

Оглавление

- 1 Прерывания 2
- 2 Протокол 3
 - 2.1 IdibusDefs 3
 - 2.2 IdiBusSerial 3
 - 2.3 IdiBusSerialLine..... 4
 - 2.4 IdiBusMes 4
 - 2.5 IdiBusDateTime..... 4
 - 2.6 IdiBusSlave 5
- 3 IDIBUS_4DC..... 5

1 Прерывания

В среде WinAVR используется предустановленная таблица векторов прерываний, содержащая адреса соответствующих подпрограмм обслуживания с заранее определенными именами. Для каждой такой подпрограммы в этом библиотечном файле определен макрос: `ISR()`. Этот макрос регистрирует и помечает некоторую функцию как обработчик прерывания ([ссылка](#)).

Для каждого прерывания может быть объявлен **только один** обработчик прерывания с привязкой к соответствующей области памяти, в которую будет осуществлен переход во время исполнения прерывания. Для разрешения проблемы статической привязки векторов прерывания к библиотекам были написаны классы группы **IdiBusComInterrupts**. Принцип работы заключается в том, что все вектора прерываний объявлены в одном месте и библиотека, которая хочет использовать вектор прописывает указатель на свой обработчик прерывания в специальную переменную.

- **IdiBusCallbackInterface** – интерфейс, который наследуют библиотеки, которые хотят вызвать прерывание. Наследуя данный интерфейс, библиотека получает список методов, переопределив которые, можно сделать свой обработчик прерывания. В перспективе нужно разделить прерывания на разные интерфейсы. Пример:

```
class IdiBusSerial : public IdiBusCallbackInterface {
    IdiBusSerial () { InterruptListenersList.USART0_CallbackInstance = this; }

    virtual void USART_RxInterruptFunc(void) override
    {
        // Этот метод будет вызван в прерывании
    }
}
```

- **IdiBusInterruptsList.h -> InterruptListenersList** – список указателей на классы, унаследовавшие Callback интерфейс, который используется при вызове прерывания в ISR.
-
- **IdiBusInterruptsDisable.h** – если возникла исключительная ситуация, когда нужно использовать библиотеку, которая не работает по общим правилам, и объявляет вектор внутри себя (например, Arduino таймер), то можно отключить компиляцию определенных участков кода через макросы (например, `IDIBUS_USART0_ISR_DISABLE`).

2 Протокол

2.1 IdibusDefs

Общие для мастера и слэйвов дефайны таймаутов, кодов ошибок, позиций байт в сообщении и т.д.

2.2 IdiBusSerial

Базовый класс для передачи данных на линию по протоколу. Данный класс отвечает за соблюдение таймингов и настройку аппаратной части контроллера. Для пользователя предоставляются следующие публичные методы:

- `Init(IDIBUS_SERIAL_INIT_TYPEDEF *InitT)` – инициализация линии, IO порты задаются в ардуино стиле (1, A0 и т.п.)
- `Init(IDIBUS_SERIAL_ALTERNATIVE_INIT_TYPEDEF *InitT)` – инициализация линии, IO порты задаются в стандартном стиле (PORTA, PORTB и т.д.)

* Остальные параметры Init берутся из enumeration в IdiBusSerial.h

```
enum IDIBUS_SERIAL_USARTN_NUMBER {  
    IDIBUS_SERIAL_USART0,  
    IDIBUS_SERIAL_USART1,  
    IDIBUS_SERIAL_USART2,  
    IDIBUS_SERIAL_USART3  
};  
enum IDIBUS_SERIAL_TIMER_NUMBER {  
    IDIBUS_TIMER_1 = 1,  
    IDIBUS_TIMER_3 = 3,  
    IDIBUS_TIMER_4,  
    IDIBUS_TIMER_5  
};  
enum IDIBUS_SERIAL_USARTN_CONFIG {  
    IDIBUS_SERIAL_8N1,  
    IDIBUS_SERIAL_8N2,  
    IDIBUS_SERIAL_8E1,  
    IDIBUS_SERIAL_8E2,  
    IDIBUS_SERIAL_8O1,  
    IDIBUS_SERIAL_8O2  
};
```

