**УТВЕРЖДЕН**

**ЮКСУ.91264-01 12 01-УД**

Пакет поддержки программирования микроконтроллера

**(ППП МК)**

Описание применения

ЮКСУ.91264-01 31 01

**МН**

**Листов 69**

**2022**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| **Инв № подл** | **Подпись и дата** | **Взам инв №** | **Инв № дубл** | **Подпись и дата** |

**Аннотация**

Настоящий документ содержит сведения по применению программы "Пакет поддержки программирования микроконтроллера" (ППП МК).

**Содержание**

1 Назначение программы 4

2 Условия применения 4

3 Описание задачи 5

3.1 Общие сведения 5

3.2 ППП МК в общей структуре ПО 6

3.3 Структура ППП МК 8

4 Входные и выходные данные 13

5 Уровень абстракции аппаратуры 13

5.1 Аппаратно-зависимая часть 13

5.2 Интерфейс I2C 14

5.3 Интерфейс SPI 19

5.4 Интерфейс UART 26

5.5 Универсальные функции работы с UART 31

5.6 Подсистема получения временных отсчетов 39

5.7 Менеджер ресурсов 40

6 Особенности К5500ВК018 43

6.1 Архитектурные особенности 43

6.2 Определения config.h 45

6.3 GPIO 46

6.4 Функции инициализации аппаратуры 49

6.5 Файлы main.c/board.c 56

6.6 Обработка прерываний 59

6.7 Примеры программ 63

7 Порядок работы 67

7.1 Установка программы на инструментальную ЭВМ 67

7.2 Сборка программы 67

7.3 Загрузка программы 68

# 1 Назначение программы

* 1. Назначение программы

Программа «Пакет поддержки программирования микроконтроллера» (далее – программа или ППП МК) предназначена для применения в качестве базового инструмента создания прикладных программ микроконтроллера К5500ВК018 (Комдив-МК) и программируемых логических контроллеров (ПЛК) на его основе.

1.2 Возможности программы

1.2.1 ППП МК предоставляет прикладному программисту средства базового уровня абстракции аппаратуры в составе микросхемы микроконтроллера и созданных на его основе устройств.

1.2.2 В программе выделен уровень абстракции работы с микропроцессором, что позволяет обеспечить перенос программного обеспечения (ПО) на различные микроконтроллеры без существенных ограничений. В частности, помимо К5500ВК018 поддерживаются STM32 F1/F3/F4 (ST Microelectronics).

1.2.3 ППП МК может использоваться для Багет-ПЛК1, Багет-ПЛК2, Багет-ПЛК3 и отладочной платы Багет-ПЛК1-01.

# 2 Условия применения

2.1 Для использования ППП МК требуются следующие технические средства:

- инструментальная ЭВМ типа IBM PC c установленной ОС семейства Linux;

- кросс-компилятор с языка С и набор утилит для MIPS (например,

<https://gmplib.org/download/gmp/gmp-6.2.1.tar.xz>

<https://www.mpfr.org/mpfr-current/mpfr-4.1.0.tar.xz>

<https://www.multiprecision.org/downloads/mpc-1.2.0.tar.gz>

<https://ftp.gnu.org/gnu/binutils/binutils-2.37.tar.bz2>

<ftp://ftp.gnu.org/gnu/gcc/gcc-11.3.0/gcc-11.3.0.tar.gz> )

для микроконтроллера К5500ВК018 (ЮКСУ.91264-01 12 02).

2.2 ПЛК или иное устройство на базе микроконтроллера К5500ВК018.

# 3 Описание задачи

## 3.1 Общие сведения

3.1.1 Программа ППП МК представляет собой базовое средство разработки программ для микроконтроллера и ПЛК на его основе, позволяя создавать полноценные встраиваемые приложения. Компактность программы дает возможность применять ее в устройствах без внешней оперативной памяти, с использованием только встроенного ОЗУ микроконтроллера объемом от нескольких десятков килобайт.

3.1.2 Созданные на основе ППП МК прикладные программы функционируют без использования среды какой-либо операционной системы (ОС). Компоненты программы, в свою очередь, могут использоваться в качестве аппаратно-зависимой части ОС.

Аналогичное положение в проектах для микроконтроллеров STM32 занимает библиотека STM32 Hal.

*Примечание: STM32 Hal – разработанная компанией ST Microelectronics (STM) библиотека (набор интерфейсов и реализующих их компонентов), позволяющая прикладному программисту разрабатывать переносимый между микроконтроллерами STM код. Библиотека включает интерфейсы для I2C, SPI, UART и прочих функциональных блоков, реализованных в микроконтроллерах STM. Как правило, используется как часть фреймворка STM STM32CubeMX, предоставляющего графическую среду для конфигурирования микроконтроллера и средства автоматической генерации инициализационного кода, включающего в том числе и вызовы библиотеки STM32 Hal.*

3.1.3 Программа ориентирована на класс микроконтроллеров с одним микропроцессорным ядром и достаточной производительностью (десятки мегагерц частоты микропроцессорного ядра) и/или аппаратные средства распараллеливания (каналы DMA, контроллеры коммуникационных шин).

3.1.4 Программа поддерживает дисциплину кооперативной многозадачности.

*Примечание: кооперативная многозадачность подразумевает такое построение функциональных компонентов программного обеспечения, что получившая управление функция должна как можно быстрее выполнить необходимый набор действий и вернуть управление для вызова следующей функции.*

*Компоненты, разрабатываемые для использования в среде кооперативной многозадачности, должны строиться в виде микрозадач, каждая из которых выполняется небольшое время (его максимальная продолжительность определяется общесистемными требованиями), после чего управление явно возвращается в планировщик или передается следующему компоненту.*

*Для эффективного использования аппаратуры компоненты, получив управление, в течение своих квантов времени осуществляют запуск процессов в независимых аппаратных блоках (контроллерах шин, работающих с использованием DMA, FIFO и т.п.) и их периодический контроль.*

*Помимо регулярной передачи управления, от компонентов, работающих в среде кооперативной многозадачности, требуется сохранение контекста выполнения и (в общем случае) отказ от использования стека (локальных переменных) между вызовами.*

*Рекомендуемым шаблоном для реализации компонентов для среды кооперативной многозадачности является реализация в виде конечного автомата (FSM – Finite State Machine).*

3.1.5 Аппаратно-независимые компоненты программы могут использоваться применительно к микроконтроллерам STM32, при этом аппаратно-зависимые компоненты фактически реализуются вызовами STM32 Hal, а конфигурирование осуществляется STM32CubeMX. Для обеспечения переносимости следует соблюдать определенный шаблон (структуру файлов) исходного кода, предполагающий изоляцию аппаратно-зависимых компонентов (п. 3.3).

3.1.6 Несовпадение набора интерфейсов в разных микроконтроллерах, а также разнообразие схемотехнических решений в устройствах на их основе диктует также выделение устройство-зависимого кода с унификацией функций вызова.

## 3.2 ППП МК в общей структуре ПО

На Рисунке 1 представлена общая структура программного обеспечения микроконтроллера.

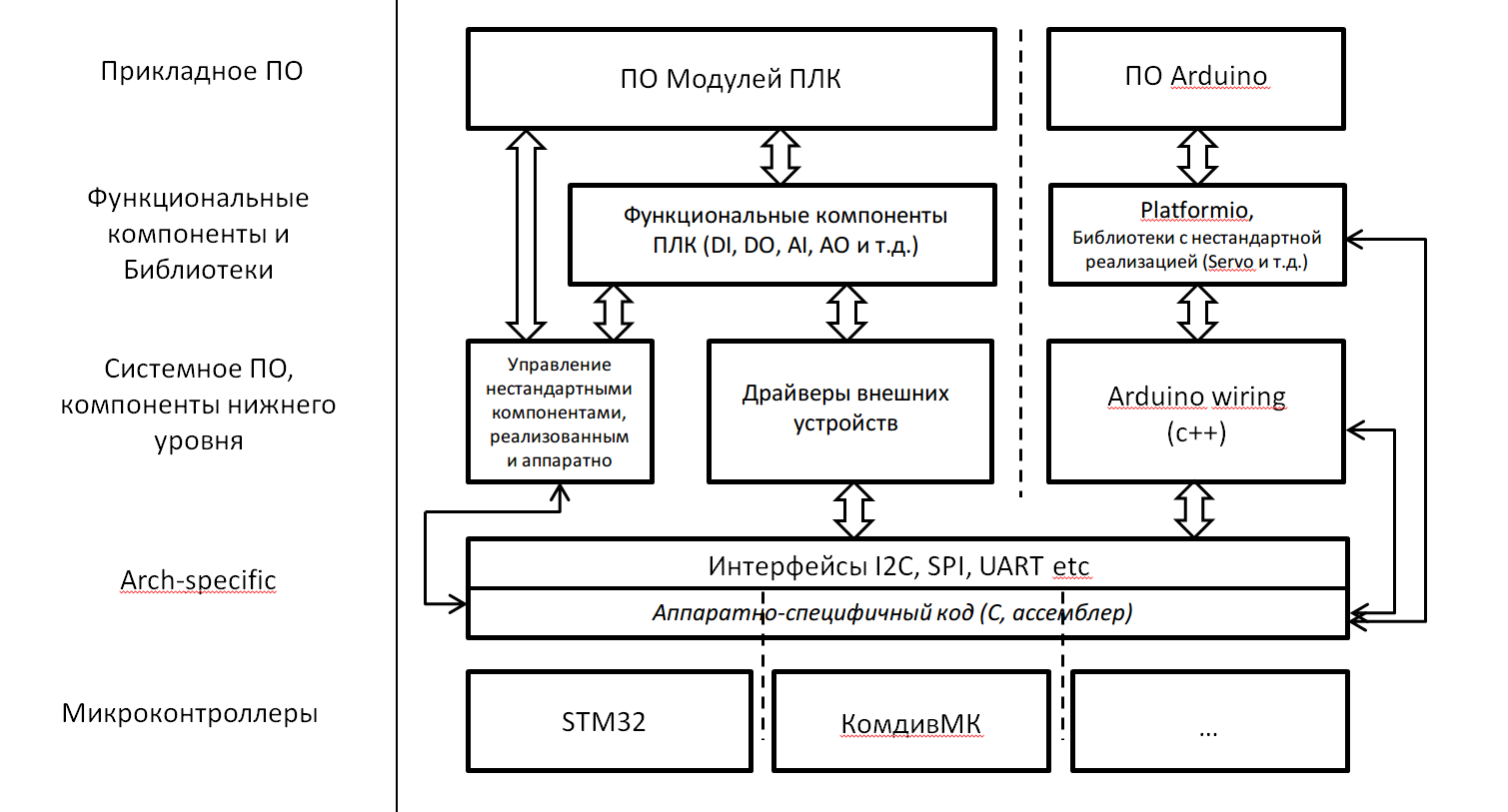


Рис. 1. Общая структура ПО

Здесь нижний уровень демонстрирует класс микроконтроллеров, на которые может портироваться ППП МК.

Уровень «Arch-specific» представляет аппаратно-зависимый и устройство-зависимый код, имеющий унифицированный интерфейс к следующему уровню. Он реализует уровень абстракции аппаратуры.

С уровня «Системное ПО, компоненты нижнего уровня» начинается портируемая часть программного обеспечения, по сути это набор драйверов, реализующий базовый уровень абстракции аппаратуры.

Основной для разработки прикладного ПО слой «Функциональные компоненты и библиотеки» предоставляет прикладному программисту функции для содержательного программирования задачи (например, вместо управления линиями ввода-вывода микроконтроллера предоставляются функции управления присоединенными к этим линиям конечными исполнительными устройствами – индикаторами, реле и пр.).

Уровень прикладного ПО использует нижележащие компоненты для реализации задачи применения микроконтроллера и ПЛК.

Правая часть рисунка демонстрирует соответствующие уровни ПО для Arduino©.

Рассматриваемую структуру также можно проиллюстрировать на примере поддержки ПО, разработанного для Arduino на платформе, образованной аппаратно-специфичными компонентами проекта. Класс задач, на которые ориентированы решения Arduino, схож с задачами модулей ПЛК, а среда выполнения программ также не предполагает операционной системы, но при этом предлагает большой набор библиотек высокого уровня, решающих функциональные задачи или абстрагирующих внешнее оборудование.

В описанной структуре к ППП МК относятся слои «Arch-specific», «Системное ПО, компоненты нижнего уровня» и «Функциональные компоненты и библиотеки».

## 3.3 Структура ППП МК

3.3.1 Пакет поддержки программирования микроконтроллера организован в виде структуры файлов и каталогов (Рисунок 2), представляющей шаблон, на основе которого создается рабочая копия конкретного проекта.

|  |
| --- |
| prg  |-- apps  | |-- myapp1  | | |-- src  | | |-- <board\_name>  | | | |-- main.c (board.c)  | | | |-- config.h  | | | |-- Makefile  | | | |-- <optional board specific>  | | |-- <board\_name>  | | |-- <...>  | |  | |-- myapp2  | | |-- src  | | |-- <...>  | |-- ...  |  |-- arch  | |--<some arch specific>  |  |-- devices  |-- system    Рис. 2. Структура файлов и каталогов |

“**prg**” – корень рабочей копии, имя обычно отражает специфику конкретного проекта.

В каталоге **apps** размещается код прикладных программ (приложений). Поскольку встраиваемое программное обеспечение предполагает функционирование на конкретных устройствах, также здесь размещаются файлы, специфические для конкретных плат, определения (defines), необходимые для конфигурирования компонентов, файл с командами сборщика (**Makefile**) и прочие файлы.

Содержание этого каталога описано ниже.

Каталог **arch** содержит компоненты абстракций аппаратуры (аппаратно-зависимый код). Для Комдив-МК здесь же находится процедура начальной инициализации аппаратуры, выполняющаяся при включении питания. Для STM же такой код, генерируемый автоматически, помещается в каталог платы (<board\_name> на схеме).

Процедура начальной инициализации достаточна для старта программы на языке C: компоненты программы корректным образом размещаются в памяти, область BSS заполняется нулями, формируется стек, передается управление в функцию main(). Также возможно выполнение программ на C++ (некоторые режимы, например, RTTI не тестировались), если в инициализационной процедуре производится обработка массива конструкторов статических объектов (.ctors для gcc). Этого достаточно, например, для C++ программ Arduino, в связи с чем Arduino wiring может быть следующим «слоем», работающим поверх рассматриваемых компонентов.

