ниже макросов. Если макрос MAKS_PROFILING_ENABLED определен как 0, эти макросы не увеличивают размер бинарного кода. Чтобы задействовать профилирование, необходимо:

1) определить макро MAKS_PROFILING_ENABLED как описано в 4.4;
2) добавить новые значения в enum PROF_EYE, определенный в файле Profiler.h;
3) добавить вызовы профилирования в код приложения.

Примеры профилирования можно посмотреть в функции настройки профайлера ProfEye::Tune() (файл Profiler.cpp), а также в демонстрационных приложениях.

Есть два режима работы профайлера:

1) от точки до точки. Для этого следует определить объект при помощи макроса PROF_DECL(**eye**, **name**), где **eye** – это ранее добавленное в enum PROF_EYE значение, a name

– любой уникальный идентификатор. После чего макрос PROF_START(**name**) запускает отсчет времени, а PROF_STOP(**name**) останавливает;

2) время нахождения в текущей области видимости автоматической переменной. Для этого служит макрос PROF_EYE(**eye**, **name**), аналогичный макросу PROF_DECL(**eye**, **name**). Разница заключается в том, что отсчет времени начинается немедленно, а заканчивается, когда автоматически уничтожается созданный макросом объект.

Вызовы могут быть вложенными, в этом случае время выполнения внутренней секции вычитается из времени внешней.

Измеренные интервалы времени группируются по разделам, соответствующим значениям в enum PROF_EYE и формируют статистику, включающую в себя:

1) полное время выполнения (сумма всех временных интервалов для группы);
2) чистое время выполнения (полное время минус время выполнения вложенных интервалов);
3) среднее время выполнения (учитываются все интервалы);
4) минимальное и максимальное время выполнения;
5) среднеквадратичное отклонение.

Доступ к статистике производится через функцию ProfEye::PrintResults().

Следует отметить, что включение вызовов профилировщика в часто исполняемый код может существенно замедлить скорость работы приложения, однако, поскольку профайлер на этапе автоматической настройки измеряет задержки, вызванные собственной работой и учитывает их при измерении времени, точность не страдает. Время измеряется в циклах процессора, причем реально достижимая точность составляет порядка одного цикла.

Профилирование возможно только если профилируемая задача имеет привилегированный доступ. Поэтому определение макроса MAKS_PROFILING_ENABLED как 1 включает запуск всех задач с привилегированным доступом.

4. СИСТЕМНЫЕ ОПЦИИ И ИХ ПЕРЕКЛЮЧЕНИЕ

4.1. **Системный квант времени**

Системный квант времени – это период времени, выделяемый задаче на выполнение. По его истечении планировщик задач проверяет, не нужно ли отдать процессорное время другой задаче. Если существует готовая к выполнению другая задача с равным или большим приоритетом, происходит переключение контекста (процессорное время отдается этой задаче). Длительность кванта времени определяет время отклика системы на события. По умолчанию длительность системного кванта составляет 1 миллисекунду.

Изменение опции делается путем модификации члена класса Scheduler в файле MaksScheduler.h:

```
static const uint32_t TICK_RATE_HZ = 1000;  // 1000 квантов в секунду, квант 1 мс
```

4.2. **Тип многозадачности**

Планировщик ОС может работать либо в режиме вытесняющей многозадачности, либо в режиме кооперативной (см. 3.2.3.2). Переключение опции делается путем модификации кода конструктора класса Scheduler в файле MaksScheduler.cpp:

```
Scheduler::Scheduler() :
      _currentTask(nullptr),
      _tickCount(0),
      _initialized(false),
      _started(false),
      _usePreemption(true)  // false для кооперативной многозадачности
```

4.3. **Профилирование**

4.3.1. В ОС предусмотрено средство сбора информации по быстродействию операций ядра. При желании, программист может встроить профилирование и в код приложения (как это делается, описано в 3.5.1). В текущей версии профилирование связано с режимом выполнения задач, которое может происходить с привилегированным доступом либо с непривилегированным (см. privileged access в спецификациях процессора). В режиме без привилегированного доступа профилирование работать не будет. Включение/выключение привилегированного доступа для задач описано ниже.

4.4. **Выполнение задач с привилегированным доступом**

По умолчанию все задачи приложения выполняются либо с непривилегированным доступом. Для запуска задачи с привилегированным доступом нужно явно указать параметр mode в методе Task::Add() (см. 3.2.2.2). Однако, если символ препроцессора MAKS_PROFILING_ENABLED определен как 1, режимом по умолчанию становится привилегированный доступ, и для запуска в непривилегированном режиме нужно указывать параметр mode явно. Это связано с текущей

реализацией профилирования, работающей только в привилегированном режиме. Ниже описано, как определить символ MAKS_PROFILING_ENABLED.

4.4.1. Управление режимом доступа задач по умолчанию

Определение символа MAKS_PROFILING_ENABLED производится в файле MaksTunes.h. Изначально этот символ определен как 0. В этом случае, задача будет запущена в непривилегированном режиме, если режим не указан явно в методе Task::Add(). Если изменить файл MaksTunes.h так, чтобы MAKS_PROFILING_ENABLED было 1, задача будет запущена в привилегированном режиме, когда режим не указан явно.

5. ОБРАБОТКА ПРЕРЫВАНИЙ

5.1. **Общие сведения**

5.1.1. Непосредственно для нужд ОСРВ МАКС используются прерывания, перечисленные в табл. 2:

Т а б л и ц а 2

Тип прерывания	Номер прерывания	Приоритет	Использование
SYSTICK	15	255	отсчет системных квантов времени
PendSV	14	255	смена контекста – переключение с одной пользовательской задачи на другую
SVC	11	0	вызов системных функций, требующих привилегированного режима, из непривилегированного режима
Reset	1	-3	инициализация ОС, создание задач пользователя, запуск планировщика задач

5.1.2. Изменение обработчиков и/или приоритетов этих прерываний не рекомендуется, т.к. это фактически означает изменения в ядре ОС. Прочие прерывания могут быть задействованы для прикладных целей и код для их обработки пишет разработчик приложения, как это описано ниже.

5.1.3. Особенностью обработчиков прерываний является то, что в них нельзя выделять или освобождать память кучи (например, операторами new/delete). Такое ограничение неприемлемо для хоть сколько-нибудь сложной логики приложения, и чтобы его обойти, применяется подход, описанный ниже.

5.1.4. В ОСРВ МАКС разработан так называемый универсальный пользовательский обработчик прерывания (см. ниже). Он переводит в состояние готовности специальные пользовательские задачи обработки прерываний (см. ниже) и завершает работу. Далее следует переключение контекста на задачу обработки прерывания, при условии, что ее приоритет достаточно высок (приоритет задачи должен быть подобран с учетом важности быстрой обработки данного прерывания). Задача производит необходимые содержательные действия по обработке прерывания, после чего блокируется до следующего прерывания. В теле этой задачи никаких запретов на действия с кучей не накладывается. Если логика обработки прерывания требует изменения системных регистров, задача обработки прерывания может быть запущена в привилегированном режиме.

5.1.5. Предположим, в системе необходимо обрабатывать некое прерывание типа IRQ с номером n (лежащим в диапазоне 16-255). Для этого программисту необходимо предпринять следующие шаги:

1) назначить прерыванию n нужный приоритет. Нужный код вносится в метод Initialize() класса приложения (производного от Application);
2) отредактировать вектор обработчиков прерываний в файле startup_stm32f1249xx.s, сопоставив прерыванию n так называемый универсальный пользовательский обработчик прерывания (см. ниже);
3) написать код для задачи обработки прерывания (см. ниже). Добавить запуск задачи обработки прерывания в метод Initialize() класса приложения (производного от Application).

5.2. **Универсальный пользовательский обработчик прерывания**

5.2.1. Код этого обработчика написан заранее и менять его не требуется. Это функция MaksIrqHandler() в файле MaksScheduler.cpp. Вектор прерывания может быть отредактирован следующим образом: в файле startup_stm32f429xx.s у метки нужного прерывания поставить безусловный переход на MaksIrqHandler

```
DMA2D_IRQHandler
                B       MaksIrqHandler ; было B .
```

Переключение на задачу-обработчик прерывания происходит почти со скоростью переключения контекста. Если даже такая задержка неприемлема, можно использовать двухуровневую схему. При этом разработчик пишет свой обычный обработчик прерывания, выполняет в нем критические по времени действия, а в конце вызывает функцию MaksIrqHandler(), которая активизирует задачу ОС.