- `SetBaudrate(enum IDIBUS_SERIAL_BAUDRATE Baudrate)` – задать скорость вручную из стандартного списка

```
enum IDIBUS_SERIAL_BAUDRATE {  
    IDIBUS_BAUDRATE_2400 = IDIBUS_BAUDRATE_DSW_CODE_2400B,  
    IDIBUS_BAUDRATE_9600 = IDIBUS_BAUDRATE_DSW_CODE_9600B,  
    IDIBUS_BAUDRATE_19200 = IDIBUS_BAUDRATE_DSW_CODE_19200B,  
    IDIBUS_BAUDRATE_115200 = IDIBUS_BAUDRATE_DSW_CODE_115200B,  
    IDIBUS_BAUDRATE_250K = IDIBUS_BAUDRATE_DSW_CODE_250K,  
    IDIBUS_BAUDRATE_500K = IDIBUS_BAUDRATE_DSW_CODE_500K,  
    IDIBUS_BAUDRATE_1M = IDIBUS_BAUDRATE_DSW_CODE_1M,  
    IDIBUS_BAUDRATE_10M = IDIBUS_BAUDRATE_DSW_CODE_10M  
};
```

- `UpdateBaudrateFromDipSwitch(void)` – установить скорость в соответствии с DIP SWITCH
- `Start (void)` – активировать линию
- `Stop (void)` – остановить линию (отпустить линии TX и RX)
- `SendRequestAsync(uint8_t *_TxBuf_, uint16_t _TxBufLength_)` – сеанс передачи типа запрос-ответ в неблокирующем режиме (асинхронно)

- `SendRequestSync (uint8_t *_TxBuf_, uint16_t _TxBufLength_)` – то же самое, только синхронно (ожидание окончания приема-передачи)
- `WriteSync (uint8_t *_TxBuf_, uint16_t _TxBufLength_);`
`WriteAsync(uint8_t *_TxBuf_, uint16_t _TxBufLength_)` – сеанс передачи запрос без ответа
- `IsTransferComplete(void)` – проверка, что сеанс приема-передачи/передачи закончен для асинхронного режима работы
- `GetRxBufCounter(void)` – ...
- `GetErrorStatus(void)` – получить статус сеанса приемо-передачи/передачи (если SumVar не ноль, то ошибка)

```
typedef union {
    uint8_t SumVar;
    struct {
        uint8_t TimeoutERR : 1;
        uint8_t TxOverflow : 1;
        uint8_t RxOverflow : 1;
        uint8_t LLComERR : 1;
        uint8_t TxZerolength : 1;
        uint8_t Reserved : 3;
    } BF;
} IDIBUS_SERIAL_ERROR_STATUS_TYPE;
```

- `GetBaudrate(void)` – получить текущее значение скорости в `uint32_t` (рудимент)

2.3 IdiBusSerialLine

Следующий уровень абстракции после `IdiBusSerial`. Наследует `IdiBusSerial`. Является основным классом для работы с линией. Умеет отправлять `MMES` и `MMESG`. Разбирает ошибки формата при приеме и передачи. Не разбирает, что передано в поле `data` и формат этого поля.

```
uint8_t sendMMES (IdiBusMMES *MMES);           // return ErrorCode but not look into [data]
uint8_t sendMMESG (IdiBusMMESG *MMESG);
```

2.4 IdiBusMes

Шаблоны сообщений `MMES` и `MMESG`, которые затем передаются в объект класса `IdiBusSerialLine`. Вызывается один из двух конструкторов:

```
IdiBusMMES (uint8_t S1Addr, uint8_t *TX_DataP, uint16_t TX_DataLength);
```

```
IdiBusMMESG (uint8_t S1Addr, uint8_t *TX_DataP, uint16_t TX_DataLength); – если нужно что-то
```

передать в поле `Data`

```
IdiBusMMES (uint8_t S1Addr);
```

```
IdiBusMMESG (uint8_t S1Addr); - если в поле Data ничего передавать не нужно (например, с_Dummy)
```

Далее в зависимости от типа сообщения конфигурируются номер девайса, канала и код функции.

Больше ничего делать не нужно – `IdiBusSerialLine` и `IdiBusMes` сами проверят данные на корректность и распахнут все в правильные места.