Каталог **devices** рекомендуется использовать для драйверов устройств, реализованных аппаратно-независимо от архитектуры центрального микроконтроллера (например, драйверы внешних контроллеров, управляемых через коммуникационные шины SPI, I2C и т.д. – внешние АЦП, Flash-память, расширители GPIO и т.д.).

Каталог **system** содержит определения и реализацию системных компонентов, также независимых от архитектуры центрального микроконтроллера (например, консоль поверх последовательного порта).

3.3.2 Пример структур каталога **apps** представлен на Рисунке 3.

|  |
| --- |
| apps  |-- demo\_app  |-- src  | |-- app.h  | |-- demo\_app.c  |  |-- demo\_kmk\_board  |-- Makefile  |-- config.h  |-- main.c  …  Рис. 3. Пример структуры каталога **apps** |

В примере присутствует приложение **demo\_app**, включающее следующие файлы:

**app.h** – объявление функции инициализации приложения и функции его внутреннего цикла. Во всех примерах файл **app.h** будет содержать следующий код:

|  |
| --- |
| #ifndef \_\_APPLICATION\_H\_\_  #define \_\_APPLICATION\_H\_\_  void app\_initialize(void);  void app\_cycle\_tick(void);  #endif /\* \_\_APPLICATION\_H\_\_ \*/ |

Здесь объявлены функции:

app\_initialize() – вызывается один раз после инициализации аппаратуры микроконтроллера, платы и аппаратно-зависимых компонентов. Предполагается, что в данной функции производится инициализация прикладных частей программы.

app\_cycle\_tick() – вызов внутреннего цикла прикладной программы. Прикладная программа должна как можно быстрее выполнить свои действия и осуществить возврат из данной процедуры. Возврат необходим, так как системе может требоваться выполнение каких-то периодических аппаратно-специфических действий (в частности, такая схема требуется при работе без использования прерываний).

Схематичным образом можно представить вызов этих процедур так:

|  |
| --- |
| void main()  {  …  app\_initialize();  while(1) {  app\_cycle\_tick();  ...  };  // never reacheable  } |

(полный код функции main() в данном примере проекта будет находиться в файле **main.c**).

**demo\_app.c** – реализация прикладной программы. Этот файл должен содержать код рассмотренных выше функций app\_initialize() и app\_cycle\_tick(), выполняющий прикладные задачи. Также в реальном проекте будут присутствовать дополнительные файлы с прикладным кодом и объявлениями прикладных функций.

Подкаталог **demo\_kmk\_board** предполагает, что прикладная программа **demo\_app** может функционировать на некой демонстрационной плате. В данном подкаталоге размещаются следующие файлы, задача которых обеспечить сборку программы под конкретную плату:

**Makefile** – файл с командами сборщика. Содержит информацию обо всех файлах, из которых собирается проект, там же могут быть прописаны параметры компилятора и сборщика, такие, как адреса памяти, размер стека и т.д. (при использовании инструментов типа STM CubeMX Makefile генерируется автоматически, и его требуется лишь дополнить именами файлов с исходным кодом).

**config.h** – файл с определениями, относящимися к конфигурации компонентов проекта. Используемые различными компонентами определения приведены в описаниях соответствующих компонентов. Прикладной программист свободен в добавлении в данный файл своих определений. Файл может быть пустым, но должен в обязательном порядке существовать.

**main.c** – в этом файле содержится вся инициализация аппаратуры, которая а) специфична для конкретной платы, и б) осуществляется на языке C, т.е. после выполнения всей низкоуровневой инициализации и готовности стека. Как правило, в данном файле содержится:

- инициализация конкретных выводов микроконтроллера (тип, мультиплексирование, установка начальных уровней), инициализация коммуникационных шин;

- определение кода работающих с линиями GPIO функций с семантикой в домене прикладной программы (например: wp\_enable(on) – установить или отключить защиту записи, set\_failure\_led\_on(on) – включить индикацию ошибки, и т.п.); эти функции на каждой конкретной плате могут быть реализованы по-разному, поэтому необходимо имплементировать их в файле, специфичном для платы;

- установка векторов прерываний;

- вызов app\_initialize() и далее в цикле app\_cycle\_tick().

При использовании инструментов типа STM CubeMX файл **main.c** генерируется автоматически в соответствии с настройками проекта. Прикладной программист дописывает необходимый код в специально отведенные для этого места.

В каталоге, соответствующем плате, также может располагаться скрипт линковщика (файл с расширением .ld, .lds или .lds.S), используемый для сборки программы. STM CubeMX генерирует такой файл автоматически в соответствии с настройками, задаваемым в графической среде. Для микроконтроллера Комдив-МК в состав ППП МК входит типовой скрипт линковщика, подходящий для большинства проектов (находится в каталоге **/arch/kmk**), но при необходимости (например, если требуется добавление обработки .ctors), для проекта может быть создан отдельный скрипт. Формат файла приведен ниже на примере типового скрипта, входящего в ППП МК:

|  |
| --- |
| MEMORY { .ram : ORIGIN = 0x80000000, LENGTH = 0x40000 }  OUTPUT\_ARCH("mips")  SECTIONS  {  .text :  {  \_stext = .;  \_text = .;  KEEP(\*(.text\_head\_entry\*))  KEEP(\*(.text\*))  } > .ram  . = ALIGN(4);  .rodata : { \*(.rodata\*) } > .ram  \_etext = .;  . = ALIGN(4);  .data : { \*(.data\*) } > .ram  \_end = .;  .bss (NOLOAD) : {  . = ALIGN(4);  \_\_bss\_start = .;  \*(.bss\*)  \_\_bss\_stop = .;  } > .ram  } |

3.3.3 Подробный пример программы приведен в п. 6.7.

# 4 Входные и выходные данные

Входных и выходных данных ППП МК не имеет.

# 5 Уровень абстракции аппаратуры

## 5.1 Аппаратно-зависимая часть

Аппаратно-зависимая часть представлена:

1) набором абстракций, используемых далее в программах на С

- I2C

- SPI

- UART (Serial port)

2) подсистемой получения временных отсчетов

3) средствами инициализации, включающими код, выполняющий в том числе:

- задание режима работы ядра процессора;

- инициализацию оперативной памяти (ОЗУ);

- релокацию сегментов программы в ОЗУ;

- подготовку стека;

- выполнение статических инициализационных процедур (например, инициализация статических объектов С++);

- вызов главной процедуры (main).

Для использования абстракций аппаратуры необходимо подключить файл **arch/arch.h** в файлы **\*.c** проекта, где эти абстракции будут использованы.

|  |
| --- |
| #include "arch/arch.h" |

Для использования в проекте интерфейсов I2C, SPI и UART необходимо задать в проекте в файле **config.h** следующие определения:

|  |
| --- |
| #define \_\_ARCH\_WITH\_SERIAL 1  #define \_\_ARCH\_WITH\_SPI 1  #define \_\_ARCH\_WITH\_I2C 1 |

Работа с GPIO, а также с таймерами для проектов, предполагающих фиксацию задач конкретных выводов микроконтроллера и таймеров (например, проекты ПЛК), не абстрагируется, а выносится в аппаратно-зависимый файл инициализации (**main.c, board.c**), где производится настройка необходимых режимов и имплементируются уже более высокоуровневые функции вида run\_led\_on(on), eeprom\_wp(on), etc.

Такой подход хорошо вписывается, например, в использование STM Hal: аппаратно-зависимый файл инициализации генерируется системой STMCubeMX автоматически, позволяя добавить пользовательский код с определением и имплементацией высокоуровневых функций.

Для проектов, в которых настройки и работа с GPIO и таймерами должна осуществляться с прикладного уровня (например, Arduino wiring), реализуются соответствующие абстракции (arduino\_pins, ..., timers).

## 5.2 Интерфейс I2C

Интерфейс работы с шиной I2C (I2C Master) представлен следующим набором функций:

i2c\_get

i2c\_master\_send

i2c\_master\_receive

i2c\_io\_ready

Программный интерфейс разработан для использования в системах с кооперативной многозадачностью, где параллельно с обменом по I2C выполняются и иные операции. Схема работы:

1. Захват шины I2C (обеспечение очереди запросов и отсутствие конфликтов обеспечивается менеджером ресурсов).

2. Формирование команды (send или receive), вызов i2c\_master\_send или i2c\_master\_receive соответственно. Если указанные функции завершились с ошибкой, команда не была принята; при завершении без ошибок процесс на шине начался.

3. Периодический вызов i2c\_io\_ready до тех пор, пока не будет возвращен код успешного завершения или ошибки. После этого шина считается свободной, а буфер чтения (в случае read) будет содержать полученные данные.

Пример простого синхронного взаимодействия с шиной I2C (вызов i2c\_io\_ready в цикле):

|  |
| --- |
| void \*pI2C = i2c\_get(1);  unsigned char ucAddr = (0x23 << 1); /\* адрес устройства на шине \*/  unsigned char pCommand[3] = { 4, 0, 0 }; /\* некая условная команда-посылка устройству \*/  if (!i2c\_master\_send(pI2C, ucAddr, pCommand, 3, 1)) {  while (!(i2c\_io\_ready(pI2C))) { \_\_asm("nop"); }  } |

При использовании 7-битных адресов I2C принято использовать младший бит адресного байта I2C как признак r/w, при этом сам адрес сдвигается на 1 бит влево. Данный комментарий и код специально проясняет этот момент: в функции i2c\_master\_send и i2c\_master\_receive передается значение, которое будет использовано в адресном байте посылки.

5.2.1 **i2c\_get**

void \*i2c\_get(unsigned char ucNum);

Получение указателя на управляющую структуру, сопоставляемую с определенной шиной i2c.

Фактически функция предназначена для установления соответствия между управляющей структурой и принятым при работе с микроконтроллерами номерами (I2C 1, I2C 2 и т.д.).

Управляющая структура формируется при инициализации (см. для примера код для микроконтроллера КомдивМК п. 6.4.1. Функция i2c\_get не осуществляет ни выделений памяти, ни копирований. Для синхронизации доступа к шине при наличии нескольких параллельно выполняющихся задач потребуется внешний компонент (менеджер ресурсов, см. далее).

Указатель на void используется для сокрытия содержимого управляющей структуры, которая различается между микроконтроллерами.

Параметры:

|  |  |
| --- | --- |
| ucNum | номер шины; может быть равен 0, если нумерация шин осуществляется с 0 (например, есть I2C0) |

Результат:

Указатель на структуру. В случае ошибки (шина не проинициализирована для определенного номера или номер неприемлем для микроконтроллера) возвращается 0.

*Замечание*: для большинства программ закрепление функционала внешних устройств, доступных на шине I2C, осуществляется еще на этапе компиляции (и зависит от устройства), в соответствии с чем все номера шин определены заранее и i2c\_get() никогда не будет возвращать ошибки. Этот принцип позволяет для устройств с очень ограниченной оперативной памятью и единственной шиной I2C не формировать в памяти управляющую структуру. Драйвер I2C для такого устройства может возвращать на i2c\_get значение 0, которое не должно интерпретироваться как ошибка. В функциях передачи/приема данных такого драйвера этот параметр будет просто игнорироваться, но должен присутствовать в коде для сохранения единства интерфейса.

Пример вызова:

void \*pI2C1 = i2c\_get(1); // get I2C1 control structure

5.2.2 **i2c\_master\_send**

unsigned char i2c\_master\_send(void \*pInstance,

unsigned short usAddr,

unsigned char \*pbBuffer,

unsigned short usLen,

unsigned char ucFinal);

Передача (отправка) данных по шине I2C.

Функция запускает процесс передачи данных по шине I2C, организуя посылку стартовый бит - адрес - буфер данных - стоповый бит (если задан параметром).

Параметры:

|  |  |
| --- | --- |
| pInstance | указатель на управляющую структуру, результат выполнения функции i2c\_get |
| usAddr | адрес целевого устройства. Данный параметр содержит данные, которые будут непосредственно отправлены на шину, в частности для схемы [7бит адрес - r/w бит] содержит уже сдвинутый на 1 бит влево адрес и добавленный бит r/w |
| pbBuffer | указатель на буфер передаваемых данных |
| usLen | длина буфера в байтах |
| ucFinal | признак необходимости отправки стопового бита:  **0** - не отправлять (предполагается, что будет дальнейшее взаимодействие с устройством),  **1** - отправить |

Результат:

0 - успешное выполнение, процесс запущен

иначе - аппаратно-специфичный код ошибки

После успешного запуска необходимо дождаться завершения операции путем периодического вызова функции i2c\_io\_ready(). При этом буфер pbBuffer на все время выполнения процесса объявляется занятым, модификация его запрещается.

Пример записи в I2C приведен в п. 6.7.

5.2.3 **i2c\_master\_receive**

unsigned char i2c\_master\_receive(void \*pInstance,

unsigned short usAddr,

unsigned char \*pbBuffer,

unsigned short usLen,

unsigned char usFinal);

Прием данных по шине I2C.

Функция запускает следующий процесс: посылка стартового бита, передача адреса, прием данных с внешнего устройства, передача стопового бита (если задан параметром).

Параметры:

|  |  |
| --- | --- |
| pInstance | указатель на управляющую структуру, результат выполнения функции i2c\_get |
| usAddr | адрес целевого устройства. Данный параметр содержит данные, которые будут непосредственно отправлены на шину, в частности для схемы [7бит адрес - r/w бит] содержит уже сдвинутый на 1 бит влево адрес и добавленный бит r/w |
| pbBuffer | указатель на буфер для приема данных |
| usLen | длина буфера в байтах |
| ucFinal | признак необходимости отправки стопового бита:  **0** - не отправлять (предполагается, что будет дальнейшее взаимодействие с устройством),  **1** - отправить |

Результат:

0 - успешное выполнение, процесс запущен

иначе - аппаратно-специфичный код ошибки

После успешного запуска необходимо дождаться завершения операции путем периодического вызова функции i2c\_io\_ready().

5.2.4 **i2c\_io\_ready**

unsigned char i2c\_io\_ready(void \*pInstance);

Функция возвращает информацию о готовности шины I2C (завершении выполнения текущей задачи).

Схема работы с шиной I2C независимо от микроконтроллера следующая:

- запуск процесса передачи/приема данных функциями i2c\_master\_send / i2c\_master\_receive;

- периодическое выполнение i2c\_io\_ready до тех пор, пока процесс не будет полностью завершен.

Параметры:

|  |  |
| --- | --- |
| pInstance | указатель на управляющую структуру, результат выполнения функции i2c\_get |

Результат:

0 – шина занята, процесс не завершен

1 – шина свободна, процесс завершен успешно

иначе – аппаратно-специфичный код ошибки

При вызове i2c\_io\_ready без предварительного запуска задачи (или при повторном вызове после получения ненулевого результата) возвращается значение 1.