5.3. **Задача обработки прерывания**

5.3.1. Класс, реализующий задачу обработки прерывания, должен быть потомком не непосредственно класса Task, а класса TaskIrq (производного от Task). Экземпляр такого класса добавляется в планировщик специальным методом TaskIrq::Add(), среди параметров которого есть номер прерывания. Пользователь должен переопределить в данной задаче виртуальный метод IrqHandler(), реализующий обработку прерывания. Метод Execute у класса TaskIrq не является виртуальным и не переопределяется. При добавлении в планировщик задача автоматически блокируется и пробуждается при возникновении прерывания.

5.3.2. Пример: задача для наибыстрейшей обработки прерывания с номером 27 (DMA1_Stream0_IRQn), требующая привилегированных действий.

```
class MyIrqHandler : TaskIrq
{
```

```
    void IrqHandler()
    {
          // здесь содержательная обработка прерывания
    }
}
…
void MyApplication::Initialize()
{
    …
    TaskIrq::Add(new MyIrqHandler(), DMA1_Stream0_IRQn, Task::PriorityRealTime,
                    Task::ModePrivileged, 0x100);
    …
}
```

6. ОПИСАНИЕ ДРАЙВЕРОВ УСТРОЙСТВ

6.1. **Драйвер экрана (дисплея)**

6.1.1. Общие сведения

Драйвер экрана реализует класс GraphDriver. Для его использования необходимо включение в код файла MaksScreen.h. В коде ОС определен (единственный) экземпляр этого класса lcd_graph_driver. Для работы с драйвером следует вызывать для этого экземпляра методы, описанные ниже.

В методах используются как аппаратно-независимые цвета и шрифты (типы COLOR, FONT), так и их аппаратно-зависимые представления (DeviceColor, DeviceFont). Последние предназначены, в основном, для ситуаций, когда требуется запомнить текущий цвет (шрифт), а позднее восстановить его или, например, инвертировать.

Координаты на экране указываются в пикселях, нумеруемых с нуля. Координаты растут вправо и вниз, т.е. (0;0) – верхний левый угол, (0; M) – верхний правый, где M – максимально возможная координата по горизонтали.

В тех случаях, когда в методах употребляется текстовый символ типа char, его числовое значение представляет собой индекс в таблице данного шрифта. Индекс 0 на экране не отображается. Индексы 1-127 отображаются символами ASCII. Отображение индексов 128 – 255 зависит от типа контроллера и аппаратной конфигурации дисплея.

6.1.2. Методы драйвера

6.1.2.1. Метод Init()

Прототип:

```
virtual bool Init();
```

Инициализирует драйвер. Возвращает истину при успехе.

6.1.2.2. Метод DeInit()

Прототип:

```
virtual bool DeInit();
```

Освобождает ресурсы драйвера. Возвращает истину при успехе.

6.1.2.3. Метод IsReady()

Прототип:

```
virtual bool IsReady() const;
```

Возвращает истину, если драйвер был инициализирован и готов к работе.

6.1.2.4. Метод DevColor()

Прототип:

```
virtual DeviceColor DevColor(COLOR);
```

Параметр – константа из набора стандартных цветов, например, COLOR_CYAN.

Возвращает соответствующий цвет в представлении конкретного устройства (дисплея).

6.1.2.5. Метод InvertColor()

Прототип:

```
virtual DeviceColor InvertColor(DeviceColor);
```

Параметр – цвет в представлении конкретного устройства (дисплея).

Возвращает цвет устройства, соответствующий инверсии данного цвета.

6.1.2.6. Метод DevFont()

Прототип:

```
virtual DeviceFont DevFont(FONT);
```

Параметр – константа из набора стандартных шрифтов, например, FONT_12.

Возвращает шрифт в представлении конкретного устройства (дисплея).

6.1.2.7. Метод GetXSize()

Прототип:

```
virtual ushort GetXSize() const;
```

Возвращает размер экрана в пикселях по горизонтали.

6.1.2.8. Метод GetYSize()

Прототип:

```
virtual ushort GetYSize() const;
```

Возвращает размер экрана в пикселях по вертикали.

6.1.2.9. Метод GetForeColor()

Прототип:

```
virtual COLOR GetForeColor() const;
```

Возвращает текущий цвет, используемый при рисовании.

6.1.2.10. Метод GetDevForeColor()

Прототип:

```
virtual DeviceColor GetDevForeColor() const;
```

Возвращает текущий цвет, используемый при рисовании, в представлении конкретного устройства (дисплея).

6.1.2.11. Метод GetBackColor()

Прототип:

```
virtual COLOR GetBackColor() const;
```

Возвращает текущий цвет, используемый для фона.

6.1.2.12. Метод GetDevBackColor()

Прототип:

```
virtual DeviceColor GetDevBackColor() const;
```

Возвращает текущий цвет, используемый для фона, в представлении конкретного устройства (дисплея).

6.1.2.13. Метод GetCurFont()

Прототип:

```
virtual FONT GetCurFont() const;
```

Возвращает текущий шрифт.

6.1.2.14. Метод GetCurDevFont()

Прототип:

```
virtual DeviceFont GetCurDevFont() const;
```

Возвращает текущий шрифт в представлении конкретного устройства (дисплея).

6.1.2.15. Метод SetForeColor()

Прототип:

```
bool SetForeColor(COLOR color);
```

Устанавливает цвет для рисования.

Параметр - цвет, который будет использоваться при рисовании.

Возвращает истину при успехе.

6.1.2.16. Метод SetBackColor()

Прототип:

```
bool SetBackColor(COLOR color);
```

Устанавливает цвет фона.

Параметр – цвет, который будет использоваться для фона.

Возвращает истину при успехе.

6.1.2.17. Метод SetCurFont()

Прототип:

```
virtual bool SetCurFont(FONT font);
```

Устанавливает текущий шрифт.

Параметр – шрифт, который будет использоваться для вывода текста.

Возвращает истину при успехе.

6.1.2.18. Метод ReadPixel()

Прототип:

```
virtual DeviceColor ReadPixel(int x, int y);
```

Считать цвет пикселя по заданным координатам.

Параметры:

x – координата по горизонтали

y - координата по вертикали

Возвращает цвет в представлении конкретного устройства (дисплея).

6.1.2.19. Метод DrawPixel()

Прототип:

```
virtual bool DrawPixel(int x, int y, COLOR color);
```

Нарисовать пиксель.

Параметры:

x – координата по горизонтали

y - координата по вертикали

color – цвет пикселя

Возвращает истину при успехе.

6.1.2.20. Метод DrawHLine()

Прототип:

```
virtual bool DrawHLine(int x, int y, int len);
```

Нарисовать горизонтальную линию.

Параметры:

x – координата начала линии по горизонтали

y - координата начала линии по вертикали

len – длина линии в пикселях.

Координата по горизонтали будет изменяться от x до (x + len – 1)

Возвращает истину при успехе.

6.1.2.21. Метод DrawVLine()

Прототип:

```
virtual bool DrawVLine(int x, int y, int len);
```

Нарисовать вертикальную линию.

Параметры:

x – координата начала линии по горизонтали

y - координата начала линии по вертикали

len – длина линии в пикселях.

Координата по вертикали будет изменяться от y до (y + len – 1)

Возвращает истину при успехе.

6.1.2.22. Метод DrawLine()

Прототип:

```
virtual bool DrawLine(int x1, int y1, int x2, int y2);
```

Нарисовать линию.

Параметры:

x1 – координата начала линии по горизонтали

y1 - координата начала линии по вертикали

x2 – координата конца линии по горизонтали

y2 - координата конца линии по вертикали

Возвращает истину при успехе.

6.1.2.23. Метод DrawRect()

Прототип:

```
bool DrawRect(int x, int y, int width, int height);
```

Нарисовать прямоугольник текущим цветом для рисования (устанавливается при помощи SetForeColor()).

Параметры:

x – координата угла прямоугольника по горизонтали

y - координата угла прямоугольника по вертикали

width – ширина прямоугольника (размер по горизонтали)

height – высота прямоугольника (размер по вертикали)

Указывается угол прямоугольника с наименьшими координатами x и y.

Возвращает истину при успехе.

6.1.2.24. Метод FillRect()

Прототип:

```
bool FillRect(int x, int y, int width, int height);
```

Нарисовать прямоугольник, заполненный текущим цветом для рисования (устанавливается при помощи SetForeColor()). Параметры и возвращаемое значение – как в методе DrawRect().