2.5 IdiBusDateTime

Шаблоны для объектов времени и даты

2.6 IdiBusSlave

Последний уровень абстракции. Предоставляет инструмент для разработчиков для быстрой интеграции в мастера. Его задача взять на себя работу со стандартными командами и стандартные обработки ошибок, которые выполняются в каждом сообщении .

Для того, чтобы сделать типизацию стандартных канальных команд по данной концепции, слэйвы, которые наследуют данный класс должны формировать девайсы и каналы, наследуя классы **IdiBusDevice** и **IdiBusChannel**.

Виртуальные методы:

```
virtual IdiBusDevice *getDevice(uint8_t Num) { return NULL; }  
virtual IdiBusChannel *getChannel(uint8_t Num) { return NULL; }
```

должны быть переопределены в библиотеки соответствующего слэйва для того, чтобы методы родительского класса могли обращаться к конкретным каналам. В реализации данных методов со стороны слэйва должна стоять проверка выхода за диапазон (см. класс IDIBUS_4DC)

Публичные методы:

```
uint8_t c_Init (void);  
uint8_t c_CheckModuleLongOp(void); - ... стандартные команды
```

```
// Standard communication API
```

```
uint8_t SendRequestMMES(IdiBusMMES *MMES);
```

```
uint8_t SendRequestMMESG(IdiBusMMESG *MMESG); - работают над sendMMES и sendMMESG, считают скользящее среднее по ошибкам CRC и определяют ошибки модуля конкретного слэйва. Возвращают код ошибки по системе IdiBus (IDIER_NOPE и т.д.)
```

Слэйв может быть инициализирован несколькими образами:

1 – через конструктор `IdiBusSlave(IdiBusSerialLine *SlaveLine, uint8_t SlaveAddress);`

2 – через метод `Init(IdiBusSerialLine *SlaveLine, uint8_t SlaveAddress);` - если объявляется большой массив объектов через цикл for

3 IDIBUS_4DC

Два девайса:

0 – управление каналами выходами

1 – измерение входного напряжения модуля

Девайс 1:

`getInputVoltage(void);` - Отправить запрос на модуль для получения данных . вернет IDIBUS ERROR

`getChData(void);` - получить копию данных о входном напряжении

```
uint16_t InpVoltage;  
union {  
    uint8_t Sum;  
    struct {  
        uint8_t Undervoltage : 1;  
        uint8_t Overvoltage : 1;  
        uint8_t Reserved : 6;  
    } BF;  
    } STATUS;
```

Девайс 0:

Если обращение идет к конкретному каналу, то возвращает IDIBUS ERROR, если к группе каналов, то void. Коды ошибок хранятся в структуре каждого канала (наследуется от IdiBusChannel), после группового запроса нужно в цикле опросить се каналы на предмет ошибок, поскольку ошибки могут присутствовать только на части запрошенных каналов.

`getChStatus(uint8_t ChNum);`

`getAllChStatus(void);` - получить статус устройства + TargetVoltage + CurrentLimit

```
uint16_t TargetVoltage;
uint16_t CurrentLimit;
struct {
    uint8_t Enable;
    uint8_t PowerGood;
    union {
        uint8_t Sum;
        struct {
            uint8_t PeripheryError : 1;
            uint8_t OvercurrentError : 1;
            uint8_t OvervoltageError : 1;
            uint8_t OffStateCurrentLeakage : 1;
            uint8_t MemoryError : 1;
            uint8_t InpVoltageError : 1;
            uint8_t Reserved : 2;
        } BF;
    } ERRORS;
} STATUS;
```

`getChAllData(uint8_t ChNum);`

`getAllChAllData(void);` - получить все, что можно получить с канала

```
int32_t Voltage;
int32_t Current;
uint16_t TargetVoltage;
uint16_t CurrentLimit;
struct {
    uint8_t Enable;
    uint8_t PowerGood;
    union {
        uint8_t Sum;
        struct {
            uint8_t PeripheryError : 1;
            uint8_t OvercurrentError : 1;
            uint8_t OvervoltageError : 1;
            uint8_t OffStateCurrentLeakage : 1;
            uint8_t MemoryError : 1;
            uint8_t InpVoltageError : 1;
            uint8_t Reserved : 2;
        } BF;
    } ERRORS;
} STATUS;
```

Остальное очевидно

`getChOutputCurrent`

`setChVoltage`

...