Запуск процессов при занятой шине не допускается (i2c\_master\_send / i2c\_master\_receive вернут информацию об ошибке).

Для разработки эффективного прикладного кода необходимо представлять, как именно реализована функция i2c\_io\_ready в драйвере конкретного микроконтроллера. Возможны следующие реализации:

1. Контроллер I2C, реализованный в микроконтроллере, принимает данные о задаче (адрес, направление передачи и т.д.) и далее осуществляет ее выполнение параллельно и независимо от работы вычислительного ядра микроконтроллера. При необходимости осуществляются прерывания, в контексте обработчиков которых производится управление контроллером. По окончанию процесса в управляющей структуре выставляется флаг готовности и код ошибки. При такой реализации сама функция i2c\_io\_ready фактически содержит возврат текущего значения кода (статуса) из управляющей структуры, в связи с чем выполняется моментально, а также (при известности скорости шины и длины посылки) допускает применение таких приемов:

|  |
| --- |
| i2c\_master\_send(pI2C, … );  … /\* программа выполняет иные задачи в течение времени,  когда i2c предполагается занятой \*/  ucResult = i2c\_io\_ready(pI2C); |

2. Обмен данными выполняется контроллером I2C без возможности использования прерываний или непосредственно микроконтроллером (bit-banging). В этом случае управление передачей осуществляется непосредственно в контексте функции i2c\_io\_ready, и если она не вызывается (причем достаточно часто), то шина будет простаивать. Разработка прикладной программы для такого случая должна учитывать разумный баланс приоритетов между данной задачей (обмен по I2C) и параллельными задачами.

## 5.3 Интерфейс SPI

Интерфейс работы с шиной SPI представлен следующим набором функций:

spi\_get

spi\_begin\_transaction

spi\_end\_transaction

spi\_transfer

spi\_io\_ready

5.3.1 **spi\_get**

void \*spi\_get(unsigned char ucNum);

Получение указателя на управляющую структуру, сопоставляемую с определенной шиной SPI.

Фактически функция предназначена для установления соответствия между управляющей структурой и принятым при работе с микроконтроллерами номерами (SPI 1, SPI 2 и т.д.).

Управляющая структура формируется при инициализации, функция spi\_get не осуществляет ни выделений памяти, ни копирований. Для синхронизации доступа к шине при наличии нескольких параллельно выполняющихся задач потребуется внешний компонент (менеджер ресурсов).

Указатель на void используется для сокрытия содержимого управляющей структуры, которая различается между микроконтроллерами.

Параметры:

|  |  |
| --- | --- |
| ucNum | номер шины; может быть равен 0, если нумерация шин осуществляется с 0 (например, есть SPI0) |

Результат:

Указатель на структуру. В случае ошибки (шина не проинициализирована для определенного номера или номер неприемлем для микроконтроллера) возвращается 0.

*Замечание*: для большинства программ закрепление функционала внешних устройств, доступных на шине SPI, осуществляется еще на этапе компиляции (и зависит от устройства), в соответствии с чем все номера шин определены заранее и spi\_get() никогда не будет возвращать ошибки. Этот принцип позволяет для устройств с очень ограниченной оперативной памятью и единственной шиной SPI не формировать в памяти управляющую структуру. Драйвер SPI для такого устройства может возвращать на spi\_get значение 0, которое не должно интерпретироваться как ошибка. В функциях передачи/приема данных такого драйвера этот параметр будет просто игнорироваться, но должен присутствовать в коде для сохранения единства интерфейса.

Пример вызова:

void \*pSPI1 = spi\_get(1); // get SPI1 control structure

5.3.2 **spi\_begin\_transaction**

unsigned char spi\_begin\_transaction(void \*pSpiInstance,

unsigned char ucMode);

Фиксация начала работы с шиной SPI, установка параметра Mode.

В соответствии со спецификацией SPI запись и чтение данных возможны на нарастающем и падающем фронтах сигнала CLK:

|  |  |  |
| --- | --- | --- |
| ucMode | CPOL | CPHA |
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 2 | 1 | 0 |
| 3 | 1 | 1 |

Так как прикладная программа может предполагать параллельное взаимодействие со множеством SPI-устройств, каждое из которых требует своего режима работы, перед началом обмена с устройством необходимо выполнить spi\_begin\_transaction и перевести контроллер SPI в нужный режим.

Параметры:

|  |  |
| --- | --- |
| pSpiInstance | указатель на управляющую структуру, результат выполнения функции spi\_get |
| ucMode | совокупность флагов, определяющих режим работы. В файле arch/arch.h определены следующие:  #define SPI\_CPOL\_0 0  #define SPI\_CPOL\_1 1  #define SPI\_CPHA\_0 0  #define SPI\_CPHA\_1 2 |

Возвращаемое значение: unsigned char

0 – транзакция уже была начата ранее, ошибка

1 – успешно, контроллер находится в нужно режиме, транзакция начата

иначе - аппаратно-специфичный код ошибки/

Данная функция не осуществляет манипуляции сигналом CS! Манипуляция этим сигналом осуществляется отдельно. Функция spi\_begin\_transaction осуществляет только настройку контроллера SPI для работы в нужном режиме.

Некоторые драйверы SPI также могут использовать spi\_begin\_transaction для управления занятостью шины.

Пример:

|  |
| --- |
| void \*pSPI = spi\_get(1);  spi\_begin\_transaction(pSPI, SPI\_CPOL\_0 | SPI\_CPHA\_0);  spi1\_cs1(0); /\* манипуляция сигналом CS  для выбора устройства \*/  … |

5.3.3 **spi\_end\_transaction**

void spi\_end\_transaction(void \*pSpiInstance);

Фиксация окончания работы с шиной SPI.

Функция сбрасывает в управляющей структуре флаг начала работы. В некоторых драйверах (например, для Комдив-МК) содержит пустое тело.

Параметры:

|  |  |
| --- | --- |
| pSpiInstance | указатель на управляющую структуру |

Возвращаемое значение отсутствует.

5.3.4 **spi\_transfer**

unsigned char spi\_transfer(void \*pSpiInstance,

unsigned char \*pbSendBuffer,

unsigned uSSize,

unsigned char \*pbReadBuffer,

unsigned uRSize);

Передача (обмен данными) по шине SPI.

Функция запускает процесс обмена данными по шине SPI. При этом:

а) в общем случае slave-устройство должно быть выбрано сигналом CS до вызова данной функции: т.е. прикладная программа может, опустив CS, сделать несколько обменов с устройством, используя небольшие буфера. Однако в некоторых контроллерах SPI управление CS осуществляется самим контроллером аппаратно непосредственно в контексте процедуры обмена (например, контроллер SPI микроконтроллера Комдив-МК), и это необходимо учитывать при разработке логики работы прикладной программы.

Пример: контроллер SPI0 Комдив-МК имеет 4 линии аппаратного управления выбором устройства, которые не мультиплицируются на GPIO и могут управляться только контроллером SPI.

В таком случае реализация управления CS может быть реализована следующим образом (линия CS активируется низким уровнем bOn=0):

void spi\_cs0(unsigned char bOn)

{

if (bOn) {

// inactive state (reset CS line in controller)

SPI\_SetReg(SPI0\_BASE, 8, (1 << 0)); // 0 is CS0

} else {

// active state (configure controller to use CS0)

SPI\_SetReg(SPI0\_BASE, 0x24, 0); // 0 is CS0

}

}

Но при такой реализации необходимо понимать, что непосредственно изменение уровня на линии CS произойдет тогда, когда контроллер начнет транзакцию (т.е. если есть необходимость выдержать CS в активном состоянии перед началом транзакции или между транзакциями, это придется делать настройками контроллера; также не удастся таким образом активировать линию для получения значений с линии MISO от некоторых АЦП, которые таким образом предоставляют информацию о готовности конверсии).

б) работа с шиной SPI конструктивно является обменом данными (т.е. движение данных в обе стороны), при этом для целей экономии памяти под буфера функция допускает применение для «передачи» (имеющие смысл данные передаются только от микроконтроллера, а ответные данные отбрасываются) и «приема» (управляемое устройство отбросит данные микроконтроллера).

Примеры:

Обмен по SPI – передача из буфера pBuf1, прием в буфер pBuf2:

spi\_transfer(pSPI, pBuf1, uSize, pBuf2, uSize);

Обмен по SPI – передача из буфера pBuf, и прием в него же:

spi\_transfer(pSPI, pBuf, uSize, pBuf, uSize);

Обмен по SPI – требуются лишь результаты чтения; контроллер SPI выставит на шину данные по своему усмотрению, как правило, 0xff.. :

spi\_transfer(pSPI, 0, 0, pBuf, uSize);

Обмен по SPI – значение имеет лишь запись, данные от устройства не интересуют:

spi\_transfer(pSPI, pBuf, uSize, 0, 0);

Параметры:

|  |  |
| --- | --- |
| pSpiInstance | указатель на управляющую структуру, результат выполнения функции spi\_get |
| pbSendBuffer | указатель на буфер для передаваемый данных. Если требуется только «чтение» данных с устройства, следует установить этот параметр равным 0 (при этом данные, которые непосредственно будут переданы контроллером SPI в цикле обмена остаются на усмотрение контроллера; как правило, это 0xFF). |
| uSSize | длина буфера передаваемых данных. Если требуется только чтение, следует установить этот параметр равным нулю. |
| pbReadBuffer | указатель на буфер для приема данных. Если требуется только передача данных (принимаемые отбрасываются), следует установить этот параметр равным 0. |
| uRSize | размер буфера для приема данных. Если требуется только передача данных (принимаемые отбрасываются), следует установить этот параметр равным 0. |

*Замечания*:

- если требуется провести обмен данными (т.е. используются как приемный, так и передающий буферы), значения uSSize и uRSize должны совпадать, в противном случае вызов завершится с ошибкой;

- после запуска процесса и до его завершения изменять данные в буфере передаваемых данных запрещено: в общем случае драйвер SPI не обязан копировать данные, а будет брать их непосредственно из буфера;

- прием данных в тот же буфер, что используется для передачи, является допустимой и разрешенной операцией, драйверы должны корректно обрабатывать такой вызов. В этом случае значения pbSendBuffer и pbReadBuffer совпадают.

Результат:

1 - успешное выполнение, процесс запущен

иначе - аппаратно-специфичный код ошибки

После успешного запуска необходимо дождаться завершения операции путем периодического вызова функции spi\_io\_ready().

5.3.5 **spi\_io\_ready**

unsigned char spi\_io\_ready(void \*pSpiInstance);

Функция возвращает информацию о готовности шины SPI (завершении выполнения обмена на шине).

Схема работы с шиной SPI независимо от микроконтроллера следующая:

- запуск процесса обмена функцией spi\_transfer;

- периодическое выполнение spi\_io\_ready до тех пор, пока процесс не будет полностью завершен.

Параметры:

|  |  |
| --- | --- |
| pSpiInstance | указатель на управляющую структуру, результат выполнения функции spi\_get |

Результат:

0 – шина занята, процесс не завершен

1 – шина свободна, процесс завершен успешно

иначе – аппаратно-специфичный код ошибки

При вызове spi\_io\_ready без предварительного запуска задачи (или при повторном вызове после получения ненулевого результата) возвращается значение 1.

Запуск процессов во время выполнения обмена на шине не допускается (spi\_transfer вернет информацию об ошибке).

Для разработки эффективного прикладного кода необходимо представлять, как именно реализована функция spi\_io\_ready в драйвере конкретного микроконтроллера. Возможны следующие реализации:

1. Контроллер SPI, реализованный в микроконтроллере, принимает данные о задаче (буфер приема, буфер передачи, размер) и далее осуществляет ее выполнение параллельно и независимо от работы вычислительного ядра микроконтроллера. По окончанию процесса в управляющей структуре выставляется флаг готовности и код ошибки. При такой реализации сама функция spi\_io\_ready фактически содержит возврат текущего значения кода из управляющей структуры, в связи с чем выполняется моментально, а также (при известности скорости шины и длины посылки) допускает применение таких приемов:

|  |
| --- |
| spi\_transfer(pSPI, … );  … /\* программа выполняет иные задачи в течение  времени, когда SPI предполагается занятой \*/  ucResult = spi\_io\_ready(pSPI); |

2. Обмен данными выполняется контроллером SPI без возможности использования прерываний или непосредственно микроконтроллером (bit-banging). В этом случае управление обменом осуществляется непосредственно в контексте функции spi\_io\_ready, и если она не вызывается (причем достаточно часто), то шина будет простаивать. Разработка прикладной программы для такого случая должна учитывать разумный баланс приоритетов между данной задачей (обмен по SPI) и параллельными задачами.

Шина SPI – быстрая (в отличие от I2C), в связи с чем на больших частотах контроллер SPI может выполнять обмен со скоростью, сопоставимой со скоростью ядра микроконтроллера. В этом случае даже для программ с кооперативной многозадачностью могут быть допустимы конструкции вида

|  |
| --- |
| spi\_transfer(pSPI, …);  while((ucRes = spi\_io\_ready(pSPI)) == 0)  { \_\_asm(“nop”); } |

## 5.4 Интерфейс UART

Интерфейс работы со встроенными UART микроконтроллера представлен следующим набором функций:

\_\_system\_serial\_get

\_\_system\_serial\_configure

\_\_system\_serial\_available

\_\_system\_serial\_read

\_\_system\_serial\_write

\_\_system\_serial\_ready

5.4.1 **\_\_system\_serial\_get**

void \* \_\_system\_serial\_get(unsigned char ucNum);

Получение указателя на управляющую структуру, сопоставляемую с определенным последовательным интерфейсом (UART) Фактически функция предназначена для установления соответствия между управляющей структурой и принятым при работе с микроконтроллерами номерами (UART 1, UART 2 и т.д.).

Управляющая структура формируется при инициализации, функция \_\_system\_serial\_get не осуществляет ни выделений памяти, ни копирований. Для синхронизации доступа к контроллеру при наличии нескольких параллельно выполняющихся задач потребуется внешний компонент (менеджер ресурсов).

Указатель на void используется для сокрытия содержимого управляющей структуры, которая не предполагает непосредственной работы с ее элементами.

Параметры:

|  |  |
| --- | --- |
| ucNum | номер интерфейса UART; может быть равен 0, если нумерация осуществляется с 0 (например, есть UART0) |

Результат:

Указатель на структуру. В случае ошибки (контроллер не проинициализирован для определенного номера или номер неприемлем) возвращается 0.