6.1.2.25. Метод DrawCircle()

Прототип:

```
bool DrawCircle(int x, int y, int rad);
```

Нарисовать окружность текущим цветом для рисования (устанавливается при помощи SetForeColor()).

Параметры:

x – координата центра окружности по горизонтали

y - координата центра окружности по вертикали

rad – радиус окружности в пикселях

Возвращает истину при успехе.

6.1.2.26. Метод FillCircle()

Прототип:

```
bool FillCircle(int x, int y, int rad);
```

Нарисовать круг текущим цветом для рисования (устанавливается при помощи SetForeColor()).

Параметры и возвращаемое значение – как в методе DrawCircle().

6.1.2.27. Метод DrawChar()

Прототип:

```
virtual bool DrawChar(int x, int y, char c);
```

Нарисовать символ цветом для рисования (устанавливается при помощи SetForeColor()).

Параметры:

x – координата начала символа по горизонтали

y - координата начала символа по вертикали

c – индекс символа в таблице шрифта

Под «началом символа» здесь понимается угол прямоугольника, занимаемого символом, с наименьшими координатами, т.е. верхнего левого. Символ будет нарисован правее и ниже указанной точки. При этом фон прямоугольника, заполняемого символом, будет принудительно изменен на текущий цвет фона (устанавливается при помощи SetBackColor()).

Возвращает истину при успехе.

6.1.2.28. Метод DrawText()

Прототип:

```
virtual bool DrawText(int x, int y, CSPTR text, TEXT_ALIGN);
```

Нарисовать текст цветом для рисования (устанавливается при помощи SetForeColor()) на текущем фоне (устанавливается при помощи SetBackColor()).

Параметры:

x – координата опорной точки по горизонтали

y - координата опорной точки по вертикали

text – указатель на строку текста – последовательность байтов, оканчивающуюся нулевым байтом, не входящим в строку.

TEXT_ALIGN – выравнивание текста. Влияет на вывод следующим образом:

- ALIGN_LEFT – текст выводится правее и ниже опорной точки
- ALIGN_CENTER – текст выводится таким образом, что середина верхней стороны занимаемого им прямоугольника совпадает с опорной точкой
- ALIGN_RIGHT – прямоугольник для текста располагается правее и ниже опорной точки и продолжается до правого края экрана. Текст располагается в прямоугольнике выровненным вправо (прижатым к правому краю экрана). Если начало текста выходит за опорную точку слева, оно обрезается

Возвращает истину при успехе.

6.1.2.29. Метод Clear()

Прототип:

```
virtual bool Clear(COLOR fill);
```

Очистить экран.

Параметр – цвет, которым заполняется экран.

Возвращает истину при успехе.

6.1.2.30. Метод Scroll()

Прототип:

```
virtual bool Scroll(int x, int y, int width, int height, int shift, COLOR fill);
```

Выполняет сдвиг содержимого указанного прямоугольника внутри него вверх на указанное расстояние, заполняя пустоту внизу заданным цветом.

Параметры:

x – координата левого верхнего угла прямоугольника по горизонтали

y - координата левого верхнего угла прямоугольника по вертикали

width – ширина прямоугольника

height – высота прямоугольника

shift – количество пикселей по вертикали, которые теряются в верхней части прямоугольника при сдвиге его содержимого вверх. Соответственно, область внизу прямоугольника размером (width x shift) заполняется цветом fill.

fill – см. выше

Возвращает истину при успехе.

6.2. **Драйверы портов**

6.2.1. Общие сведения

В ОСРВ МАКС использована объектно-ориентированная технология для интерфейса работы с портами. При этом все варианты конкретной реализации портов (например, UARTPort) представлены соответствующими классами, которые выводятся из базового класса Port. На уровне базового класса задается простейшая функциональность, которую способен предоставить любой порт. В производных классах могут быть определены методы, специфичные для данной реализации. Таким образом, в тех случаях, когда от порта не требуется сложная работа, в программе можно писать просто Port * port; и использовать любой вариант.

6.2.2. Класс Buf

Представляет собой универсальный буфер, используемый в функциях портов. Имеется реализация как на динамической памяти, так и на статическом массиве.

6.2.2.1. Константа DEF_BUF_SIZE

Задает размер буфера по умолчанию.

6.2.2.2. Метод Ready

Запрос готовности буфера.

Прототип:

```
bool Ready() const;
```

Возвращает истину при положительном ответе.

6.2.2.3. Метод Len

Возвращает размер данных (в байтах), хранящихся в буфере.

Прототип:

```
size_t Len() const;
```

6.2.2.4. Метод Size

Возвращает размер буфера (в байтах).

Прототип:

```
virtual size_t Size() const = 0;
```

6.2.2.5. Метод Alloc

Обеспечивает размер буфера (в байтах) не меньше указанного.

Прототип:

```
virtual bool Alloc(size_t bufsz);
```

Возвращает истину при успехе.

6.2.2.6. Метод AddLen

Увеличивает размер данных в буфере на указанное значение (например, если данные добавлены в конец), но не больше чем до размера буфера.

Прототип:

```
void AddLen(size_t len);
```

6.2.2.7. Метод ShowPtrBeg

Возвращает указатель на начало данных в буфере для доступа только на чтение.

Прототип:

```
virtual const byte * ShowPtrBeg() const = 0;
```

6.2.2.8. Метод ShowPtrEnd

Возвращает указатель на конец данных в буфере (сразу за последним байтом) для доступа только на чтение.

Прототип:

```
const byte * ShowPtrEnd() const;
```

6.2.2.9. Метод UsePtrBeg

Возвращает указатель на начало данных в буфере для доступа на чтение и запись.

Прототип:

```
virtual byte * UsePtrBeg() = 0;
```

6.2.2.10. Метод UsePtrEnd

Возвращает указатель на конец данных в буфере (сразу за последним байтом) для доступа на чтение и запись.

Прототип:

```
byte * UsePtrEnd();
```

6.2.2.11. Метод Give

Отдает указатель на начало данных в буфере. Буфер обнуляется.

Прототип:

```
virtual byte * Give() = 0;
```

6.2.2.12. Метод Copy

Копирует данные в буфер.

Прототип:

```
virtual void Copy(const byte * ptr, size_t len);
```

Параметры:

ptr – указатель на данные

len – размер данных в байтах.

6.2.2.13. Метод Grab

Помещает данные в буфер (при возможности используется переданный указатель без копирования).

Прототип:

```
virtual void Grab(byte * ptr, size_t len);
```

Параметры:

ptr – указатель на данные

len – размер данных в байтах.

6.2.2.14. Метод Copy

Прототип:

```
void Copy(const Buf & buf);
```

Копирует данные из другого буфера.

Параметры:

buf – ссылка на источник

6.2.2.15. Метод Grab

Забирает данные из другого буфера (при возможности без копирования).

Прототип:

```
void Grab(Buf & buf);
```

Параметры:

buf – ссылка на источник

6.2.2.16. Метод Add1B

Записывает один байт в конец буфера с увеличением длины.

Прототип:

```
void Add1B(int8_t val);
```

Параметры:

val – записываемый байт

6.2.2.17. Метод Add2B

Записывает два байта в конец буфера с увеличением длины.

Прототип:

```
void Add2B(int16_t val);
```

Параметры:

val – записываемые байты

6.2.2.18. Метод Add4B

Записывает четыре байта в конец буфера с увеличением длины.

Прототип:

```
void Add4B(int32_t val);
```

Параметры:

val – записываемые байты

6.2.2.19. Метод Pos

Возвращает позицию указателя чтения в буфере (в байтах).

Прототип:

```
int Pos() const;
```

6.2.2.20. Метод Rewind

Устанавливает позицию указателя чтения в буфере (в байтах) на начало.

Прототип:

```
void Rewind();
```

6.2.2.21. Метод Read1B

Считывает один байт с текущей позиции и смещает указатель чтения. Возвращает считанное значение.

Прототип:

```
int8_t Read1B();
```

6.2.2.22. Метод Read2B

Считывает два байта с текущей позиции и смещает указатель чтения. Возвращает считанное значение.

Прототип:

```
int16_t Read2B();
```

6.2.2.23. Метод Read4B

Считывает четыре байта с текущей позиции и смещает указатель чтения. Возвращает считанное значение.

Прототип:

```
int32_t Read4B();
```

6.2.2.24. Метод Add

Дописывает данные в конец буфера.

Прототип:

```
void Add(const byte * ptr, size_t len);
```

Параметры:

ptr – указатель на данные

len – размер данных в байтах.

6.2.2.25. Метод Add

Дописывает данные в конец буфера.

Прототип:

```
void Add(const byte * ptr, size_t len);
```

Параметры:

ptr – указатель на данные

len – размер данных в байтах.

6.2.3. Класс StatBuf

Реализация буфера на статическом массиве. Необходимые виртуальные методы модифицированы или реализованы так, чтобы обеспечить интерфейс класса Buf.

Объявление:

```
template <size_t buf_size = Buf::DEF_BUF_SIZE>
   class StatBuf : public Buf
```

Существует объявленный тип

```
typedef StatBuf<Buf::DEF_BUF_SIZE> DefStatBuf;
```

для буфера с размером по умолчанию.

6.2.4. Класс DynBuf

Реализация буфера на динамической памяти. Необходимые виртуальные методы модифицированы или реализованы так, чтобы обеспечить интерфейс класса Buf.

Объявление:

```
class DynBuf : public Buf
```

6.2.5. Класс Port

6.2.5.1. Перечисление MODE

Используется при открытии порта. Задает режим его работы.

`PM_READ_ALLOWED` - разрешено чтение из порта;

`PM_WRITE_ALLOWED` - разрешена запись в порт.

6.2.5.2. Перечисление STATE

Описывает состояние порта:

`PS_OPENED` - порт открыт;

`PS_READ_ALLOWED` - разрешено чтение из порта;

`PS_WRITE_ALLOWED` - разрешена запись в порт;

`PS_READ_BUSY` - порт сейчас обрабатывает операцию чтения;

`PS_WRITE_BUSY` - порт сейчас обрабатывает операцию записи;

`PS_DATA_READY` - данные приняты и готовы к выдаче.

6.2.5.3. Конструктор

```
Port(MODE mode = PM_RW_ALLOWED)
```

Создает объект порт и при необходимости открывает его в указанном режиме.

6.2.5.4. Метод Mode

Возвращает битовую маску режима работы порта

Прототип:

```
BitMask<MODE> Mode();
```

6.2.5.5. Метод State

Возвращает битовую маску состояния порта

Прототип:

```
BitMask<STATE> State();
```

6.2.5.6. Метод Open

Открывает порт в указанном режиме.

Прототип:

```
virtual bool Open(MODE mode = PM_RW_ALLOWED);
```

Возвращает истину при успехе.

6.2.5.7. Метод Close

Закрывает порт.

Прототип:

```
virtual bool Close();
```

Возвращает истину при успехе.

6.2.5.8. Метод IsOpened

Прототип:

```
bool IsOpened() const;
```

Возвращает истину если порт открыт.

6.2.5.9. Метод MayRead

Прототип:

```
bool MayRead() const;
```

Возвращает истину если порт допускает чтение данных.

6.2.5.10. Метод MayWrite

Прототип:

```
bool MayWrite() const;
```

Возвращает истину если порт допускает запись данных.

6.2.5.11. Метод Send

Посылает данные в порт.

Прототип:

```
virtual bool Send(const byte * ptr, size_t len) = 0;
```

Параметры:

ptr – указатель на данные

len – размер данных в байтах.

Возвращает истину при успехе.

6.2.5.12. Метод Require

Запрашивает данные из порта. Следующая операция чтения должна вернуть указанное количество байт.

Прототип:

```
virtual bool Require(size_t len);
```

Параметры:

len – размер данных в байтах.

Возвращает истину при успехе.

6.2.6. Класс BufferedPort

Порт с буфером. Производный класс от Port.

Объявление:

```
template<typename buf_type>
class BufferedPort : public Port
```

6.2.6.1. Константа DEF_BUF_SIZE

Задает размер буфера в байтах по умолчанию.

6.2.6.2. Конструкторы

```
BufferedPort(MODE mode = PM_RW_ALLOWED);
```

Создает объект BufferedPort с размером буфера установленным по умолчанию и открывает порт в указанном режиме.

```
BufferedPort(size_t bufsz);
```

Создает объект BufferedPort с указанным размером буфера и открывает порт в режиме по умолчанию.

```
BufferedPort(MODE mode, size_t bufsz);
```

Создает объект BufferedPort с указанным размером буфера и открывает порт в указанном режиме.

6.2.6.3. Метод Open

Открывает порт в указанном режиме и устанавливает размер буфера.

Прототип:

```
bool Open(MODE mode = PM_RW_ALLOWED, size_t bufsz = DEF_BUF_SIZE);
```

Возвращает истину при успехе.

6.2.6.4. Метод Close

Закрывает порт.

Прототип:

```
virtual bool Close();
```

Возвращает истину при успехе.

6.2.6.5. Определение типа

```
typedef BufferedPort<DefStatBuf> DefBufferedPort;
```

Задает порт с буфером размера по умолчанию.

6.2.7. Класс PortUART

Порт для интерфейса UART. Производный класс от DefBufferedPort. Прием данных в режиме прерываний осуществляется с помощью специальной задачи ОС, выполняющейся в режиме обработчика прерываний. Определен объект с именем uart_port, заранее подготовленный для доступа к первому (в смысле контроллера) интерфейсу UART.

Объявление:

```
class PortUART : public DefBufferedPort
```

6.2.7.1. Конструктор

```
PortUART(MODE mode = PM_ZERO);
```

Создает объект PortUART и при необходимости открывает его в указанном режиме.

6.2.7.2. Метод Open

```
virtual bool Open(Receive_Handler_Type handler, MODE mode = PM_RW_ALLOWED);
```

Открывает порт в указанном режиме и устанавливает callback функцию для обработки операций приема данных. Эта функция вызывается из обработчика прерываний микроконтроллера, так что в ней не следует выполнять сложную работу.

6.2.7.3. Метод Close

virtual bool Close();

Закрывает порт.

Возвращает истину при успехе.

Остальные методы являются реализацией виртуальных методов базовых классов.

7. ОБРАЩЕНИЕ К ПРОГРАММЕ

7.1. Обращение к программе производится пользовательским приложением с использованием метода Run(), см. 3.2.1.1.

8. ВХОДНЫЕ И ВЫХОДНЫЕ ДАННЫЕ

8.1. Конкретный перечень входных и выходных данных зависит от прикладных задач, решаемых системой реального времени, в составе которой функционирует ОСРВ МАКС. В общем случае входными данными ОСРВ МАКС являются события, данные о которых ОСРВ МАКС получает от аппаратного средства (платы с поддерживаемым микроконтроллером), а выходными данными – вызов обработчиков таких событий. Обработчики событий входят в состав пользовательского приложения.

9. СООБЩЕНИЯ

9.1. Непосредственно ОСРВ МАКС не формирует и не отображает сообщений программисту. В процессе разработки пользовательского приложения и совместной компиляции могут отображаться сообщения используемой среды разработки. Описание таких сообщений и указания действий программиста в случае их отображения приведены в документации соответствующей среды разработки.

ПРИЛОЖЕНИЕ
ИНСТРУКЦИЯ ПО КОМПИЛЯЦИИ И ЗАГРУЗКЕ НА ЦЕЛЕВОЕ УСТРОЙСТВО

В настоящем приложении приведена инструкция по совместной компиляции ОСРВ МАКС и пользовательского приложения и загрузке полученных в результате компиляции исполняемых файлов на целевое устройство. В качестве примера целевого устройства взят контроллер STM32F429.

1.1 Компиляция в среде Keil µVision 5

Компиляция в среде Keil µVision 5 производится в следующем порядке:

1) перед началом процесса компиляции необходимо убедиться, что для Keil µVision 5 установлен пакет поддержки целевого устройства. Для рассматриваемого в качестве примера контроллера STM32F429 такой пакет имеет имя Keil.STM32F4xx_DFP.2.5.0. При отсутствии такого пакета его следует установить. Экран управления установленными пакетами и кнопка вызова данного экрана из основного окна Keil µVision 5 показаны на рис. 1.1;

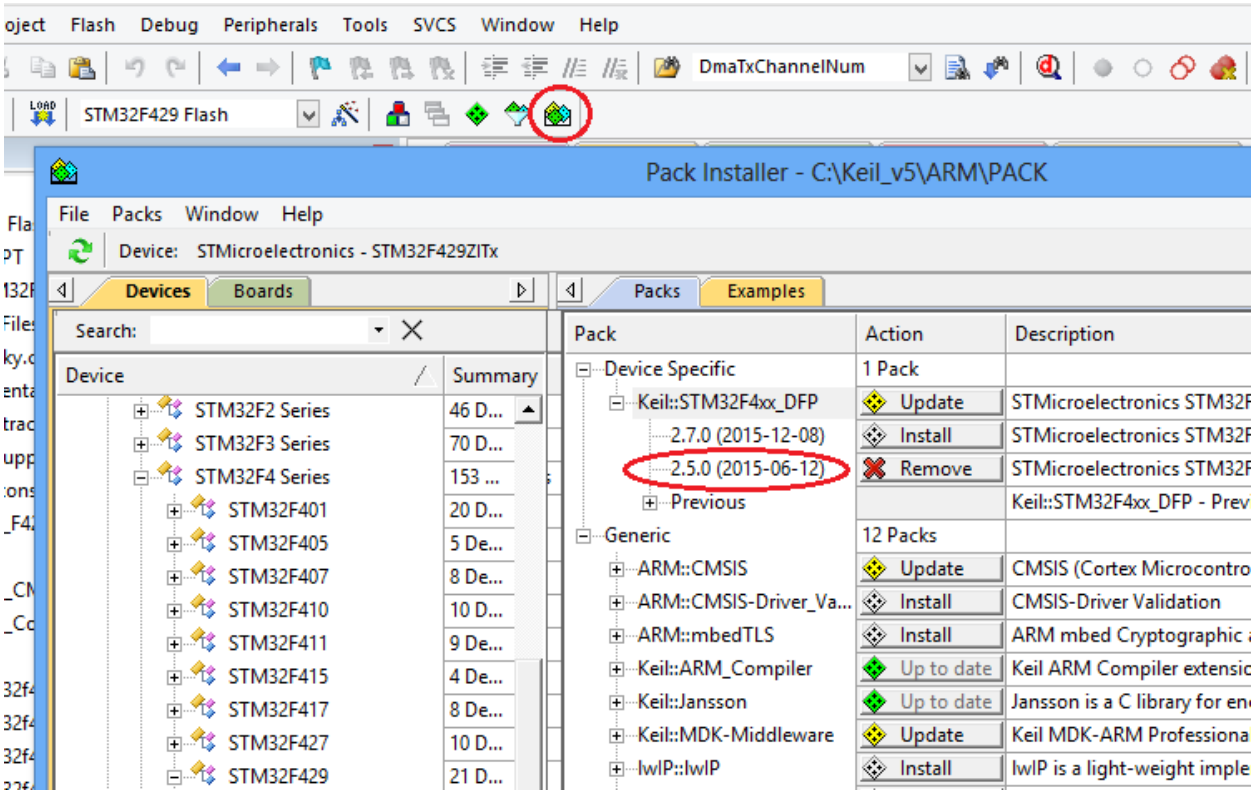

Рисунок 1.1

2) откройте проект maksRTOS.uvprojx. Откройте свойства проекта на вкладке «Device» и убедитесь, что выбрано целевое устройство. Кнопка вызова свойств проекта показана на рис. 1.2;

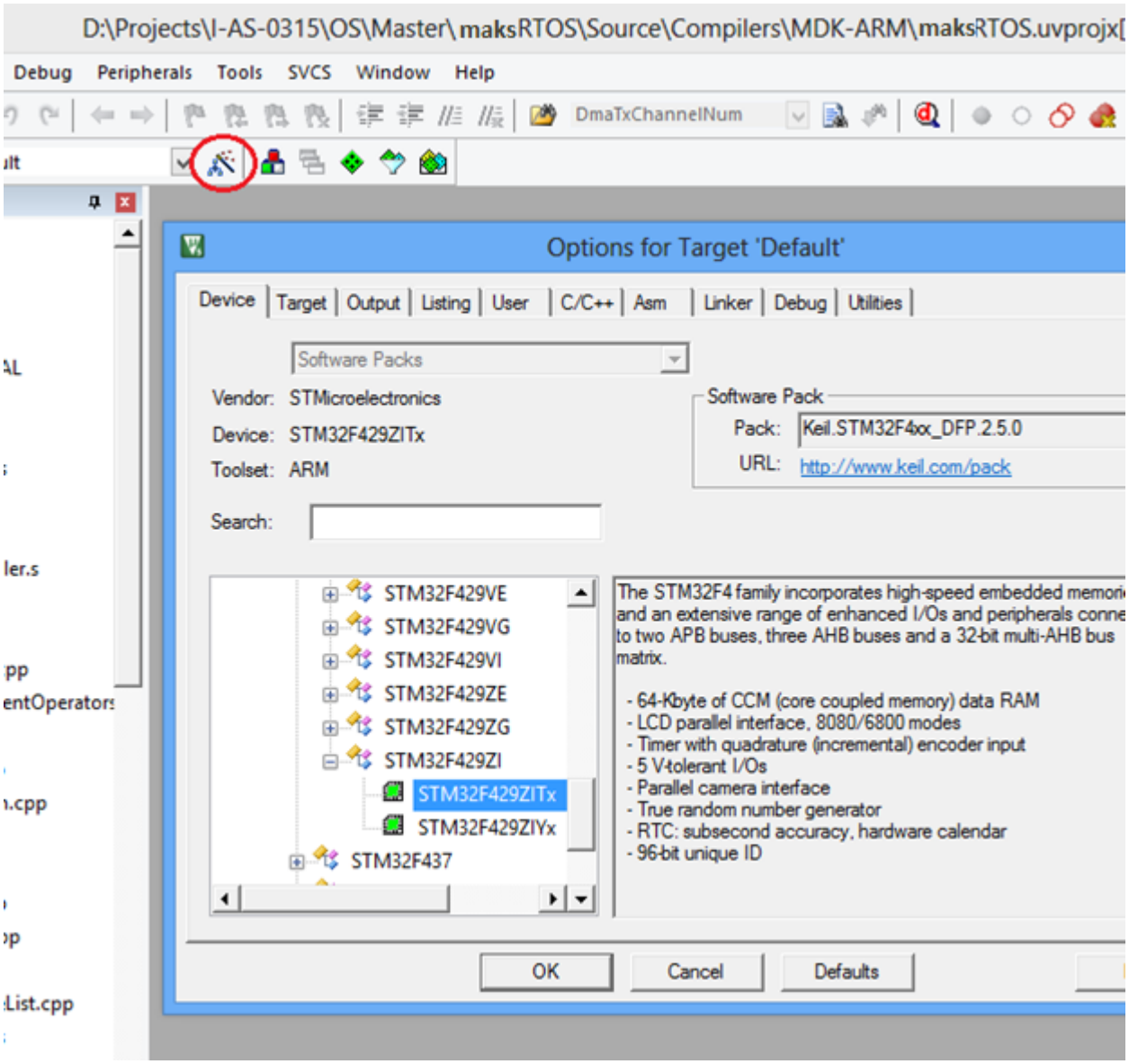

Рисунок 1.2

3) выберите пункты меню «Project/Rebuild all target files». Убедитесь, что компиляция прошла без ошибок (возможна выдача предупреждений). Ошибки в процессе компиляции могут быть связаны с некорректностью исходного кода пользовательского приложения.

Результатом компиляции будет являться образ системы, включающей в себя ОСРВ МАКС и пользовательское приложение. Данный образ может быть загружен в целевое устройство (см. Приложение, 1.2).

1.2 Загрузка образа в целевое устройство из среды Keil µVision 5

Загрузка образа системы в целевое устройство производится в следующем порядке:

1) подключите целевое устройство к компьютеру через кабель mini-USB или иным доступным способом. На рис. 1.3 в качестве примера показано подключение контроллера STM32F429 с помощью кабеля mini-USB;

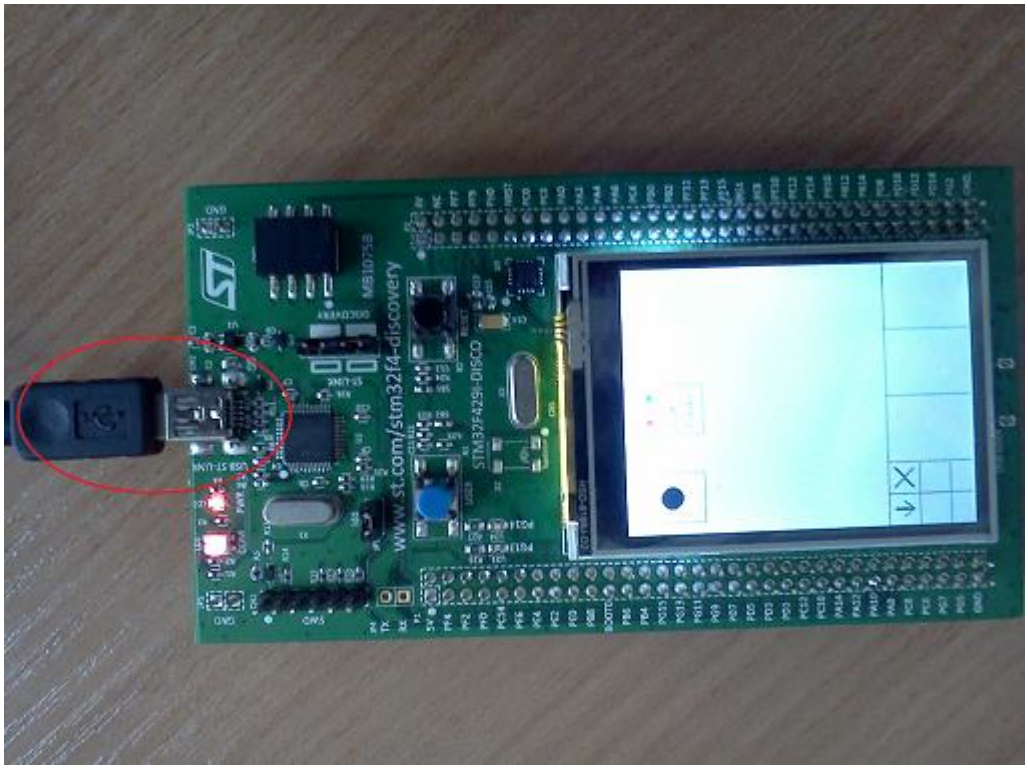

Рисунок 1.3

2) в среде Keil µVision 5 откройте свойства проекта maksRTOS.uvprojx на вкладке «Debug», убедитесь, что выбрана радиокнопка «Use» и в выпадающем списке выбран пункт «ST-Link Debugger»[1)]. Нажмите кнопку «Settings» (рис. 1.4);

[1)] Данная опция зависит от используемого в целевом устройстве программатора. Приведенные значения актуальны для программатора ST-Link. Если в целевом устройстве используется иной программатор, указываемые значения следует соответствующим образом изменить.

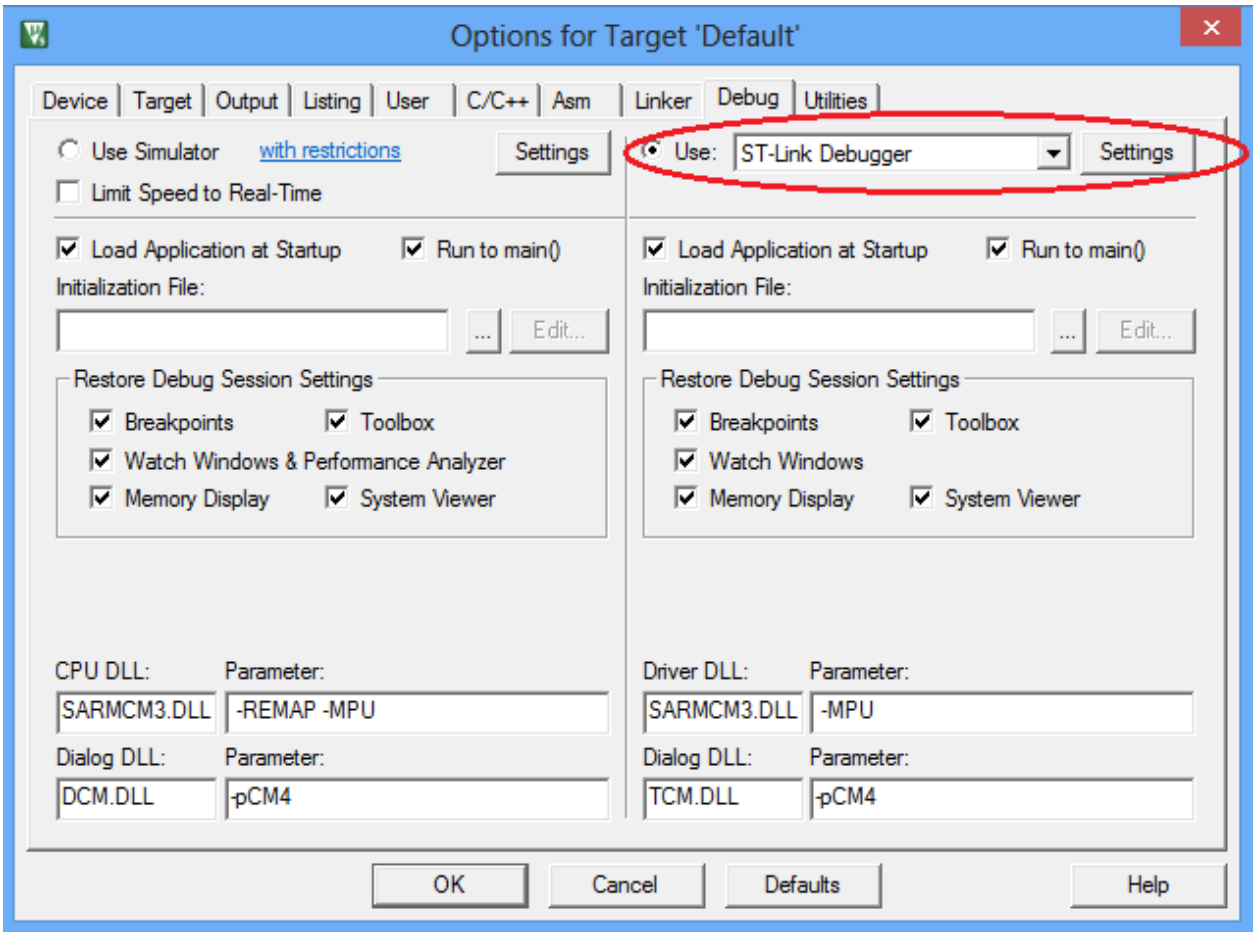

Рисунок 1.4

3) убедитесь, что в отобразившемся окне опций целевого драйвера (рис. 1.5), в выпадающем списке «Unit», выбран ST-LINK/V2[1)], а в выпадающем списке «Port» выбран SW[2)]. Убедитесь, что программатор ST-Link[3)], находящийся на плате контроллера, опознан успешно (в поле «SWDIO» должны отобразиться соответствующие целевому устройству значения). Возможно, для этого придется закрыть диалоговое окно опций целевого драйвера, нажав OK, и открыть его снова;

[1)] Данная опция зависит от используемого в целевом устройстве программатора. Приведенные значения актуальны для программатора ST-Link. Если в целевом устройстве используется иной программатор, указываемые значения следует соответствующим образом изменить.

[2)] То же

[3)] То же

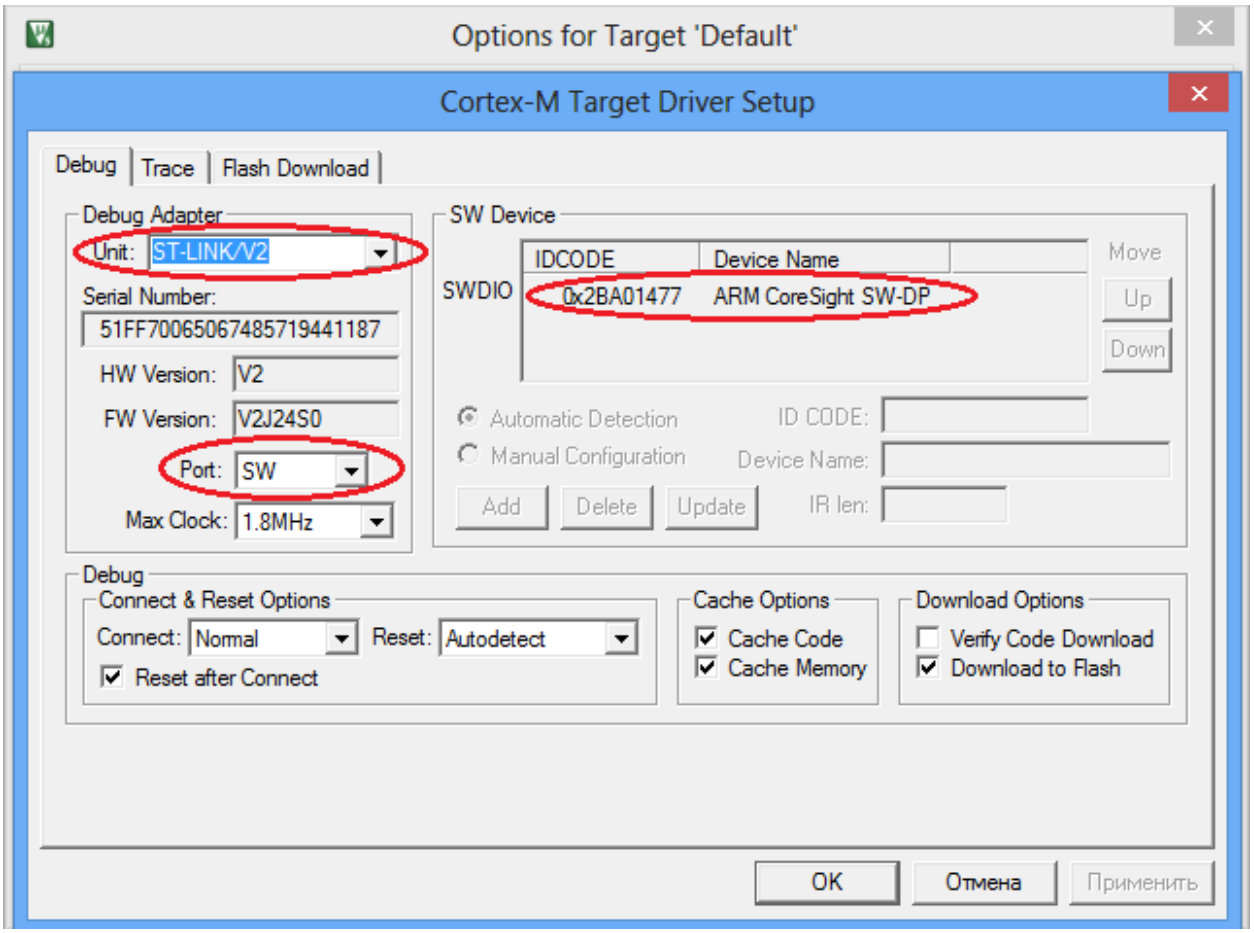

Рисунок 1.5

4) закройте нажатием кнопки «ОК» оба открытых диалоговых окна. Начните загрузку, нажав кнопку панели инструментов «Load» («  ») или выбрав в меню пункт «Flash/Download»;
5) если необходима отладка, ее можно начать кнопкой «Start/Stop Debug Session» («  »). Система запоминает дату/время последнего загруженного образа. Если собранный образ новее, при нажатии кнопки «Start/Stop Debug Session» загрузка стартует автоматически.

1.3 Компиляция в среде EWARM 7.5

Компиляция в среде EWARM 7.5 производится в следующем порядке:

1) откройте рабочее пространство maksRTOS_workspace.eww;
2) выберите в меню пункт «Project/Rebuild All» (расположение пункта меню показано на рис. 1.6);

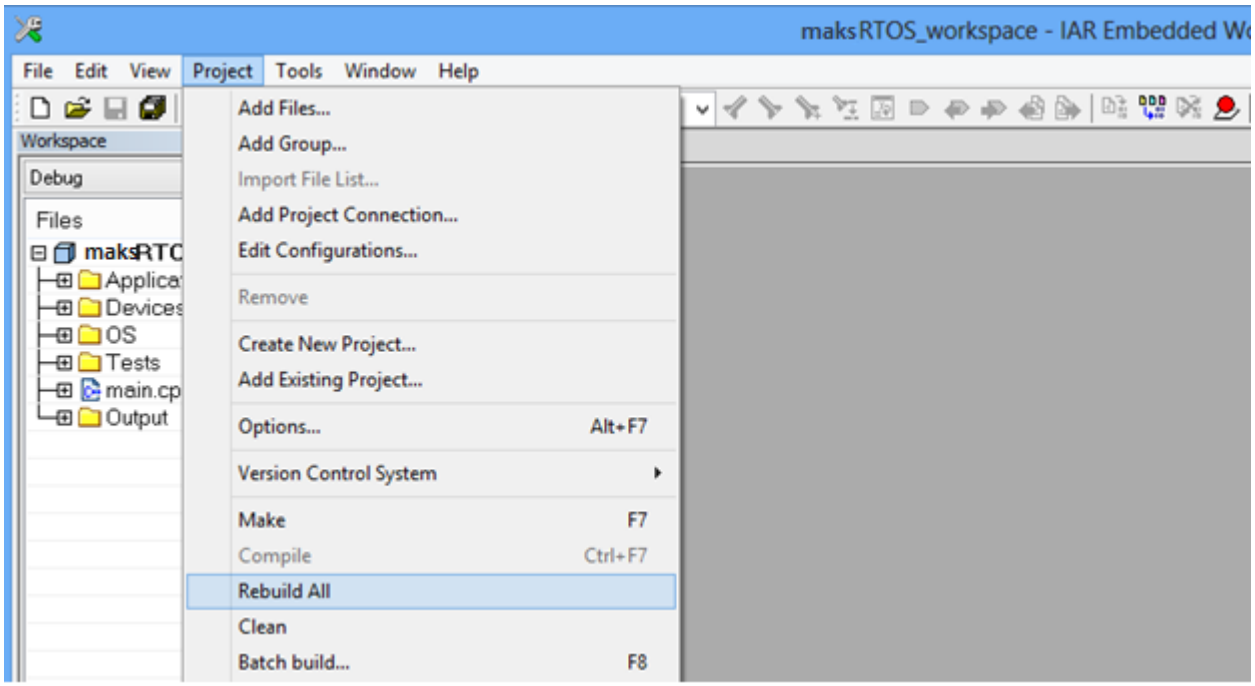

Рисунок 1.6

3) убедитесь, что компиляция прошла без ошибок (возможна выдача предупреждений). Ошибки в процессе компиляции могут быть связаны с некорректностью исходного кода пользовательского приложения.

Результатом компиляции будет являться образ системы, включающей в себя ОСРВ МАКС и пользовательское приложение. Данный образ может быть загружен в целевое устройство (см. Приложение, 1.3).

1.3 Загрузка образа в целевое устройство из среды EWARM 7.5

Загрузка образа в целевое устройство из среды EWARM 7.5 производится в следующем порядке:

1) подключите целевое устройство к компьютеру через кабель mini-USB или иным доступным способом;
2) в среде EWARM 7.5 откройте свойства проекта откройте настройки проекта, выбрав пункт меню «Project/Options…»;
3) в списке категорий (в левой части экрана) выберите пункт «Debugger». Убедитесь, что в поле «Driver» выбран пункт «ST-Link»[1)] (рис. 1.7);

[1)] Данная опция зависит от используемого в целевом устройстве программатора. Приведенные значения актуальны для программатора ST-Link. Если в целевом устройстве используется иной программатор, указываемые значения следует соответствующим образом изменить.

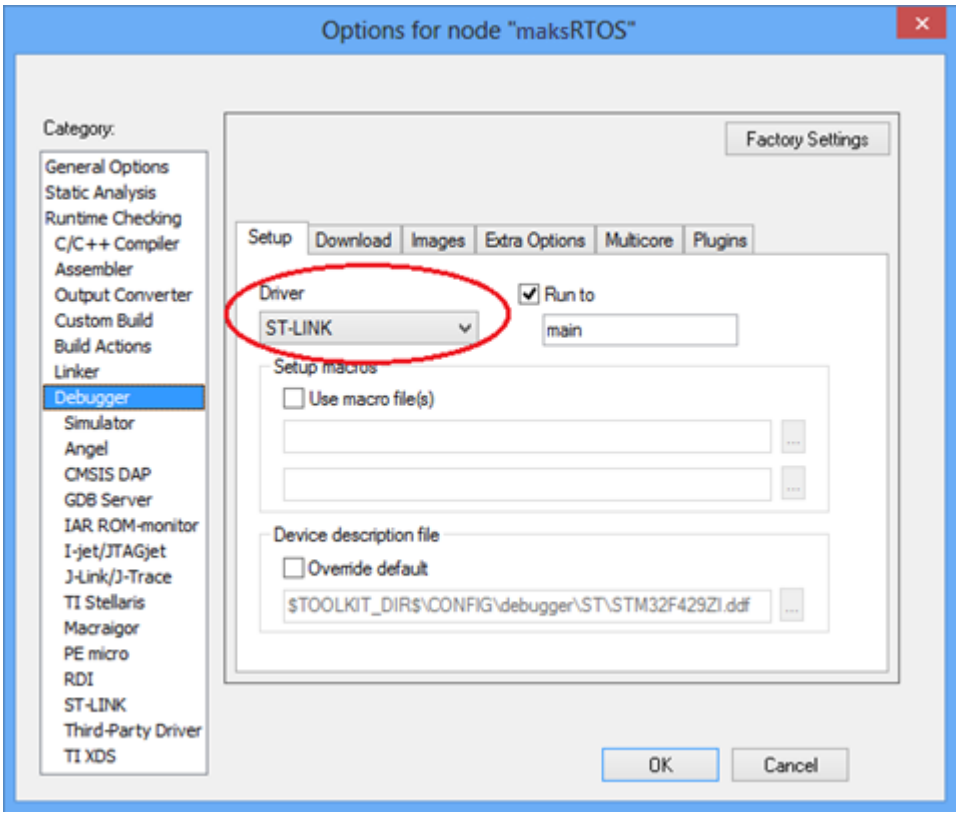

Рисунок 1.7

4) в списке категорий (в левой части экрана) выберите пункт «ST-LINK»[1)], убедитесь, что переключатель «Interface» установлен в положение «SWD»[2)] (рис. 1.8);

[1)] Данная опция зависит от используемого в целевом устройстве программатора. Приведенные значения актуальны для программатора ST-Link. Если в целевом устройстве используется иной программатор, указываемые значения следует соответствующим образом изменить.

[2)] То же

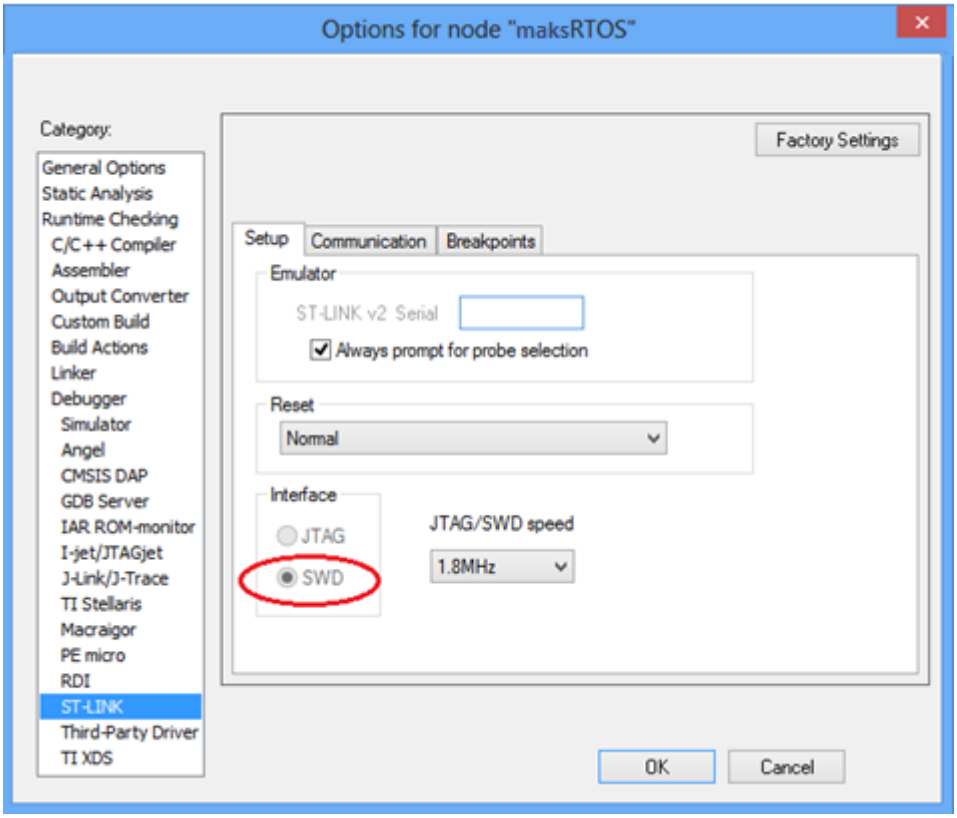

Рисунок 1.8

5) проверьте значения полей на вкладке «Communication». Они должны соответствовать[1)] показанным на рис. 1.9;

[1)] Данная опция зависит от используемого в целевом устройстве программатора. Приведенные значения актуальны для программатора ST-Link. Если в целевом устройстве используется иной программатор, указываемые значения следует соответствующим образом изменить.

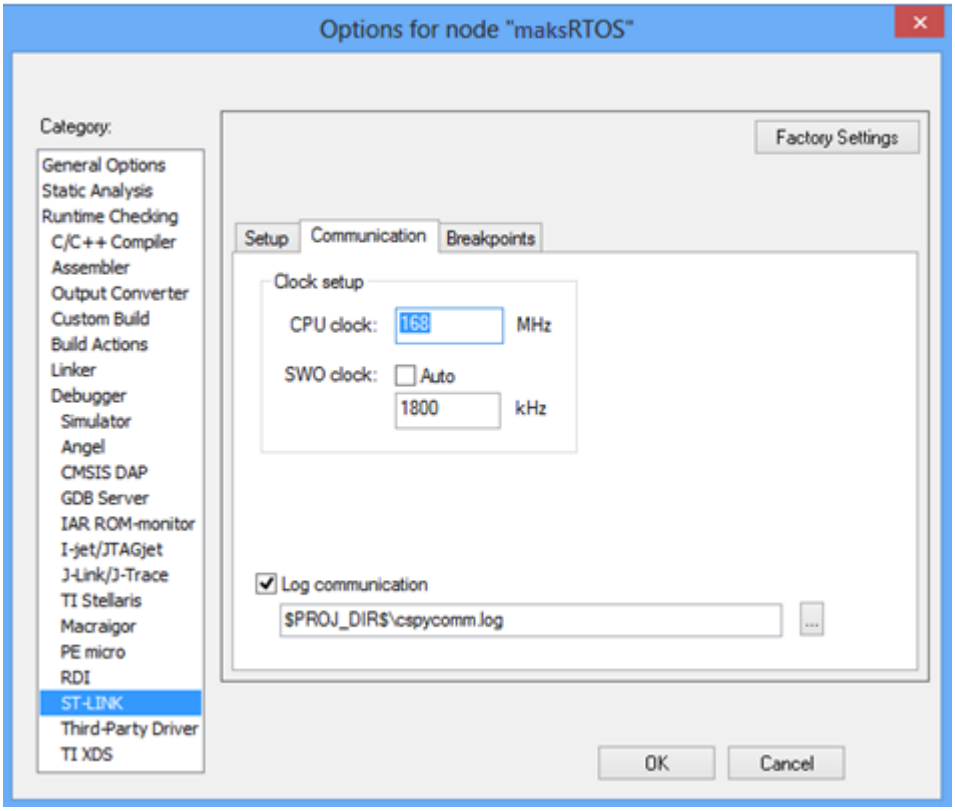

Рисунок 1.9

6) нажмите на панели инструментов кнопку «Download and Debug» («  ») или выберите этот пункт в меню «Project». Впоследствии можно выбирать пункт «Debug without downloading».

ПЕРЕЧЕНЬ ТЕРМИНОВ

Мьютекс	Простейший двоичный семафор, предназначенный для синхронизации одновременно выполняющихся потоков
Пользовательское приложение	Приложение, разрабатываемое с использованием предоставленного ОСРВ МАКС API, предназначенное для функционирования под управлением ОСРВ МАКС
Семафор	Объект, ограничивающий количество потоков, которые могут войти в заданный участок кода
Система реального времени	Система, обеспечивающая реакцию на события во внешней по отношению к системе среде или взаимодействие со средой в рамках заданных временных ограничений
Узел	Микроконтроллер с загруженным в него пользовательским приложением и ОСРВ МАКС
Целевое устройство	Устройство, для управления которым разрабатывается пользовательское приложение
API	*Application Program Interface* – программный интерфейс приложений
MMU	*Memory Management Unit* – блок управления памятью

ПЕРЕЧЕНЬ ПРИНЯТЫХ СОКРАЩЕНИЙ

ОС	Операционная система
ОСРВ МАКС	Операционная система реального времени для мультиагентных когерентных систем

Лист регистрации изменений

Номера листов (страниц)					Всего листов (страниц) в докум.	№ документа	Входящий № сопроводительного докум. и дата	Подп.	Дата
Изм.	измененных	замененных	новых	аннулированных					