*Замечание*: для большинства программ распределение задач по интерфейсам UART осуществляется еще на этапе компиляции (и зависит от конкретной платы), в соответствии с чем все интерфейсы UART известны и проинициализированы заранее. Таким образом \_\_system\_serial\_get() никогда не будет возвращать ошибки. Этот принцип позволяет для устройств с очень ограниченной оперативной памятью и единственным UART не формировать в памяти управляющую структуру. Функция \_\_system\_serial\_get() для такого устройства может возвращать значение 0, которое не должно интерпретироваться как ошибка. В функциях передачи/приема данных такого драйвера этот параметр будет просто игнорироваться, но должен присутствовать в коде для сохранения единства интерфейса.

Пример вызова:

|  |
| --- |
| void \*pSerial1 = \_\_system\_serial\_get(1); // get serial 1 control structure |

5.4.2 **\_\_system\_serial\_configure**

int \_\_system\_serial\_configure(void \*pInstance,

unsigned long ulBaudrate,

unsigned char ucBits,

unsigned char ucParity,

unsigned char ucStops);

Конфигурирование последовательного интерфейса. Данная функция является непосредственно функцией драйвера, и прикладная программа не должна без необходимости вызывать ее напрямую. Для вызова из прикладной программы существует функция serial\_configure, позволяющая помимо интерфейсов UART микроконтроллера прозрачно работать с внешними UART (например, доступным через SPI XR20M).

Параметры:

|  |  |
| --- | --- |
| pInstance | указатель на управляющую структуру, результат функции \_\_system\_serial\_get() |
| ulBaudrate | скорость обмена в битах в секунду (например, 115200) |
| ucBits | количество бит (5, 6, 7, 8) |
| ucParity | четность (отсутствует, четность, нечетность):  0 – нет четности  1 – нечетность  2 – четность |
| ucStops | количество стоповых битов (1, 2) |

Результат:

1 – успешно

иначе – аппаратно-специфичный код ошибки

5.4.3 **\_\_system\_serial\_available**

int \_\_system\_serial\_available(void \*pInstance, unsigned \*puLen);

Запуск процесса проверки наличия принятых UART данных.

Параметры:

|  |  |
| --- | --- |
| pInstance | указатель на управляющую структуру, результат функции \_\_system\_serial\_get() |
| puLen | указатель на переменную для размещения количества имеющихся данных |

Результат:

0 – устройство занято, требуется повторить вызов позже

1 – успешно (процесс запущен)

иначе – аппаратно-специфичный код ошибки

После получения кода 1 необходимо осуществлять вызов \_\_system\_serial\_ready (serial\_ready) до тех пор, пока процесс не будет завершен.

До завершения процесса изменение данных в буфере puLen не допускается.

5.4.4 **\_\_system\_serial\_read**

int \_\_system\_serial\_read(void \*pInstance,

unsigned char \*pbBuffer,

unsigned \*puLen);

Запуск процесса чтения принятых UART данных.

Параметры:

|  |  |
| --- | --- |
| pInstance | указатель на управляющую структуру, результат функции \_\_system\_serial\_get() |
| pbBuffer | указатель на буфер для принимаемых данных |
| puLen | указатель на переменную, в которой будет передан размер буфера для чтения, а после успешного выполнения функции будет содержаться количество реально прочитанных данных |

Результат:

0 – устройство занято, требуется повторить вызов позже

1 – успешно (процесс запущен)

иначе – аппаратно-специфичный код ошибки

После получения кода 1 необходимо осуществлять вызов \_\_system\_serial\_ready (serial\_ready) до тех пор, пока процесс не будет завершен.

До завершения процесса изменение данных в буфере pbBuffer и переменной puLen не допускается.

5.4.5 **\_\_system\_serial\_write**

int \_\_system\_serial\_write(void \*pInstance,

unsigned char \*pbBuffer,

unsigned \*puLen);

Запуск процесса передачи данных в UART.

Параметры:

|  |  |
| --- | --- |
| pInstance | указатель на управляющую структуру, результат функции \_\_system\_serial\_get() |
| pbBuffer | указатель на буфер для передаваемых данных |
| puLen | указатель на переменную, в которой будет передан размер буфера для передачи, а после успешного выполнения функции будет содержаться количество реально переданных данных |

Результат:

0 – устройство занято, требуется повторить вызов позже

1 – успешно (процесс запущен)

иначе – аппаратно-специфичный код ошибки

После получения кода 1 необходимо осуществлять вызов \_\_system\_serial\_ready (serial\_ready) до тех пор, пока процесс не будет завершен.

До завершения процесса изменение данных в буфере pbBuffer и переменной puLen не допускается.

5.4.6 **\_\_system\_serial\_ready**

int \_\_system\_serial\_ready(void \*pInstance, unsigned char ucOperation);

Функция возвращает информацию о готовности последовательного интерфейса (завершении выполнения определенной операции).

Так как в общем работа с последовательным интерфейсом допускает одновременное выполнение операций (например, чтения и записи), вызов \_\_system\_serial\_ready должен осуществляться с указанием именно той операции, которая была запущена ранее конкретным компонентом/задачей.

Предусмотрены (serial.h) следующие варианты:

|  |
| --- |
| #define SERIAL\_OPERATION\_RX  #define SERIAL\_OPERATION\_TX  #define SERIAL\_OPERATION\_CONF  #define SERIAL\_OPERATION\_AVAIL |

Параметры:

|  |  |
| --- | --- |
| pInstance | указатель на управляющую структуру, результат функции \_\_system\_serial\_get() |
| ucOperation | операция |

Результат:

0 – устройство занято, процесс еще продолжается

1 – успешно (процесс закончен, можно интерпретировать данные в буферах)

иначе – аппаратно-специфичный код ошибки

## 5.5 Универсальные функции работы с UART

Часто встречаются устройства, в которых последовательная передача данных происходит не только через встроенные последовательные порты микроконтроллера, но и через внешние устройства, например, отдельные контроллеры UART, подключаемые через шину SPI (XR20M и др.), или устройства, эмулирующие последовательный порт поверх сетевой или иной среды.

Архитектура ППП МК предполагает абстрагирование работы с последовательными портами (единый набор функций), что позволяет разрабатывать компоненты (консоли, коммуникационные сервисы и т.д.), прозрачно работающие с UART.

Сигнатуры универсальных функций в целом совпадают с функциями, рассмотренными в п. 5.4, но не имеют префикса \_\_system. Также универсальными функциями реализуются дополнительные возможности (как правило, они доступны и для встроенных UART микроконтроллера \*) – это установка режима работы приема посредством коллбэка (обратного вызова, callback) и управление полудуплексными линиями. Эти функции будут рассмотрены ниже.

Примечание: \* возможности, предоставляемые различными реализациями UART (как аппаратурой, так и драйверами) могут отличаться; некоторые возможности могут быть недоступны, некоторые дополнительные функции могут потребовать прямой работы с аппаратурой.

Например, проблема может возникнуть из-за разных требований к сигнализации окончания процесса передачи данных для полнодуплексных и полудуплексных линий (особенно при наличии FIFO). Возврат готовности функцией serial\_ready сразу по загрузке данных в FIFO позволит сразу же продолжить передачу, обеспечивая таким образом максимальную пропускной способность. Однако для полудуплексных линий, где необходимо переключить режим порта с передачи на прием, требуется сигнализации готовности по завершению передачи всего буфера на линию. ППП МК предполагает, что подобная настройка должна обеспечиваться при инициализации аппаратуры.

Подключение дополнительных последовательных портов осуществляется посредством определения специальных макросов в файле **config.h**.

Рассмотрим пример драйвера xr20m, реализующего набор функций

int xr20m\_serial\_configure(void \*pInstance,

unsigned long ulBaudrate,  
 unsigned char ucBits,

unsigned char ucParity,

unsigned char ucStops);

int xr20m\_serial\_available(void \*pInstance,

unsigned \*puLen);

int xr20m\_serial\_read(void \*pInstance,

unsigned char \*pbBuffer,  
 unsigned \*puLen);

int xr20m\_serial\_write(void \*pInstance,

unsigned char \*pbBuffer,

unsigned \*puLen);

int xr20m\_serial\_ready(void \*pInstance,

unsigned char ucOperation);

Все эти функции имеют префикс xr20m.

Чтобы эти функции могли быть доступны через универсальный интерфейс, необходимо добавить в **config.h** следующие определения:

#define SERIAL\_DRV\_XR20M 1

#define SERIAL\_INSERT\_DRIVER\_1 xr20m

SERIAL\_DRV\_XR20M – данный макрос задает значение типа (идентификатора) для драйвера xr20m.

SERIAL\_INSERT\_DRIVER\_1 – макрос задает префикс функций для драйвера с типом (идентификатором) 1.

При подключении к проекту следующего дополнительного последовательного порта необходимо будет использовать тип (идентификатор) 2 и так далее.

5.5.1 **serial\_set\_rx\_callback**

int serial\_set\_rx\_callback(void \*pInstance,

serial\_rx\_callback fRXC,

void \*pHandle);

Установка режима коллбэка (callback, обратного вызова) для приема данных с последовательного порта.

Механизм приема данных от последовательного порта предусматривает следующие режимы:

1. Контроллер порта при приеме данных размещает их в приемном регистре (или приемной очереди FIFO, если она предусмотрена) и затем эти данные должны быть явно забраны вызовами функций serial\_available/serial\_read.

2. Контроллер порта, получив данные, формирует прерывание и в его контексте вызывает обработчик, передавая в него принятые данные.

Каждый из этих режимов имеет преимущества и недостатки, при этом не все драйверы могут предоставлять оба режима.

Явная выборка данных посредством serial\_available/serial\_read на аппаратуре с FIFO позволяет обрабатывать данные с минимальными накладными расходами (в частности, пакетным образом), что при больших скоростях (1 Mbit и более) является единственным возможным вариантом. Также этот режим позволяет использовать преимущества DMA. Недостатком является сложность разбора пакета, особенно для протоколов, где имеют значение временные промежутки между символами (например, modbus RTU).

Режим с прерываниями позволяет в частности точно фиксировать время приема символа, что существенно упрощает реализацию основанных на временных промежутках протоколов, а в некоторых случаях повышает скорость реакции системы. Но большие накладные расходы на вход в прерывание приводят к ограничениям по скорости приема, а работа в контексте обработчика прерывания требует механизмов синхронизации и в целом ограничивает возможности обработки.

Параметры:

|  |  |
| --- | --- |
| pInstance | указатель на управляющую структуру (для встроенного порта - результат функции \_\_system\_serial\_get(), для внешних – зависит от драйвера); |
| fRXC | функция обратного вызова. Имеет следующую сигнатуру:  typedef int (\*serial\_rx\_callback)(void \*pHandle,  unsigned char \*pbBuffer,  unsigned uSize);  Где pHandle – контекст вызова (см. ниже), pbBuffer – указатель на буфер с принятыми данными (как правило если речь идет об обработчике прерывания, там будет всего один символ, но это не обязательно), uSize – размер данных в буфере.  Функция возвращает 1, если данные успешно обработаны, и 0 – если произошла ошибка. В общем случае драйвер может проигнорировать это значение, но может и использовать его, например, для управления потоком.  Сигнатура функции объявлена в файле **serial.h** |
| pHandle | Контекст вызова, который будет передаваться в функцию обратного вызова. Позволяет реализовать универсальные обработчики для набора портов. Может содержать любое значение, интерпретируется только внутри функции обратного вызова. |

Функция возвращает 1 в случае, когда адрес функции был успешно сохранен в управляющей структуре порта, и 0 – если нет (был передан нулевой указатель на pInstance).

Как правило, режим работы устанавливается при инициализации программного обеспечения. Тем не менее, возможно переключение между режимами (для отказа от режима с обратным вызовом необходимо передать в параметре fRCX нулевой указатель). При этом потребуется синхронизация: запрет прерывания порта и завершение всех операций по обработке принятых данных.

5.5.2 **serial\_set\_driver\_enable**

int serial\_set\_driver\_enable(void \*pInstance, serial\_driver\_enable fDE);

Программное управление приемопередатчиком для полудуплексных линий. В случае, когда порт должен работать в полудуплексном режиме (например, RS485 и т.п.), но не имеет аппаратных средств для переключения приемопередатчика, можно задать внешнюю функцию управления. Драйвер должен при начале передачи (вызов serial\_write) вызвать данную функцию с параметром 1 (переключение в режим передачи), а по ее окончанию – с параметром 0 (переключение в режим приема).

Данная функциональность может быть отключена для экономии ресурсов включением в **config.h**

|  |
| --- |
| #define CONFIG\_SERIAL\_SKIP\_RS485 |

Так как объявлена в файле **serial.h** в следующем виде:

|  |
| --- |
| #ifndef CONFIG\_SERIAL\_SKIP\_RS485  int serial\_set\_driver\_enable(void \*pInstance  serial\_driver\_enable fDE);  #endif |

Параметры:

|  |  |
| --- | --- |
| pInstance | указатель на управляющую структуру (для встроенного порта - результат функции \_\_system\_serial\_get(), для внешних – зависит от драйвера); |
| fDE | функция переключения приемопередатчика. Имеет следующую сигнатуру:  void (\*serial\_driver\_enable)(unsigned char ucDE);  Где ucDE – режим работы: 1 – передача, 0 – прием.  Сигнатура функции объявлена в файле **serial.h** |

5.5.3 **serial\_configure**

int serial\_configure(void \*pInstance,

unsigned long ulBaudrate,

unsigned char ucBits,

unsigned char ucParity,

unsigned char ucStops);

Конфигурирование последовательного интерфейса.

Параметры:

|  |  |
| --- | --- |
| pInstance | указатель на управляющую структуру, например, результат функции \_\_system\_serial\_get() |
| ulBaudrate | скорость обмена в битах в секунду (например, 115200) |
| ucBits | количество бит (5, 6, 7, 8) |
| ucParity | четность (отсутствует, четность, нечетность):  0 – нет четности  1 – нечетность  2 – четность |
| ucStops | количество стоповых битов (1, 2) |

Результат:

1 – успешно

иначе – аппаратно-специфичный код ошибки

Функция рассчитана на использование в том числе с последовательными портами, работа с которыми осуществляется посредством длительных процессов (например, обращение через SPI или иные шины). После того, как функция вернула 1, необходимо вызывать serial\_ready с параметром SERIAL\_OPERATION\_CONF (определен в **system/serial.h**)

rc = serial\_ready(pInstance, SERIAL\_OPERATION\_CONF);

до тех пор, пока не будет возвращено ненулевое значение. Значение 1 будет означать, что операция конфигурирования завершена успешно.

5.5.4 **serial\_available**

int serial\_available (void \*pInstance, unsigned \*puLen);

Запуск процесса проверки наличия принятых UART данных.

Параметры:

|  |  |
| --- | --- |
| pInstance | указатель на управляющую структуру, например, результат функции \_\_system\_serial\_get() |
| puLen | указатель на переменную для размещения количества имеющихся данных |

Результат:

0 – устройство занято, требуется повторить вызов позже

1 – успешно (процесс запущен)

иначе – аппаратно-специфичный код ошибки

После получения кода 1 необходимо осуществлять вызов serial\_ready с параметром SERIAL\_OPERATION\_AVAIL (определен в **system/serial.h**) до тех пор, пока процесс не будет завершен.

5.5.5 **serial\_read**

int serial\_read(void \*pInstance,

unsigned char \*pbBuffer,

unsigned \*puLen);

Запуск процесса чтения принятых UART данных.

Параметры:

|  |  |
| --- | --- |
| pInstance | указатель на управляющую структуру, например, результат функции \_\_system\_serial\_get() |
| pbBuffer | указатель на буфер для принимаемых данных |
| puLen | указатель на переменную, в которой будет передан размер буфера для чтения, а после успешного выполнения функции будет содержаться количество реально прочитанных данных |

Результат:

0 – устройство занято, требуется повторить вызов позже

1 – успешно (процесс запущен)

иначе – аппаратно-специфичный код ошибки

После получения кода 1 необходимо осуществлять вызов serial\_ready с параметром SERIAL\_OPERATION\_RX (определен в **system/serial.h**) до тех пор, пока процесс не будет завершен.

До завершения процесса изменение данных в буфере pbBuffer и переменной puLen не допускается.

5.5.6 **serial\_write**

int serial\_write(void \*pInstance,

unsigned char \*pbBuffer,

unsigned \*puLen);

Запуск процесса передачи данных в UART.

Параметры:

|  |  |
| --- | --- |
| pInstance | указатель на управляющую структуру, например, результат функции \_\_system\_serial\_get() |
| pbBuffer | указатель на буфер для передаваемых данных |
| puLen | указатель на переменную, в которой будет передан размер буфера для передачи, а после успешного выполнения функции будет содержаться количество реально переданных данных |

Результат:

0 – устройство занято, требуется повторить вызов позже

1 – успешно (процесс запущен)

иначе – аппаратно-специфичный код ошибки

После получения кода 1 необходимо осуществлять вызов serial\_ready с параметром SERIAL\_OPERATION\_TX (определен в **system/serial.h**) до тех пор, пока процесс не будет завершен.

До завершения процесса изменение данных в буфере pbBuffer и переменной puLen не допускается.

5.5.7 **serial\_ready**

int serial\_ready(void \*pInstance, unsigned char ucOperation);

Функция возвращает информацию о готовности последовательного интерфейса (завершении выполнения определенной операции).

Так как в общем работа с последовательным интерфейсом допускает одновременное выполнение операций (например, чтения и записи), вызов serial\_ready должен осуществляться с указанием именно той операции, которая была запущена ранее конкретным компонентом/задачей.

Предусмотрены (serial.h) следующие варианты:

|  |
| --- |
| #define SERIAL\_OPERATION\_RX  #define SERIAL\_OPERATION\_TX  #define SERIAL\_OPERATION\_CONF  #define SERIAL\_OPERATION\_AVAIL |

Параметры:

|  |  |
| --- | --- |
| pInstance | указатель на управляющую структуру, например, результат функции \_\_system\_serial\_get() |
| ucOperation | операция |

Результат:

0 – устройство занято, процесс еще продолжается

1 – успешно (процесс закончен, можно интерпретировать данные в буферах)

иначе – аппаратно-специфичный код ошибки

## 5.6 Подсистема получения временных отсчетов

ППП МК предоставляет прикладному программисту готовый механизм для временных отсчетов в микросекундах.

Инициализация данного механизма производится среди базовых низкоуровневых инициализационных процедур и к моменту вызова app\_initialize() (инициализационные процедуры прикладной программы, см. 3.3) он уже полностью функционален.

unsigned long \_\_micros(void);

Функция получения текущего времени в микросекундах с момента включения устройства.

Параметры: нет

Особенности:

* Отсчет времени ведется от момента старта устройства, но на самом деле от момента запуска соответствующего таймера, что может иметь некоторое (до нескольких миллисекунд) запаздывание в зависимости от процедуры инициализации (например для платформы STM32 ее код генерируется STM CubeMX);
* Функция возвращает 32-битное беззнаковое значение, что означает, что примерно раз в 1,2 часа будет происходить его переполнение. Это необходимо учитывать, рассчитывая длительность прошедшего времени, например, так:

#define UL\_POSITIVE(ul) ((ul) < (((unsigned long)(-1))>>1))

if (UL\_POSITIVE(\_\_micros() - ulWaitTO)) {

. . .

}

* Для некоторых устройств (по причине недостаточной производительности или особенностям реализации) может иметь гранулярность не в 1 мкс, а больше (например, 5-10 мкс); при необходимости следует ознакомиться с реализацией на конкретной платформе;

Также при разработке прикладных программ необходимо учитывать, что функция \_\_micros может быть реализована разными способами, а именно:

1. Посредством прерываний. Микроконтроллер содержит таймер, настроенный на отсчет с большой частотой и формированием прерывания с коротким обработчиком, который осуществляет инкремент глобальной переменной текущего времени. Сама функция \_\_micros просто возвращает текущее значение этой глобальной переменной.

Данный вариант позволяет прикладной программе сколь угодно часто обращаться к \_\_micros, которая выполняется очень быстро. Однако имеет и недостатки: функция будет ограниченно функциональна при вызове из обработчика прерываний (когда прерываний запрещены она всегда будет возвращать одно и то же значение), а также сам микроконтроллер может быть значительно нагружен вызовами обработчика прерывания таймера, в связи с чем может потребоваться сделать их вызов реже с соответствующим снижением точности.

2. Посредством считывания значения непосредственно из регистров аппаратного таймера в контексте вызова \_\_micros.

\_\_micros(), реализованный посредством такого подхода, можно полноценно использовать из контекста прерываний. Однако чтение регистров таймера может вносить накладные расходы (время на непосредственное чтение значений, плюс в некоторых случаях пересчет в микросекунды, например, если таймер считает в наносекундах).

В общем случае рекомендуется не использовать \_\_micros для измерения промежутков времени в прерываниях, не использовать слишком часто и не полагаться на точность до микросекунды. Если же что-то из этого требуется, разработчику прикладной программы необходимо ознакомиться с деталями реализации \_\_micros на текущей платформе и/или реализовать задачу с использованием средств доступных аппаратных таймеров.

## 5.7 Менеджер ресурсов

Одной из задач, решаемых при разработке программного обеспечения, является обеспечение максимально слабой связанности между компонентами, что в свою очередь позволяет существенно упростить структуру ПО и снизить количество ошибок. В операционных системах такая задача обычно решается введением понятий процессов и потоков, выполняющихся псевдопараллельно (в случае одного процессорного ядра).

ППП МК содержит программные средства, позволяющие разделять доступ к ресурсам микроконтроллера (прежде всего коммуникационным шинам), предоставляя компонентам программы возможность иметь независимую друг от друга логику функционирования.

В качестве типовых ситуаций можно привести пример, когда устройство содержит расширение GPIO с индикацией и ПЗУ (EEPROM), подключенных к одной шине I2C микроконтроллера или несколько независимо работающих устройств на одной шине SPI.

Программным средством, обеспечивающим захват и освобождение ресурсов компонентами, является ***менеджер ресурсов***. Его код содержится в файлах **/core/res\_manager.h** и **/core/res\_manager.c**.

Менеджер ресурсов предназначен для использования компонентами, реализующими концепцию кооперативной многозадачности на одном процессорном ядре. Это обеспечивает отсутствие конкурентности и, таким образом, отсутствие необходимости синхронизации доступа к интерфейсу менеджера ресурсов, упрощая его реализацию.

Однако это вводит и ограничение: **работать с менеджером ресурсов из контекста прерываний нельзя!**

Работа с менеджером ресурсов осуществляется с помощью команд-макросов и специальных структур данных, далее будет показано на примере SPI:

// Providing space for a context.

// For cooperative multitasking component

// it should NOT be in stack!

res\_spi\_ctx sSPICtx;

sSPICtx.ucChannel = 1; // F.e., we need SPI1

// FSM should yield in fixed state while SPI is not acqiured

if (!ACQUIRE\_SPI(&sSPICtx)) return;

FSM.ucState++;

… component performs SPI transactions

RELEASE\_SPI(&sSPICtx);

FSM.ucState++;

Макрос ACQUIRE\_SPI() в случае, если шина свободна, помечает ее занятой, сохраняет указатель на ее управляющую структуру (см. spi\_get ()) и возвращает ненулевое значение. Компонент, захвативший таким образом шину, может перейти в состояние работы, осуществляя необходимые транзакции.

В случае, когда шина занята, структура res\_spi\_ctx захватывающего компонента помещается в очередь, а макрос возвращает нулевое значение, в соответствии с которым компонент должен вернуть управление обратно в основной цикл, не меняя своего состояния чтобы повторить попытку позже.

Функция менеджера ресурсов, реализующая ACQUIRE\_SPI, проверяет наличие переданной структуры res\_spi\_ctx в очереди, что делает безопасным ее многократный вызов и упрощает логику работы компонента.

По завершению работы с ресурсом необходимо использовать макрос RELEASE\_SPI(), который передаст указатель на управляющую структуру следующему компоненту в очереди (если очередь пуста, то шина будет помечена свободной). При этом в структуре res\_spi\_ctx отдающего компонента поле соответствующего указателя будет обнулено.

Использование макросов позволяет сделать для компонента прозрачным работу как в приложении, где осуществляется управление доступом к ресурсам, так и в приложении, где такого управления не требуется. Для этого достаточно определить макросы ACQUIRE\_SPI и RELEASE\_SPI пустыми (но сохраняющими возврат ненулевого значения) и сразу записать ненулевой указатель на управляющую структуру шины SPI (spi\_get()) в res\_spi\_ctx. Это требуется потому, что все драйверы устройств, разрабатываемые для ППП МК, включают в логику работы с ресурсами макросы менеджера ресурсов.

В текущей версии менеджер ресурсов ППП МК обеспечивает разделение доступа к шинам SPI (res\_spi\_ctx, ACQUIRE\_SPI, RELEASE\_SPI) и I2C (res\_i2c\_ctx, ACQUIRE\_I2C, RELEASE\_I2C).

Для конфигурирования предусмотрены определения

CONFIG\_RM\_SHARE\_SPI – будет разделение доступа к шинам SPI

CONFIG\_RM\_SHARE\_I2C – будет разделение доступа к шинам I2C

Данные определения необходимо включать в файл **config.h**

#define CONFIG\_RM\_SHARE\_I2C 1

#define CONFIG\_RM\_SHARE\_SPI 1

Менеджер ресурсов для организации очереди ожидания использует единый статически выделяемый блок памяти на количество ожидающих компонентов, заданных значением RW\_WAITERS (см. **res\_manager.h**). По умолчанию это значение равно 20. Если устройство предполагает использование большего числа компонентов, которые могут одновременно ожидать захвата шины, это значение следует увеличить.

# 6 Особенности К5500ВК018

## 6.1 Архитектурные особенности

Микроконтроллер К5500ВК018 (Комдив-МК) содержит следующие архитектурные особенности, влияющие на реализацию программного обеспечения.

6.1.1 Микроконтроллер имеет развитую систему кэширования оперативной памяти, при этом в соответствии с архитектурой MIPS, использует разные диапазоны виртуальных адресов для обращения к одним и тем же ячейкам памяти через систему кэширования или без нее.

Использование кэш-памяти существенно повышает производительность системы, ее инициализация выполняется в системном загрузчике или в программе начальной инициализации аппаратуры.

Однако, при работе со встроенными контроллерами, регистры которых отображены в общее адресное пространство, и с подсистемой DMA необходимо обращение к памяти в обход кэш-памяти, то есть в данном случае через соответствующий диапазон виртуальных адресов. Использование для этого ключевого слова **volatile** (как, например, в компиляторах для архитектуры ARM) не приведет к требуемому результату, необходимо явно приводить адреса к некэшируемому диапазону:

|  |
| --- |
| unsigned char \*pbBuf = (unsigned char \*)TO\_UNCACHED(pbBuf); |

*Замечание*: ключевое слово volatile при определении данных компилятором **gcc** для MIPS воспринимается как указание не оптимизировать работу с этими данными, что и требуется при работе с регистрами внешних устройств. Никаких трансформаций адресов при этом не делается.

6.1.2 Микроконтроллер имеет одно вычислительное ядро, но при этом имеет средства распараллеливания выполняемых задач в виде нескольких подсистем DMA:

- подсистема DMA, обеспечивающая трансфер данных в/из оперативной памяти для последовательных портов (UART), аналого-цифровых преобразователей (ADC), а также имеющая режимы для копирования данных между участками ОЗУ;

- отдельная подсистема DMA для контроллера SPI.

Драйверы Комдив-МК дают возможность использования DMA. Необходимые для работы DMA данные указываются в функциях инициализации (см. далее описание функций kmk\_spi\_init, kmk\_serial\_init)

Непосредственное обращение прикладной программы к контроллеру DMA не предполагается, вся работа с ним осуществляется в драйверах Комдив-МК. Тем не менее, для эффективной работы прикладной программы необходимо контролировать параллельно запущенные процессы, задействующие DMA, так как в случае занятости шины одним процессом другие будут простаивать, ожидая ее освобождения.

6.1.3 В последовательных портах (UART) микроконтроллера присутствует встроенное FIFO размером 255 байт (отдельно на прием и на передачу), при этом FIFO может быть использовано совместно с DMA. Драйвер UART Комдив-МК поддерживает эту связку.

Также UART Комдив-МК содержит встроенный механизм управления переключением направления (прием-передача) для полудуплексных коммуникаций (например, RS485). Для этого при разработке схемотехнического решения требуется соединение сигнала RTS соответствующего UART с контактом Driver Enable приемо-передатчика RS485, а также указание флага SERIAL\_FLAG\_AUTO\_RTS в функции инициализации UART (см. далее описание функции kmk\_serial\_init).

6.1.4 Программа микроконтроллера Комдив-МК размещается во внешней по отношению к микроконтроллеру флеш-памяти, подключающейся посредством шины SPI. Микроконтроллер содержит специализированную (выделенную) шину SPI (QSPI) для обслуживания такой флеш-памяти и специальные средства отображения этой флеш-памяти на пространство адресуемой памяти, доступной вычислительному ядру. Это позволяется осуществить старт программы, однако скорость ее выполнения будет низкой из-за необходимости постоянно выполнять запросы по SPI для выборки команд.

По данной причине инициализационная часть ПО содержит код, осуществляющий копирование всего исполняемого кода в оперативную память микроконтроллера.

Время, требующееся на выполнение программы из внешней флеш-памяти до момента перехода на исполнение из ОЗУ (включая инициализацию системы кэширования и копирование кода и данных) составляет около 20 мс.

## 6.2 Определения config.h

Файл **config.h** предусмотрен для размещения глобальных определений, которые влияют на все компоненты и/или на системные и низкоуровневые компоненты, для которых непосредственное вмешательство в код нежелательно.

Для проектов, которые должны исполняться на микроконтроллере Комдив-МК, предусмотрены следующие определения:

|  |
| --- |
| #define CONFIG\_KMK\_BOARD 1 |

Определение CONFIG\_KMK\_BOARD указывает, что проект собирается для микроконтроллера Комдив-МК и позволяет учесть это на уровне прикладной программы: например, задействовать дополнительную функциональность (присутствующую в Комдив-МК) или оптимизировать логику каких-либо компонентов.

|  |
| --- |
| #define \_\_ARCH\_INCLUDE\_\_ "kmk\_uke.h" |

Данное определение включает в проект заголовочный файл, который в свою очередь содержит параметры для конкретного микроконтроллера, такие, как адреса отображенных в адресное пространство регистров контроллеров, а также имеет возможность переопределить некоторые «универсальные» реализации и заменить их специфическими для Комдив-МК (например, замена некоторых функций макросами).

|  |
| --- |
| #define CONFIG\_RM\_FROM\_ZERO 1 |

Определение CONFIG\_RM\_FROM\_ZERO включается для выделения ресурсов (памяти) под управляющие структуры для контроллеров шин, имеющих номер 0 (SPI0, I2C0, UART0). Комдив-МК нумерует все контроллеры с нуля, но некоторые из контроллеров (SPI0, UART0) имеют ряд ограничений (например, UART0 не может работать с DMA), и если в проекте важнее сэкономить память, можно отказаться от выделения памяти под эти компоненты.

|  |
| --- |
| #define CONFIG\_KMK\_INIT\_QSPI\_PLAIN\_DUAL\_IO 1 |

Микроконтроллер Комдив-МК размещает программу во внешнем относительно микроконтроллера ПЗУ (подключаемой посредством QSPI). При этом выполнение программы непосредственно из ПЗУ означает, что для выполнения кода приходится осуществлять «прозрачную» выборку инструкций по шине QSPI. При старте микроконтроллера шина QSPI, к которой подключается ПЗУ, работает в «совместимом» режиме SPI, не задействуя предусмотренные QSPI дополнительные линии. Если микросхема ПЗУ поддерживает работу в более производительных режимах QSPI, загрузчик может перевести системный контроллер в такой более производительный режим.

Определение CONFIG\_KMK\_INIT\_QSPI\_PLAIN\_DUAL\_IO переводит системный контроллер в режим запросов QSPI DUAL IO.

|  |
| --- |
| #define CONFIG\_KMK\_INIT\_COPY\_DMA\_QUAD\_IO 1 |

Как было указано выше, микроконтроллер Комдив-МК размещает программу во внешнем относительно микроконтроллера ПЗУ (когда для выполнения кода приходится осуществлять выборку инструкций по шине QSPI), в связи с чем программа будет выполняться намного медленнее, нежели из ОЗУ. Для достижения высокой производительности входящий в состав ПО загрузчик при старте устройства осуществляет копирование программы из ПЗУ в ОЗУ.

Определение CONFIG\_KMK\_INIT\_COPY\_DMA\_QUAD\_IO указывает загрузчику, что копирование необходимо производить посредством DMA, при этом используя самый высокопроизводительный режим контроллера QSPI (QUAD IO). Если аппаратура (в том числе примененная микросхема ПЗУ) поддерживает такой режим, ускорение копирования может достигать десятков и даже сотен раз.

## 6.3 GPIO

Микроконтроллер Комдив-МК имеет 6 контроллеров ввода-вывода общего назначения (GPIO) по 8 линий (всего 48). Каждый контроллер GPIO имеет набор регистров, отображенных в адресное пространство микроконтроллера.

В файле **arch/kmk\_common.h** определено несколько макросов для манипуляции GPIO.

6.3.1 **GPIO\_SetPinMode**

GPIO\_SetPinMode(GPIO, pin, mode)

Макрос устанавливает режим работы конкретной линии порта GPIO. Режимы работы – порт работает на выход (на нем можно программно устанавливать уровень сигнала – 0 или 1), на вход, либо предоставляется определенному контроллеру (SPI, I2C и т.д.) для выполнения специфических функций.

Параметры:

|  |  |  |
| --- | --- | --- |
| GPIO | указание на контроллер, для линии которого требуется установить режим. Для 6 контроллеров в файле **arch/kmk/uke.h** предусмотрены обозначения:   |  | | --- | | #define GPIOA\_BASE  . . .  #define GPIOF\_BASE | |
| pin | номер линии, целое число от 0 до 7 |
| mode | режим работы линии. Предусмотрены (**arch/kmk\_common.h**) следующие режимы:   |  | | --- | | #define GPIO\_MODE\_INPUT  #define GPIO\_MODE\_INPUT\_PULLUP  #define GPIO\_MODE\_OUTPUT  #define GPIO\_MODE\_ALT1  #define GPIO\_MODE\_ALT2  #define GPIO\_MODE\_ALT3  ... | |

6.3.2 **GPIO\_WritePin**

GPIO\_WritePin(GPIO, pin, state)

Макрос устанавливает сигнал на конкретной линии. Для того, чтобы его использование имело смысл, линия должна быть установлена в режим OUTPUT (GPIO\_MODE\_OUTPUT).

Параметры:

|  |  |  |
| --- | --- | --- |
| GPIO | указание на контроллер, для линии которого требуется установить режим. Для 6 контроллеров в файле **arch/kmk/uke.h** предусмотрены обозначения:   |  | | --- | | #define GPIOA\_BASE  . . .  #define GPIOF\_BASE | |
| pin | номер линии, целое число от 0 до 7 |
| state | требуемый уровень: 0 или 1 (любое ненулевое значение интерпретируется как 1) |

6.3.3 **GPIO\_ReadPin**

GPIO\_ReadPin(GPIO, pin)

Макрос позволяет получить состояние (уровень сигнала) конкретной линии. Как правило, используется для линий с режимом GPIO\_MODE\_INPUT и GPIO\_MODE\_INPUT\_PULLUP (входная линия с подтяжкой к высокому уровню).

Параметры:

|  |  |  |
| --- | --- | --- |
| GPIO | указание на контроллер, для линии которого требуется установить режим. Для 6 контроллеров в файле **arch/kmk/uke.h** предусмотрены обозначения:   |  | | --- | | #define GPIOA\_BASE  . . .  #define GPIOF\_BASE | |
| pin | номер линии, целое число от 0 до 7 |

Макрос возвращает значение 0 или 1, например

|  |
| --- |
| unsigned char ucState =  GPIO\_ReadPin(GPIOA\_BASE, 2); //значение PA2 (GPIOA, пин 2) |

Программным обеспечением не предусматривается создание абстракций GPIO, а рекомендуется написание функций в домене предметной области, например:

|  |  |  |
| --- | --- | --- |
| Функция, включающая или выключающая индикацию сбоя (предполагается, что на плате к пину PF7 подключен светодиод, зажигающийся высоким уровнем):   |  | | --- | | void led\_failure(unsigned char bOn)  {  GPIO\_WritePin(GPIOF\_BASE, 7, bOn);  } |   Функция чтения состояния (нажатия) кнопки, предполагается, что на плате кнопка подключена к пину PD6 и нажатие на кнопку создает высокий уровень на пине:   |  | | --- | | unsigned char button\_state(void)  {  return GPIO\_ReadPin(GPIOD\_BASE, 6);  } | |

## 6.4 Функции инициализации аппаратуры

6.4.1 kmk\_i2c\_init

void kmk\_i2c\_init(unsigned char ucNum,

void \*pI2CBase,

unsigned long ulFreq);

Функция инициализации интерфейса I2C.

Функция выполняется однократно и производит инициализацию контроллера I2C микроконтроллера Комдив-МК, а также инициализирует область памяти для хранения управляющей структуры. Как правило, вызов функции производится из файла **main.c/board.c**

Параметры:

|  |  |
| --- | --- |
| uсNum | номер шины; может быть равен 0, если для микроконтроллера нумерация шин осуществляется с 0 (например, есть I2C0). |
| pI2CBase | адрес контроллера (определения I2C0\_BASE, I2C1\_BASE и т.д. находятся в файле **arch/kmk/uke.h**) |
| ulFreq | частота; стандартом предусмотрены 100000, 400000, при этом контроллер I2C Комдив-МК позволяет работать на большем спектре частот |

Функция kmk\_i2c\_init НЕ осуществляет настройку линий GPIO, что необходимо предварительно делать, если шина I2C подключается к мультиплексированным пинам.

Пример:

|  |
| --- |
| GPIO\_SetPinMode(GPIOC\_BASE, 0, GPIO\_MODE\_ALT1);  GPIO\_SetPinMode(GPIOC\_BASE, 1, GPIO\_MODE\_ALT1);  kmk\_i2c\_init(1, KMK\_ADDR(I2C1\_BASE), 400000); |

После выполнения kmk\_i2c\_init() появляется возможность использования функции i2c\_get() для проинициализированной шины; для примера выше

|  |
| --- |
| void \*pI2C1 = i2c\_get(1); |

Область памяти, указатель на которую будет возвращаться (pI2C1), статически выделяется и инициализируется функцией kmk\_i2c\_init.

6.4.2 kmk\_spi\_init

void kmk\_spi\_init(unsigned char ucNum,

void \*pSPIBase,

unsigned char ucBaudrate,

unsigned char ucFlags,

unsigned char ucIRQ,

unsigned char \*pbDMABuf,

unsigned short usDMABufSize,

unsigned long ulDMABufHi,

unsigned long ulDMABufLo);

Функция инициализации интерфейса SPI.

Функция выполняется однократно и производит инициализацию контроллера SPI микроконтроллера Комдив-МК, а также инициализирует область памяти для хранения управляющей структуры. Как правило, вызов функции производится из файла **main.c/board.c**.

Контроллеры SPI микроконтроллера Комдив-МК имеют в составе собственную подсистему DMA, поэтому функция kmk\_spi\_init содержит большой набор параметров, необходимых в том числе и для инициализации этой подсистемы.

Параметры:

|  |  |
| --- | --- |
| uсNum | номер шины; может быть равен 0, если для микроконтроллера нумерация шин осуществляется с 0 (например, есть SPI0) |
| pSPIBase | адрес контроллера (определения SPI0\_BASE, SPI1\_BASE и т.д. находятся в файле **arch/kmk/uke.h**) |
| ucBaudrate | значение, позволяющее задать частоту сигнала CLK SPI. Подробнее см. ниже. |
| ucFlags | флаги, поддерживаются следующие:   |  | | --- | | #define KMK\_SPI\_FLAG\_USE\_IRQ 1 |   Если не указан, контроллер SPI не будет использовать прерывания (см. схемы работы функции spi\_io\_ready()) |
| ucIRQ | номер линии IRQ (в случае, если используются прерывания); номера линий IRQ всех контроллеров определяются в файле **arch/kmk/uke.h** |
| pbDMABuf | виртуальный адрес буфера DMA в некешируемом адресном пространстве микроконтроллера |
| usDMABufSize | размер буфера DMA в байтах |
| ulDMABufHi | старшая часть физического адреса буфера DMA (для 32-битного режима Комдив-МК равен 0) |
| ulDMABufLo | старшая часть физического адреса буфера DMA (контроллер DMA обращается к памяти иначе, чем ядро MIPS, поэтому данный адрес отличается от pbDMABuf, см. ниже) |

Для задания частоты CLK SPI задается делитель, рассчитываемый по специальной формуле и позволяющий получить необходимую (или близкую к ней) частоту из частоты тактирования шин самого микроконтроллера. Это значение можно вычислить заранее, либо вызвать функцию kmk\_spi\_calc\_baudrate

|  |
| --- |
| unsigned char kmk\_spi\_calc\_baudrate(unsigned long ulAXIBusFreq, unsigned long ulReqSPIFreq); |

с параметрами

ulAXIBusFreq – частота шины микроконтроллера (например, 150000000)

ulReqSPIFreq – требующаяся частота CLK SPI

Например:

|  |
| --- |
| ucSPIBaudrate = kmk\_spi\_calc\_baudrate(150000000, 5500000); |

При этом необходимо иметь в виду, что расчет делителя осуществляется алгоритмом приближения в цикле и занимает время, поэтому следует рассчитать его однократно заранее и затем передавать уже готовым.

Работа по SPI для Комдив-МК реализована следующим образом:

- буфер передачи (если есть) целиком или частично копируется в буфер DMA (размер буфера DMA выбирается исходя из протокола обмена с реальными устройствами, нет необходимости делать его большим даже если может идти обмен большими блоками данных)

- запускается процесс DMA, в контексте которого данные из буфера выдвигаются на шину SPI, а принятые данные копируются в буфер, замещая там отправленные данные;

- когда все данные из буфера отправлены (и в случае, если обмен предполагает обработку принятых данных), данные из буфера DMA копируются в буфер-приемник

- процесс повторяется до тех пор, пока не будет передано и принято изначально требуемое количество байт (размер буфера DMA – это размер «порции» данных, обрабатываемых за один процесс DMA); при этом при использовании прерываний подготовка и старт следующей порции будет выполнен из контекста обработчика прерывания, что сократит простой шины до очередного вызова функции spi\_io\_ready()

Функция kmk\_spi\_init НЕ осуществляет настройку линий GPIO, что необходимо предварительно делать, если шина SPI подключается к мультиплексированным пинам.

Пример инициализации SPI (шина SPI1)

|  |
| --- |
| #define SPI\_DMA\_BUFFER\_SIZE 16  unsigned char \*pbBuf, ucFlags, ucSPIBaudrate;  unsigned long ulDMAHi = 0, ulDMALo;  GPIO\_SetPinMode(GPIOA\_BASE, 3, GPIO\_MODE\_ALT1);  GPIO\_SetPinMode(GPIOA\_BASE, 5, GPIO\_MODE\_ALT1);  GPIO\_SetPinMode(GPIOA\_BASE, 6, GPIO\_MODE\_ALT1);  pbBuf =  (unsigned char \*)res\_manager\_smalloc(SPI\_DMA\_BUFFER\_SIZE);  ulDMALo = (unsigned long)(CPHYSADDR(pbBuf));  pbBuf = (unsigned char \*)CKSEG1ADDR(pbBuf);  ucSPIBaudrate = 0x16;  //kmk\_spi\_calc\_baudrate(150000000, 5500000);  kmk\_spi\_init(1, KMK\_ADDR(SPI1\_BASE), ucSPIBaudrate, KMK\_SPI\_FLAG\_USE\_IRQ, IRQ\_SPI1, pbBuf, SPI\_DMA\_BUFFER\_SIZE, ulDMAHi, ulDMALo); |

После выполнения kmk\_spi\_init() появляется возможность использования функции spi\_get() для проинициализированной шины; для примера выше

|  |
| --- |
| void \*pSPI1 = spi\_get(1); |

Область памяти, указатель на которую будет возвращаться (pSPI1), статически выделяется и инициализируется функцией kmk\_spi\_init.

6.4.3 kmk\_serial\_init

void kmk\_serial\_init(unsigned char ucNum,

void \*pUARTBase,

unsigned char ucFlags,

unsigned char ucIRQ,

unsigned char ucDMAUARTNum,

unsigned char ucDMAChannel,

unsigned char \*pbDMABuf,

unsigned short usDMABufSize,

unsigned long ulDMABufHi,

unsigned long ulDMABufLo);

Функция инициализации системного последовательного порта (UART).

Функция выполняется однократно и производит инициализацию системного последовательного порта микроконтроллера Комдив-МК, а также инициализирует область памяти для хранения управляющей структуры. Как правило, вызов функции производится из файла **main.c/board.c**.

Контроллеры UART микроконтроллера Комдив-МК (за исключением UART0) имеют поддержку работы через подсистему DMA, поэтому функция kmk\_serial\_init содержит большой набор параметров, необходимых в том числе и для настройки этого режима.

Параметры:

|  |  |
| --- | --- |
| uсNum | номер контроллера; может быть равен 0, если для микроконтроллера нумерация последовательных портов осуществляется с 0 (например, есть UART0) |
| pUARTBase | адрес контроллера (определения UART0\_BASE, UART1\_BASE и т.д. находятся в файле **arch/kmk/uke.h**) |
| ucFlags | флаги, предусмотрены следующие:  использование DMA для отправляемых данных   |  | | --- | | #define KMK\_SERIAL\_FLAG\_DMA\_TX |   использование автоматического переключения приема-передачи для полудуплексных каналов; необходимо схематехническое подключение сигнала RTS UART ко входу Driver Enable приемопередатчика   |  | | --- | | #define KMK\_SERIAL\_FLAG\_AUTO\_RTS |   использование прерывания при приеме данных (если нет – данные просто помещаются в FIFO без уведомлений)   |  | | --- | | #define KMK\_SERIAL\_FLAG\_IRQ\_RX |   использование прерывания при отправке данных (может быть полезным при необходимости переслать большой объем данных, а размер FIFO всего 255 байт)   |  | | --- | | #define KMK\_SERIAL\_FLAG\_IRQ\_TX | |
| ucIRQ | номер линии IRQ (см. **arch/kmk/uke.h**, если прерывания не используются, рекомендуется указывать 0) |
| ucDMAUARTNum | номер канала UART в системе DMA Комдив-МК (см. документацию на подсистему DMA Комдив-МК) |
| ucDMAChannel | номер канала DMA, который будет использоваться; всего доступно 12 каналов (0-11) |
| pbDMABuf | адрес буфера DMA в адресном пространстве микроконтроллера (должен быть адресом, определяющим некешируемую память) |
| usDMABufSize | размер буфера DMA в байтах |
| ulDMABufHi | старшая часть физического адреса буфера DMA (для 32-битного режима Комдив-МК равен 0) |
| ulDMABufLo | старшая часть физического адреса буфера DMA (контроллер DMA обращается к памяти иначе, чем ядро MIPS, поэтому данный адрес отличается от pbDMABuf, см. ниже) |

Для kmk\_serial\_init предусмотрен набор параметров, которые могут (при необходимости) быть заданы в проекте (**config.h** или **Makefile**):

|  |
| --- |
| #define KMK\_UART\_DMA\_SUPPORT 1 |

Если задан, то в проект будет включен код работы UART с DMA, соответствующие параметры и флаги будут интерпретироваться. Если этого не требуется, то можно сэкономить память, задав

|  |
| --- |
| #define KMK\_UART\_DMA\_SUPPORT 0 |

В отличие от функций kmk\_i2c\_init и kmk\_spi\_init, функция kmk\_serial\_init имеет средства настройки пинов (в том числе для настройки пина RTS для режима AUTO\_RTS). Для этого предусмотрен следующий параметр:

|  |
| --- |
| #define KMK\_UART\_MAPPING\_ENABLE 1 |

Если предполагается задать режимы работы пинов отдельно или самостоятельно, следует задать

|  |
| --- |
| #define KMK\_UART\_MAPPING\_ENABLE 0 |

Пример инициализации UART для работы в режиме RS485:

|  |
| --- |
| #define UART\_DMA\_BUFFER\_SIZE 16  unsigned char \*pbBuf, ucFlags;  unsigned long ulDMAHi = 0, ulDMALo;  ucFlags = KMK\_SERIAL\_FLAG\_IRQ\_RX | KMK\_SERIAL\_FLAG\_IRQ\_TX |  KMK\_SERIAL\_FLAG\_DMA\_TX | KMK\_SERIAL\_FLAG\_AUTO\_RTS;  pbBuf =  (unsigned char \*)res\_manager\_smalloc(UART\_DMA\_BUFFER\_SIZE);  ulDMALo = (unsigned long)(CPHYSADDR(pbBuf));  pbBuf = (unsigned char \*)CKSEG1ADDR(pbBuf);  kmk\_serial\_init(1, KMK\_ADDR(UART1\_BASE), ucFlags, IRQ\_UART1, 0, 0, pbBuf, UART\_DMA\_BUFFER\_SIZE, ulDMAHi, ulDMALo);  \_\_system\_serial\_configure(\_\_system\_serial\_get(1), 115200, 8, 0, 1); |

Как видно в примере, после выполнения kmk\_serial\_init() появляется возможность использования функции \_\_system\_serial\_get() для проинициализированной шины. После чего требуется выполнение функции \_\_system\_serial\_configure().

Область памяти, указатель на которую будет возвращаться (\_\_system\_serial\_get()), статически выделяется и инициализируется функцией kmk\_serial\_init.

## 6.5 Файлы main.c/board.c

Файл **main.c** (или **board.c**) проектов, предназначенных для микроконтроллера Комдив-МК содержит следующие элементы:

- функции, имеющие прикладную семантику, при этом реализованные специфично для конкретной платы (примеры см. п. 6.3 GPIO);

- функция init\_board(), выполняющая инициализацию всех компонентов конкретной платы, в первую очередь микроконтроллера Комдив-МК: контроллеры I2C, SPI, UART и т.д. Рекомендуется скопировать код из примеров и самостоятельно внести в него необходимые правки;

- функция main(), в которую передается управление после завершения инициализации микроконтроллера загрузчиком, написанным на ассемблере. В общем случае в функции main для платы с микроконтроллером Комдив-МК выглядит следующим образом:

|  |
| --- |
| int main(void)  {  init\_board();  app\_initialize();  while(1) {  app\_cycle\_tick();  };  // never reacheable  return 1;  } |

- обработчики прерываний (если проект предусматривает работу с прерываниями). Методика и средства для работы с прерываниями описаны в следующих разделах.

Примерный вид **main.c** для проекта для Комдив-МК:

|  |
| --- |
| #include "app/app.h"  #include "arch/arch.h"  #include "system/serial.h"  #include "arch/kmk\_common.h"  ...  // Board-specific functions  void led\_run(unsigned char bOn)  {  GPIO\_WritePin(GPIOB\_BASE, 1, bOn);  }  void led\_failure(unsigned char bOn)  {  GPIO\_WritePin(GPIOB\_BASE, 2, bOn);  }  ...  // Board initialization  static inline int init\_board(void)  {  ...  // Add our handler to INT2  // We'll use INT2 for UART/I2C/SPI/DMA interrupts  kmk\_set\_int\_vector(2, &kmk\_my\_int2);  GPIO\_SetPinMode(GPIOB\_BASE, 1, GPIO\_MODE\_OUTPUT); // RUN  GPIO\_WritePin(GPIOB\_BASE, 1, 1);  GPIO\_SetPinMode(GPIOB\_BASE, 2, GPIO\_MODE\_OUTPUT); // FAIL  GPIO\_WritePin(GPIOB\_BASE, 2, 1);  GPIO\_SetPinMode(GPIOC\_BASE, 6, GPIO\_MODE\_INPUT\_PULLUP); // Button  kmk\_serial\_init(0, KMK\_ADDR(UART0\_BASE), KMK\_SERIAL\_FLAG\_IRQ\_TX, IRQ\_UART0, 0, 0, 0, 0, 0, 0);   \_\_system\_serial\_configure(\_\_system\_serial\_get(0), 115200, 8, 0, 1);  kmk\_init\_int\_line(IRQ\_UART0, 2, 0);  ...  // SPI1  GPIO\_SetPinMode(GPIOA\_BASE, 3, GPIO\_MODE\_ALT1);  GPIO\_SetPinMode(GPIOA\_BASE, 5, GPIO\_MODE\_ALT1);  GPIO\_SetPinMode(GPIOA\_BASE, 6, GPIO\_MODE\_ALT1);    pbBuf =  (unsigned char \*)  res\_manager\_smalloc(SPI\_DMA\_BUFFER\_SIZE);  ulDMALo = (unsigned long)(CPHYSADDR(pbBuf));  pbBuf = (unsigned char \*)CKSEG1ADDR(pbBuf);  // SPI baudrate precalculated.  ucSPIBaudrate = 0x16;  //kmk\_spi\_calc\_baudrate(150000000, 5500000);  kmk\_spi\_init(1, KMK\_ADDR(SPI1\_BASE),  ucSPIBaudrate,  KMK\_SPI\_FLAG\_USE\_IRQ, IRQ\_SPI1, pbBuf,  SPI\_DMA\_BUFFER\_SIZE, ulDMAHi, ulDMALo);  kmk\_init\_int\_line(IRQ\_SPI1, 2, 1);    ...  }  // main function  int main(void)  {  init\_board();  app\_initialize();  while(1) {  app\_cycle\_tick();  };  // never reacheable  return 1;  } |

## 6.6 Обработка прерываний

В соответствии с архитектурой MIPS ядро микропроцессора имеет 6 входов внешних прерываний – INT0-INT5. В микроконтроллере К5500ВК018 есть контроллер прерываний на 48 источников запросов – IRQ0-IRQ47, отображаемых на любую из шести линий прерывания микропроцессорного ядра. Распределение устройств-источников прерываний по 48 линиям фиксировано в аппаратуре и отражено в файле **arch/kmk/uke.h**. Отображение прерывания от любого из 48 источников на одну из шести входных линий ядра выполняется в контроллере прерываний при инициализации контроллера прерываний функцией

|  |
| --- |
| void kmk\_init\_int\_line(unsigned char IRQ, /\* 0..47 \*/  unsigned char INT, /\* 0..5 \*/  unsigned char Priority) /\* 0..47(hi) \*/ |

*Замечание 1*: в соответствии с архитектурой MIPS на линию INT5 наряду с внешними запросами подключены также запросы прерываний от внутренних счетчиков производительности и таймеров, поэтому использовать линию INT5 для прикладных задач не рекомендуется.

*Замечание 2*: необходимо помнить, что для запросов IRQ, направленных на одну и ту же линию INT, приоритеты должны быть различны, иначе корректная работа контроллера прерываний не обеспечивается (см. Указания по применению К5500ВК018).

ППП МК предлагает два варианта обработки прерываний – обработчиком на языке ассемблера и обработчиком на языке Си (или С++, в этом случае необходимо точно понимать как выполняются используемые абстракции С++, поскольку за ними может стоять весьма активная и не всегда ожидаемая работа с кучей и другими ресурсами).

Обработчик прерывания, написанный на языке ассемблера, необходим в случаях, когда требуется быстро выполнить простые действия. При возникновении прерывания в процессорах MIPS автоматического сохранения контекста не происходит, запоминается лишь адрес возврата и запрещаются все прерывания (при выходе из обработчика восстанавливается прежнее состояние флага разрешения прерываний). При этом специально для начальных действий при входе в обработку прерывания соглашениями выделены регистры процессора k0 и k1 (26 и 27), которые никогда не используют компиляторы С/С++ и их значения можно не сохранять перед использованием. Например, обработчик прерывания, который должен просто изменять значение флажка в переменной с 0 на 1 и наоборот, может выглядеть следующим образом:

|  |
| --- |
| simple\_handler:  .set push  .set noreorder  .set noat  la k0, flag  lw k1, (k0)  xori k1, k1, 1  andi k1, k1, 1  sw k1, (k0)  eret  nop  .set pop |

В данном случае все действия легко реализуются с использованием всего двух регистров k0 и k1.

Если двух регистров недостаточно, но необходимо сохранить высокую скорость обработки прерывания, этот код можно слегка усовершенствовать:

|  |
| --- |
| simple\_handler:  .set push  .set noreorder  .set noat  addiu sp, sp, -4\*3 /\* для сохранения 3 регистров \*/  sw t0, 0(sp)  sw t1, 4(sp)  sw t2, 8(sp)  <действия с использованием регистров k0, k1, t0-t2>  lw t0, 0(sp)  lw t1, 4(sp)  lw t2, 8(sp)  addiu sp, sp, 4\*3  eret  nop  .set pop |

Такой обработчик «привязывается» непосредственно к одному из шести процессорных входов INT (например, INT3) функцией

|  |
| --- |
| kmk\_set\_int\_vector(3, &simple\_handler); |

Если на одну линию INT будут отражены два или несколько из 48 источников, то в таком простом коде затруднительно анализировать от какого именно, поэтому рекомендуется на один вход INT отображать один источник, требующий быстрой и простой обработки.

Для более сложной обработки прерывания функцией, написанной на языке Си, ППП МК предлагает второй вариант. Механизм предусматривает сохранение всех регистров процессора и вызов обработчика прерывания для конкретного источника.

В рамках приложения необходимо создать файл с программой на языке ассемблера, например, **my\_irqs.S** (или включить его код в уже существующий ассемблерный файл), следующего вида:

|  |
| --- |
| #include "regdef.h"  #include "kmk\_uke.h"  #include "kmk\_irqs.h"  .global my\_int2  my\_int2:  c\_handle\_irq\_saving\_regs(2) |

Здесь my\_int2() - функция, с помощью которой «сложный» обработчик будет привязан к одному из входов INT, в данном случае ко второму. Макрос c\_handle\_irq\_saving\_regs(INT\_n) предопределен в файле **kmk\_irqs.h**. В результате компиляции в коде программы будет создана функция my\_int2(), которая сохранит все регистры, вызовет установленный в прикладной программе обработчик конкретного источника прерывания, по завершении его работы восстановит регистры и завершит обработку прерывания.

Для привязывания обработчика handler

|  |
| --- |
| void handler(char param); |

к запросу, например, от интерфейса SPI1, в прикладной программе нужно выполнить следующие действия:

|  |
| --- |
| void kmk\_init\_int\_line(IRQ\_SPI1, 2, 8);  . . .  kmk\_set\_int\_vector(2, &my\_int2);  int\_connect(IRQ\_SPI1, &handler, 1); |

При этом handler в качестве аргумента получает параметр param (0..255), обычно обозначающий номер устройства из ряда одинаковых.

Для управления конкретными линиями прерываний в файле **kmk\_common.h** предусмотрено два макроса:

INTC\_irq\_disable(irq\_num, old)

Макрос осуществляет запрет прерывания на линии irq\_num, при этом

- остальные прерывания (иные irq) могут продолжать происходить и обрабатываться;

- в переменной old (должна иметь тип 32 битного целого) возвращается 0 если запрещаемое прерывание уже было запрещено, либо ненулевое число, если было разрешено (значение данной переменной удобно использовать, чтобы определить, нужно ли восстанавливать (размаскировать) прерывание после окончания критической секции;

- имплементация макроса выполнена безопасно для использования в среде с разрешенными прерываниями и в обработчиках прерываний (для MIPS используется специальный набор команд LL/SC).

INTC\_irq\_enable(irq\_num)

Макрос разрешает (размаскирует) прерывание на линии irq\_num. Также, как и INTC\_irq\_disable, безопасен для работы в среде с разрешенными прерываниями и в обработчиках прерываний.

Пример использования:

|  |
| --- |
| unsigned long ulRestoreState = 0;  . . .  INTC\_irq\_disable(IRQ\_UART4, ulRestoreState);  /\* critical section \*/  if (ulRestoreState) INTC\_irq\_enable(IRQ\_UART4); |

## 6.7 Примеры программ

В качестве примера будет рассмотрена простая программа, выполняющая индикацию двумя светодиодами, один из которых подключен непосредственно к GPIO микроконтроллера, а второй – через распространенный I2C расширитель GPIO – PCA9555A (Рисунок 4).

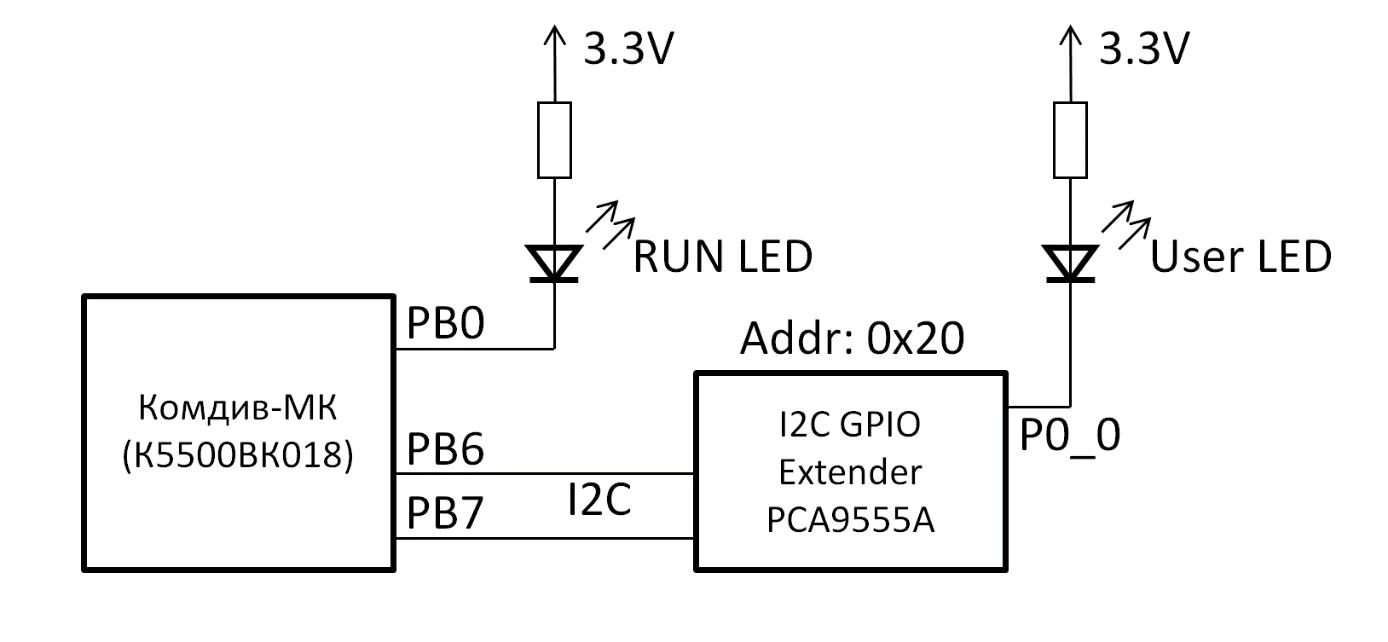


Рис. 4. Упрощенная схема демонстрационного устройства

Демонстрационная программа должна обеспечить включение индикатора RUN LED и мигание индикатора User LED с частотой 0,5 герц (секунду выключен, секунду включен). При этом структура программы будет рассчитана на кооперативную многозадачность (что упростит ее использование в качестве шаблона для более сложных задач).

Полный код проекта входит в комплект данной поставки.

Прикладная часть программы - Файл **app.c**. Данный код является переносимым и при необходимости может быть запущен на любом микроконтроллере, поддерживаемом ППП МК.

Функция работы с GPIO экстендером выполнена в виде конечного автомата с 6 состояниями (для детального описания манипуляций с регистрами экстендера – в том числе с его инициализацией – следует обратиться к документации на PCA9555A (см. (<https://datasheets.su/nxp/22823-PCA9555A.html> ).

В соответствии с рис. 4, оба светодиода зажигаются низким уровнем на соответствующей линии, при этом для светодиода, подключенного непосредственно к выводу микроконтроллера, этот нюанс учитывается в аппаратно-зависимом файле **main.c** (см. ниже), а для светодиода, подключенного через экстендер – в функции extender\_tick( ).

//

#include "app.h"

#include "../../../arch/arch.h"

#define I2C\_ADDRESS 0x20

// blinker period defined in microseconds

#define BLINK\_PERIOD 1000000

// MACRO to check timeouts

#define UL\_POSITIVE(ul) ((ul) < (((unsigned long)(-1))>>1))

// RUN LED switching function, released in hardware-specific part (main.c)

//

void led\_run(unsigned char bOn);

// Some variables for blinking FSM

static unsigned char ucBlinkerFSMState = 0;

static unsigned long ulBlinkerTO = 0;

static inline void extender\_tick(unsigned long ulNow)

{

unsigned char pbI2CBuf[2];

void \*pI2C = i2c\_get(0);

switch(ucBlinkerFSMState) {

case 0:

case 3:

// Waiting for timeout to switch User LED ON (state 0) of OFF (state 3)

if (UL\_POSITIVE(ulNow - ulBlinkerTO)) {

ulBlinkerTO += BLINK\_PERIOD;

ucBlinkerFSMState++;

} else break;

case 1:

case 4:

// trying to send a command to extender

// (we work with pin 0 of port 0, all the other pins assumes zeroed)

// Led is switching ON by LOW level at our board

pbI2CBuf[0] = 2; pbI2CBuf[1] = (ucBlinkerFSMState == 1)? 0 : 1;

if (i2c\_master\_send(pI2C, (I2C\_ADDRESS << 1), pbI2CBuf, 2, 1)) break;

ucBlinkerFSMState++;

case 2:

case 5:

if (!i2c\_io\_ready(pI2C)) break;

ucBlinkerFSMState++;

if (ucBlinkerFSMState > 5) ucBlinkerFSMState = 0;

}

}

// Application-level Initialization.

// - switch RUN LED on

// - configure extender to OUTPUT/HIGH for channel P0\_0

//

void app\_initialize(void)

{

unsigned char pbI2CBuf[3];

void \*pI2C = i2c\_get(0);

led\_run(1);

// set output levels to HIGH on port 0 pin 0

// (see PCA9555A datasheet for details)

pbI2CBuf[0] = 2; pbI2CBuf[1] = 1;

if (!i2c\_master\_send(pI2C, (I2C\_ADDRESS << 1), pbI2CBuf, 2, 1)) {

while (!(i2c\_io\_ready(pI2C))) { \_\_asm("nop"); }

}

// set inversion to OFF on both extender ports

// (see PCA9555A datasheet for details)

pbI2CBuf[0] = 4; pbI2CBuf[1] = 0; pbI2CBuf[2] = 0;

if (!i2c\_master\_send(pI2C, (I2C\_ADDRESS << 1), pbI2CBuf, 3, 1)) {

while (!(i2c\_io\_ready(pI2C))) { \_\_asm("nop"); }

}

// configure pin 0 on port 0 as OUTPUT

pbI2CBuf[0] = 6; pbI2CBuf[1] = 0xFE;

if (!i2c\_master\_send(pI2C, (I2C\_ADDRESS << 1), pbI2CBuf, 2, 1)) {

while (!(i2c\_io\_ready(pI2C))) { \_\_asm("nop"); }

}

}

// Application internal cycle.

// Functional component gets the time quant

//

void app\_cycle\_tick(void)

{

unsigned long ulNow = \_\_micros();

extender\_tick(ulNow);

}

Файл **main.c** содержит код, специфический для микроконтроллера Комдив-МК и для конкретной схемы (рис. 4).

//

// Configuring KMK board and start application

//

#include "app/app.h"

#include "arch/arch.h"

#include "arch/kmk\_common.h"

//////////

void led\_run(unsigned char bOn)

{

GPIO\_WritePin(GPIOB\_BASE, 0, (bOn)? 0 : 1);

}

//////////

static inline int init\_board(void)

{

GPIO\_SetPinMode(GPIOB\_BASE, 0, GPIO\_MODE\_OUTPUT); // RUN LED

// I2C 0

GPIO\_SetPinMode(GPIOB\_BASE, 6, GPIO\_MODE\_ALT1);

GPIO\_SetPinMode(GPIOB\_BASE, 7, GPIO\_MODE\_ALT1);

kmk\_i2c\_init(0, KMK\_ADDR(I2C0\_BASE), 400000);

return 1;

}

//////////

int main(void)

{

init\_board();

app\_initialize();

while (1) {

app\_cycle\_tick();

}

// unreacheable

return 1;

}

# 7 Порядок работы

## 7.1 Установка программы на инструментальную ЭВМ

7.1.1 В качестве инструментальной ЭВМ применяется компьютер типа IBM PC с установленной ОС Linux.

7.1.2 Установка программы

Для установки программы на инструментальную ЭВМ необходимо скопировать исходный текст программы из документа ЮКСУ.91264-01 12 01 на инструментальную ЭВМ и распаковать его в рабочем каталоге (например, **prj**).

Затем, в файле **prj/arch/kmk/Makefile\_kmk\_common.mk** в разделе TOOLS необходимо заменить на актуальные значения переменных, указывающих путь к кросс-компилятору и его название:

|  |
| --- |
| GNU\_TOOLS\_PATH ?= /usr/mips-linux-gnu/bin/  CROSS\_COMPILE ?= mips-linux-gnu- |

7.1.3 Установка кросс-компилятора и утилит

Для Комдив-МК К5500ВК018 кросс-компилятор, утилиты и необходимые для установки библиотеки здесь:

<https://gmplib.org/download/gmp/gmp-6.2.1.tar.xz>

<https://www.mpfr.org/mpfr-current/mpfr-4.1.0.tar.xz>

<https://www.multiprecision.org/downloads/mpc-1.2.0.tar.gz>

<https://ftp.gnu.org/gnu/binutils/binutils-2.37.tar.bz2>

<ftp://ftp.gnu.org/gnu/gcc/gcc-11.3.0/gcc-11.3.0.tar.gz>

Можно воспользоваться подготовленным пакетом **GNUTK\_MIPS.tgz** из документа ЮКСУ.91264-01 12 02, в котором имеется инструкция (**readme**) и скрипт установки **INSTALL\_GCC.sh**.

Установка этого кросс-компилятора проверена на следующих версиях ОС Linux:

Fedora release 16 (Verne)

Fedora release 35 (Thirty Five)

CentOS Linux release 7.9.2009 (Core)

Ubuntu 22.10 Kinetic

7.1.4 Установка программы flashrom

## 7.2 Сборка программы

После установки компилятора и исходного кода программы необходимо перейти в каталог **prj/apps/<name>/…**, где находится **Makefile** и выполнить в командной строке **make**. Программа будет собрана в этом же каталоге, где появятся файлы **\*.bin** и **\*.elf**. Любой из них может быть использован для загрузки программы на устройство – в модуль с микросхемой К5500ВК018.

Рекомендуется перед переходом к другому проекту выполнять команду ‘**make clean**’.

## 7.3 Загрузка программы

Собранная программа должна быть загружена в соответствующее устройство – плату с микросхемой К5500ВК018 с помощью программы **flashrom**.

Лист регистрации изменений

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Номера листов (страниц) | | | |  |  |  |  |  |
| Изм. | изме-ненных | заме-нен-ных | новых | анну-лиро-ванных | Всего листов (стра-ниц) в доку-менте | Номер документа | Входящий номер сопрово-  дительного документа и дата | Под-пись | Дата |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |