**Системный каталог drv**

**1.** **Назначение**

Доступ компонентов ПО к аппаратным средствам.

**2. Область применения**

Для использования в проектах двухканальных безопасных приборов.

**3. Термины и определения**

МК – микроконтроллер.

ЗС – защитное состояние.

БС – безопасное состояние.

**4.** **Состав**

В состав каталога входят две группы модулей и файлов:

1. Модули и файлы, присутствующие, как правило, во всех проектах;
2. Модули и файлы, специфичные для конкретного проекта.

Здесь описывается первая группа. В ее состав входят:

1. iodrv.h – стандартные интерфейсы драйверов;
2. mainRegisters.h – настройка основных портов и регистров для семейства МК dsPIC33E;
3. IOports.h – макросы для работы с портами ввода-вывода;
4. turnOffMC.h – отключение периферии МК;
5. timeSynchronizationPorts.h – порты для временной синхронизации МК;
6. ConfigMcPorts.h – порты ввода/вывода определения типа канала: мастер или слейв;
7. ConfigurationMC.h – конфигурация МК (настройка fuse регистров);
8. checkRegistersDrv – проверка основных регистров, используемых при работе;
9. СheckExitPSdriver — драйвер проверки условий выхода из ЗС;
10. checkCauseReset - определение причин сброса МК;
11. infinityWhile – функция бесконечного пустого цикла;
12. SignalFilter.s — функция КИХ фильтрации с использованием DSP;
13. TechnologPorts.h — определение портов средств отладки и регулировки;
14. AdcDriver, Adc1Macros.h, AdcPorts.h — драйвер АЦП;
15. EcanDriver, ECAN1\_PinRemap.h, ECAN2\_PinRemap.h, Ecan\_DMA.h, Ecan\_regsPIC33.h – драйвер CAN;
16. ManageCanDriver – управление драйвером CAN;
17. readInstructionCode - чтение кода команды из ПЗУ.

**5. Общие сведения**

С целью разделения логических функций ПО от аппаратуры используются следующие уровни:

1. уровень драйверов − обеспечивает программный интерфейс доступа к аппаратуре;
2. логический уровень − реализует логику принятия решений и при помощи уровня драйверов формирует команды (сигналы) для выдачи в аппаратуру.

Данное разделение позволяет логическому уровню быть пригодным к автономному тестированию и проще переносить этот уровень на другие аппаратные платформы.

Драйвера с логическим уровнем могут связываться статически и динамически. Статическое связывание осуществляется на этапе сборки проекта. Компонент логического уровня использует универсальные имена функций. Модуль драйвера содержит реализацию этих функций. Компоновщик находит нужный объектный файл и связывает компонент логического уровня с драйвером. Таким образом, подмена драйверов выполняется при помощи удаления или добавления нужной реализации драйвера в проект. Динамическое связывание осуществляется через структуру, которая содержит поля, хранящие адреса вызовов функций. Компонент ПО, получив указатель на эту структуру, обращается к функциям драйвера через поля этой структуры. Таким образом, драйвер может быть подменен в процессе выполнения кода. Использование динамического связывания драйверов позволяет предопределять драйвера без изменения компонента логического уровня, не вызывая при этом конфликта имен. А также проводить автономное тестирование компонента логического уровня без пересборки тестового проекта, подменяя драйвер различными объектами, имитирующими его поведение с разными тестовыми наборами данных.

6. Модуль iodrv

Модуль iodrv предназначен для реализации интерфейса между компонентами логического уровня и модулями уровня драйверов, поддерживающих динамическое связывание. Модуль iodrv определяет типы указателей на функции драйверов и структуры данных (таблицы функций), которые содержат адреса вызовов этих функций.

Модуль iodrv классифицирует драйвера по нескольким критериям в зависимости от:

1. прототипов функций, содержащихся в таблице функций;
2. набора функций в таблице функций.

В зависимости от прототипов функции драйвера отличаются набором параметров и типом этих параметров.

В зависимости от набора функций драйвера бывают:

1. драйвера ввода;
2. драйвера вывода;
3. драйвера ввода/вывода.

Все функции драйвера должны содержать параметр *self*, который является указателем на тип *void*. Через данный указатель в функцию передается указатель на объект драйвера (объект, реализующий структуру данных, специфическую для каждого драйвера и определенную разработчиком драйвера).

Для того, чтобы компонент логического модуля мог использовать динамическое связывание, при разработке компонента должен быть определен тип драйвера, который будет использован при работе компонента. В интерфейсах компонента должен присутствовать параметр, принимающий указатель на таблицу функций определенного типа драйвера и параметр, принимающий указатель на объект драйвера. Компонент должен обращаться к функциям драйвера через указанный параметр, обращаясь к полям структуры таблицы функций и передавая им указатель на объект драйвера.

Должен быть разработан модуль драйвера, который содержит реализации всех функций, перечисленных в таблице функций данного типа и функцию-конструктор («имя драйвера\_ctor»), которая принимает настройки драйвера, а возвращает указатель на объект драйвера типа *void \**. В дальнейшем это значение будет передаваться в качестве параметра *self* функциям этого драйвера.

Связывание имеет следующий сценарий:

1. объявляется структура, которая содержит таблицу функций выбранного типа драйвера;
2. поля структуры инициализируются адресами соответствующих функций драйвера;
3. объявляется переменная-указатель на void;
4. вызывается конструктор драйвера, а результат, возвращаемый конструктором, сохраняется в предварительно объявленную переменную (указатель на *void*);
5. вызывается функция-конструктор компонента логического уровня, которому передаются указатели на таблицу функций драйвера и объект драйвера;
6. компонент логического уровня используется по назначению.

7 **Файл MainRegisters.h**

Этот файл определяет сущности, имеющие существенное значение для организации работы МК. Определение сущностей через макросы позволяет легко переносить проект на другой тип МК или использовать другой компилятор. Он содержит:

1. константы частоты работы МК;
2. макрос настройки частоты работы МК;
3. макросы, определяющие функции обработки прерываний;
4. макросы работы с прерываниями: проверка, запрет, разрешение, сброс флага прерывания;
5. определение главного таймера и макросов для работы с ним
6. макросы работы с таймером, обеспечивающего обмен по CAN для синхронизации данных;
7. макросы работы со сторожевым таймером;
8. макрос настройки прерываний МК;
9. макрос запрета прерываний МК.

Для использования этого файла его следует при необходимости откорректировать и включить в проект.

8 **Файл** **IOports.h**

Этот файл создан для облегчения настройки портов дискретного ввода/вывода. Он содержит макросы настройки портов на ввод и вывод.

9 **Модуль** turnOffMC

В этом файле реализована функция выключения работы периферии МК. Работа этой функции происходит по следующему сценарию:

1. запрет маскируемых прерываний командой disi;
2. запрет работы сторожевого таймера;
3. остановка работы главного таймера;
4. запрет всех прерываний;
5. отключение периферийных модулей, используемых в проекте;
6. перевод всех портов ввода/вывода в третье состояние;
7. отключение периферийных модулей МК от портов ввода/вывода.

Для использования этого файла его следует при необходимости откорректировать и включить в проект.

**10 Файл timeSynchronizationPorts.h**

Файле содержит макросы инициализации и работы с портами ввода/вывода временной синхронизации. Эти макросы используется модулями:

1. timeBeginSynchronization — для синхронизации каналов по включению питания;
2. InterruptHandlerTimer и InterruptHandlerCN — для синхронизации каналов во время работы.

**11 Файл ConfigMcPorts.h**

Файле содержит макросы инициализации и работы с портами ввода/вывода определения типа канала: мастер или слейв. Эти макросы используется модулем ConfigMC.

**12 Файл ConfigurationMC.h**

Файле содержит директивы для компилятора, которые определяют запись значений во FUSE-регистры. Содержимое FUSE-регистров задает:

1. режим защиты памяти;
2. определяет тактовый генератор;
3. параметры работы сторожевого таймера;
4. разрешение или запрет контроля микроконтроллером его питания;
5. подключение альтернативных пинов для I2C;
6. параметры подключения программатора.

Для использования этого файла его следует при необходимости откорректировать и включить в проект. Включение в проект происходит директивой #include "ConfigurationMC.h" в файле проекта, обычно MaintInitial.c.

**13 Модуль checkRegistersDrv**

Модуль содержит функцию контроля исправности регистров модулей периферии. Контроль исправности заключается в проверке, что в битах регистров находятся определенные значения. Проверяются обычно регистры таких модулей периферии, как главный таймер, система прерываний, АЦП, CAN, DMA. Набор контролируемых модулей периферии может меняться от проекта к проекту. Вызывается на выполнение эта функция из модуля CheckRegisters (компонент ControlMC).

Для использования этого модуля его следует при необходимости откорректировать и включить в проект.

**14 Модуль СheckExitPSdriver**

Модуль состоит из трех файлов:

1. CheckExitPSdriver.h — заголовочный файл;
2. CheckExitPSdriver.c – реализация алгоритма;
3. CheckExitPSdriverPorts.h — определение портов, по которым проверяется наличие перемычки снятия ЗС.

Модуль проверяет условия выхода из ЗС. Условием выхода является наличие установленной перемычки снятия ЗС. Перемычка замыкает дискретный выход дискретным выходом. Контроль происходит по следующему алгоритму:

1. Инициализируются порты дискретного ввода и вывода;
2. На дискретный выход подается высокий уровень;
3. На дискретном входе контролируется высокий уровень;
4. На дискретный выход подается низкий уровень;
5. На дискретном входе контролируется низкий уровень.

Для использования этого модуля его следует при необходимости откорректировать и включить в проект. Файл CheckExitPSdriverPorts.h при необходимости откорректировать.

**15 Модуль checkCauseReset**

Модуль контролирует причины сброса МК. Следующий набор (по логическому «и») определяет нормальную работу МК:

1. сброс произошел по включению питания или по сигналу внешнего сброса (этот сигнал формирует внешний супервизор питания);
2. отсутствует программный флаг сброса МК;
3. отсутствует флаг сброса по сторожевому таймеру.

Во всех остальных случаях происходит переход в ЗС.

Для использования этого модуля его следует включить в проект.

**16 Модуль infinityWhile**

Модуль реализует бесконечный пустой цикл. На самом деле в бесконечном цикле повторяется команда «pwrsav #0», которая переводит процессор в спящий режим.

Для использования этого модуля его следует включить в проект.

**17 Файл SignalFilter.s**

В файле реализована функция КИХ фильтрации с использованием DSP. Функция реализована на ассемблере. Прототип функции для использования Си-модулями имеет такой вид:

void SignalFilter( int16\_t \*FilterCoeffStartAddress, unt16\_t \*FilterDelayPtr, uint16\_t FilterCoeffLen , uint16\_t SampleCount , int16\_t \*FilterDelayStartAddress, int16\_t \*FilterDelayEndAdress, int16\_t \*InputSamples, int16\_t PSV\_Addishift);

Параметры:

*FilterCoeffStartAddress* − стартовый адрес B-коэффициентов фильтра.

*FilterDelayPtr*  − указатель на ячейку, содержащую указатель текущей позиции в линии задержки.

*FilterCoeffLen*  − количество B-коэффициентов фильтра.

*SampleCount*  − количество отсчётов в массиве InputSamples.

*FilterDelayStartAddress*  − указаиель на первый отсчёт в линии задержки (начало кольцевого буфера).

*FilterDelayEndAdress* − указаиель на последний отсчёт в линии задержки (конец кольцевого буфера).

*InputSamples* − указатель на входные данные.

*PSV\_Addishift* − биты 7...0 содержат значение регистра PSV, биты 15...8 содержат число сдвигов аккумулятора для приведения результата к формату int16\_t.

Для использования этого модуля его следует включить в проект.

**18 Модуль TechnologPorts.h**

Файле содержит определение портов средств отладки и регулировки. Является проектно зависимым.

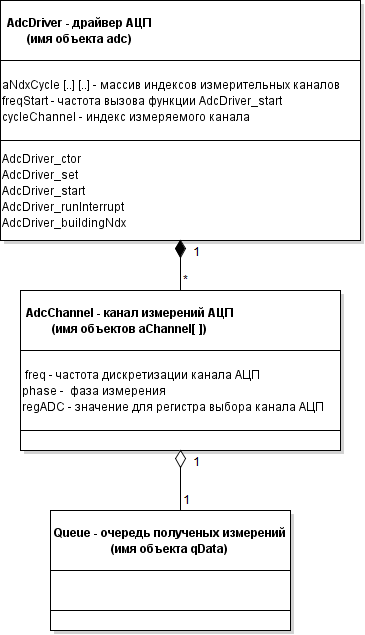
Для использования этого модуля его следует откорректировать и включить в проект.

**19 Драйвер АЦП**

Драйвер АЦП предназначен для инициализации встроенного в процессор АЦП и работы с ним. Драйвер АЦП состоит из следующих файлов:

1. AdcDriver .h — интерфейсы драйвера;
2. AdcDriver .c — реализация драйвера;
3. Adc1Macros.h — макросы работы с АЦП;
4. AdcPorts.h — макросы работы с портами дискретных входов использующихся, как входы АЦП.

Статическая модель драйвера АЦП представлена на рисунке ниже.



Статическая модель драйвера АЦП

Драйвер АЦП представляет из себя набор измерительных каналов (AdcChannel), которые последовательно измеряются по каналу АЦП CH0. Управляет работой драйвер АЦП AdcDriver. Работа драйвера АЦП происходит следующим образом.

Перед началом работы происходит настройка драйвера АЦП.

По вызову функции *AdcDriver\_ctor()* проводится инициализация периферийного модуля МК АЦП. При инициализации используются макрос *ADC\_INIC* для инициализации АЦП и макрос *ADC\_INIC\_PIN* для инициализации дискретных входов МК — настройка их, как входов АЦП.

По вызовам функции AdcDriver\_setChannel() происходит:

1. Инициализация массива структур измерительных каналов adc.aChannel[].
2. Инициализация массива aNdxCycle[][] (реализовано в функции *AdcDriver\_buildingNdx()* ).

Массив aNdxCycle задает очередность (циклограмму) работы измерительных каналов, представляет собой двухмерный массив, содержащий индексы измерительных каналов (см. Таблицу ниже). Строка массива содержит индексы измерительных каналов, которые соответствуют одному вызову функции *AdcDriver\_start()*. По вызову *AdcDriver\_start()* запускается измерения канала с индексом Индекс1 (индекс канала указывает на измерительный канал в массиве *adc.aChannel[]*). По завершению измерения вызывается функция *AdcDriver\_runInterrupt()*, которая сохраняет результат измерения в очереди *adc.aChannel[ Индекс1].qData*. Переполнение очереди приводит к переходу прибора в ЗС. Затем запускается измерение канала с индексом Индекс2 и т. д. до конца строки. Если значение индекса равно -1, то измерения по индексам этой строки прекращаются. По следующему вызову *AdcDriver\_start()* идет работа со следующей строкой. По завершению таблицы переходим к 1 строке. Длина таблицы находится в переменной *adc.freqStart*.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № строки |  |  |  |  |
| 1 | Индекс1 | Индекс2 | … | ИндексN |
| 2 | Индекс1 | Индекс2 | … | ИндексN |
|  |  |  |  |  |
| 16 | Индекс1 | Индекс2 | … | ИндексN |

Таблица, соответствующая массиву aNdxCycle.

Все вышеизложенное поясним на примере.

Прошла следующая настройка:

AdcDriver\_ctor( 8 ) - частота вызова *AdcDriver\_start()* 8 кГц.

AdcDriver\_setChannel( Канал1, , 8, 0); // Fd = 8 кГц, phase = 0

AdcDriver\_setChannel( Канал2, , 4, 0); // Fd = 8 кГц, phase = 0

AdcDriver\_setChannel( Канал3, , 2, 1); // Fd = 8 кГц, phase = 1

AdcDriver\_setChannel( Канал4, , 1, 3); // Fd = 8 кГц, phase = 3

AdcDriver\_setChannel( Канал5, , 1, 0); // Fd = 8 кГц, phase = 0

Fd – частота дискретизации измерительного канала, Канал соответствует индексу в массиве *adc.aChannel[]*). Параметр phase задает сдвиг по строкам таблицы.

Тогда таблица aNdxCycle будет иметь следующий вид:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № строки |  |  |  |  |
| 1 | Канал1 | Канал2 | Канал5 | -1 |
| 2 | Канал1 | Канал3 | -1 | -1 |
| 3 | Канал1 | Канал2 | -1 | -1 |
| 4 | Канал1 | Канал4 | -1 | -1 |
| 5 | Канал1 | Канал2 | -1 | -1 |
| 6 | Канал1 | Канал3 | -1 | -1 |
| 7 | Канал1 | Канал2 | -1 | -1 |
| 8 | Канал1 | -1 | -1 | -1 |
| 9 | -1 | -1 | -1 | -1 |
| ... | | | | |
| 16 | -1 | -1 | -1 | -1 |

Таблица, соответствующая массиву aNdxCycle, пример.

Параметр phase имеет смысл, если требуется уменьшить максимальное количество каналов в одной строке, что соответствует уменьшению пиковой нагрузке потока измерения АЦП.

Драйвер имеет следующее ограничения:

1. Частота вызова функции *AdcDriver\_start()* (в кГц) не должна превышать параметр ADC\_DRIVER\_N\_CYCLES\_F. При необходимости этот параметр корректируется.
2. Количество измерительных каналов, приходящихся на один вызов *AdcDriver\_start()* не должно превышать параметр ADC\_DRIVER\_NDX.При необходимости этот параметр корректируется.
3. Частота дискретизации измерительного канала не должна превышать частоту вызова функции *AdcDriver\_start()* и должна быть ей кратна.
4. Необходимо постоянно считывать результаты измерений функцией *AdcDriver\_getDataChannel()*, что бы не допустить переполнение очереди.
5. Необходимо обеспечить своевременное считывание измерений с регистров АЦП функцией *AdcDriver\_runInterrupt()*, обычно для этого организовывается обработка прерывания по завершению конвертации АЦП.

Для использования этого драйвера требуется:

1. Включить его в проект.
2. При необходимости откорректировать файлы Adc1Macros.h, AdcPorts.h.
3. Организовать настройку драйвера, обычно проводится в файле MainInitial.c.
4. Организовать вызов на выполнение функции *AdcDriver\_start().* Обычно проводится в файле InterruptHandlerCN.c с частотой, равной максимальной частоте измерительных каналов.
5. Организовать вызов функции *AdcDriver\_runInterrupt().* Обычно проводится в файле InterruptHandlerADC.c — обработка прерывания по завершению конвертации АЦП.
6. Организовать своевременное считывание результатов измерений функцией *AdcDriver\_getDataChannel().*

**20 Драйвер EcanDriver**

**20.1 Назначение**

Драйвер предназначен для передачи и приема сообщений по шине CAN.

**20.2 Состав**

Драйвер АЦП состоит из следующих файлов:

1. EcanDriver .h — интерфейсы драйвера;
2. EcanDriver .c — реализация драйвера;
3. ECAN1\_PinRemap.h — макросы подключения модуля ECAN1 к пинам МК;
4. ECAN2\_PinRemap.h — макросы подключения модуля ECAN2 к пинам МК;
5. Ecan\_DMA.h — макрос конфигурирования модуля DMAх для работы с модулем ECANх;
6. Ecan\_regsPIC33.h - структура регистров модуля ECAN.

**20.3 Описание работы**

Драйвер обеспечивает интерфейс типа ArrayIoDriver модуля iodrv и работу с двумя модулями ECAN МК семейства dsPIC33. Драйвер использует модули МК DMA: DMA0, DAM1 – для ECAN1 и DMA2, DAM3 – для ECAN2.

20.3.1 Модель использования  **драйвера**

Перед использованием драйвера должна быть вызвана функция ***Ecan\_ctor()***, которая проинициализирует драйвер для работы с указанным модулем ECAN.

Работа драйвера построена на опросе SFR-регистров МК. Опрос регистров осуществляется функцией ***Ecan\_run()***.

Для отправки сообщения при помощи функции ***Ecan\_isOutReady()*** необходимо убедиться что передатчик готов к передаче новых данных, задать данные для передачи и инициировать передачу функцией ***Ecan\_set()***. После начала передачи функция ***Ecan\_isOutReady()*** будет возвращать *false* до тех пор, пока передача не завершится успешно или с ошибкой, после функция вернет *true*. Успешность передачи можно проверить с помощью функции ***Ecan\_getError()***. При ошибке можно запустить повтор передачи функцией ***Ecan\_startTransmite().***

Для приема сообщения при помощи функции ***Ecan\_isInReady()*** необходимо убедиться что приемник принял сообщение и прочитать его при помощи функции ***Ecan\_get()***. После прочтения принятого сообщения признак его наличия будет сброшен и функция ***Ecan\_isInReady()*** будет возвращать *false* до тех пор, пока не примет новое сообщение. Если данные не будут вычитаны до приема следующего сообщения они будут утеряны.

**20.3.2 Функция Ecan**\_ctor

Функция ***Ecan\_ctor()*** инициализирует внутренние данные драйвера и настраивает указанные периферийные устройства МК.

Для настройки ECAN модуля функция принимает параметры: *numPort*, *txSid*, *rxSid,* *mode и lengthPacket*. Исходя из значения *numPort* выбирается переменная *eEcan1* или *eEcan2*, которая будет использоваться для доступа к SFR-регистрам соответствующего модуля CAN. Значения параметров *txSid* и *lengthPacket* записываются в соответствующие байты буферов aEcan1MsgBuf[] или aEcan1MsgBuf[]. Значение параметра *rxSid* записывается регистр RXF0SIDbits.SID.

Модуль CAN настраивается на частоту Fcan = 2 \* 59,904 MHz = 119,808 MHz. Tq (квант времени) настраивается равным 1 / F, где F = 119.808 MHz / 2 / 8 = 135/2 = 7,488 MHz. *SEG1PH*, *SEG2PH* и *PRSEG* настраиваются равными Tq, 2Tq и Tq соответственно. Ширина перехода синхронизации равна Tq. Номинальная скорость передачи равна 7936000 бит/с.

Для настройки дискретных портов вызывается макрос ***ECAN1\_PIN\_REMAP*** или ***ECAN2\_PIN\_REMAP***.

Принимаемые и выдаваемые данные модуля ECAN находятся в буферах ОЗУ. Для работы с ними модуль ECAN использует DMA. Для увязки буферов ОЗУ, модулей МК ECAN, DMA вызывается макрос ***ECAN\_CONFIG\_DMA***.

Функция возвращает указатель на структуру, в которой хранятся внутренние данные драйвера. В дальнейшем этот указатель должен передаться в качестве первого параметра для всех функций этого драйвера.

**20.3.3 Функция Ecan**\_get

Функция читает принятое сообщение из буфера *aEcan1MsgBuf* или *aEcan2MsgBuf*. Функция возвращает длину принятых данных.

**20.3.4 Функция Ecan**\_isInReady

Функция возвращает значение признака готовности принятых данных, который устанавливается функцией ***Ecan\_run()*** когда модуль ECAN принял сообщение с идентификатором, равным заданному при вызове конструктора в параметре *rxSid*. Когда сообщение принято, функция возвращает *true*. Если после инициализации модуля или последнего чтения новое сообщение принято не было, функция возвращает *false*.

**20.3.5 Функция Ecan**\_set

Функция копирует данные, принятые в параметре *array*, в буфер *aEcan1MsgBuf* или *aEcan2MsgBuf* и запускает передачу вызовом функции ***Ecan\_startTransmite***.

**20.3.6 Функция Ecan**\_**startTransmite**

Функция запускает передачу в модуле ECAN и переводит драйвер в состояние «передатчик в процессе передачи данных», которое обрабатывается при вызове ***Ecan\_run()***.

**20.3.7 Функция Ecan**\_isOutReady

Функция возвращает значение признака готовности модуля ECAN к передаче нового сообщения. Функция возвращает *true* когда функция *Ecan\_run()* находится в состоянии «передатчик свободен и готов к работе» или *false* когда модуль находится в состояниях «передатчику поступил запрос на передачу данных» и «передатчик в процессе передачи данных».

**20.3.7 Функция Ecan**\_run

Функция реализует прием сообщений и ожидание завершения передачи сообщений. Работа функции организована по принципу периодического опроса. Функция вызывается каждые 62,5 мкс, опрашивает состояние регистров модуля ECAN и на основании чего выполняет необходимые действия для реализации алгоритма обмена данными.

При каждом вызове функция проверяет бит *EWARN* (передатчик или приемник в состоянии ошибки) регистра *INTF* и, если он установлен, выполняется сброс модуля CAN. Сброс модуля осуществляется при помощи входа в режим конфигурации и возврата из него в рабочий режим.

Далее функция работает с приемником модуля ECAN. Сначала проверяется наличие ошибок приема, проверяются биты *RBOVIF*, *IVRIF* регистра *INTF*. Если хоть один бит установлен, считается что произошла ошибка, обнуляются биты *RBOVIF, RBIF, IVRIF, ERRIF* регистра *INTF* и бит *RXFUL* регистраRXFUL1. Если ни один из битов ошибок не установлен, но установлен бит RBIF, устанавливается признак готовности принятых данных. Состояние признака готовности принятых данных можно прочитать при помощи функции ***Ecan\_isInReady()***, а принятые данные можно прочитать из буфера приемника при помощи функции ***Ecan\_get()***. Если данные не будут прочитаны до приема следующего сообщения, произойдет ошибка переполнения буфера 0 и данные будут утеряны, поэтому после вызова функции ***Ecan\_run()*** обязательно должна следовать обработка принятых данных.

Далее функция работает с передатчиком модуля CAN. При работе с передатчиком функция может находится в трех состояниях: «Передатчик свободен и готов к работе», «Передатчик в процессе передачи данных», «Передатчик в состоянии ошибки».

В состояниях «Передатчик свободен и готов к работе» и «Передатчик в состоянии ошибки»опрос регистров модуля ECAN не производится.

В состоянии «Передатчик в процессе передачи данных» функция опрашивает бит *TXREQ0* регистра *TR01CON*. Если бит *TXREQ* информирует об окончании передачи то функция переходит в состояние «Передатчик свободен и готов к работе». Если при этом установлен бит *ERRIF* регистра *INTF*, то в переменной драйвера устанавливается признак отказа устройства: *error = ePS\_ErrorDevice*.

**20.4 Файлы ECAN1\_PinRemap.h и ECAN2\_PinRemap.h**

Файлы содержат макросы ECAN1\_PIN\_REMAP и ECAN2\_PIN\_REMAP выполняющих подключение модулей ECAN к пинам МК. Вынесение этой операции в отдельный файл сделано для облегчения:

1. настройки драйвера ECAN под схему прибора;
2. переноса драйвера на другой тип МК.

**20.5 Ecan\_DMA.h**

Файл содержит макрос ECAN\_CONFIG\_DMA выполняющий конфигурацию DMA под ECAN. Вынесение этой операции в отдельный файл сделано для облегчения переноса драйвера на другой тип МК.

**20.6 Файл Ecan\_regsPIC33.h**

В состав компилятора входит файл p33EP512GM710.h, который определяет структуру SFR-регистров МК, управляющих работой модулей ECAN1 и ECAN2, как структуру типа CAN. Все поля этой структуры определены как *unsigned int*, что не позволяет обращение к отдельным полям регистра через эту структуру напрямую. В этом же файле есть определение битовых полей каждого регистра и объявление внешних переменных, соответствующих каждому регистру.

Файл pEcan\_regsPIC33.h определяет тип *dsPIC33E\_CanRegs*, который объединяет в одной структуре все регистры модуля CAN и описание битовых полей для каждого из них. Это позволяет напрямую обращаться к любому полю любого из регистров через единую структуру. Данная возможность полезна при разработке драйверов модулей CAN. Драйвер может хранить указатель на область памяти SFR-регистров модуля CAN1 или CAN2 и через этот указатель управлять работой нужного модуля без дополнительных ветвлений.

**20.7 Применение**

Для использования этого драйвера требуется:

1. Включить его в проект.
2. При необходимости откорректировать файлы ECAN1\_PinRemap.h, ECAN2\_PinRemap.h, Ecan\_DMA.h.
3. Организовать настройку драйвера, обычно проводится в файле MainInitial.c.
4. Организовать вызов функции ***Ecan\_run()****.*

**21 Драйвер ManageCanDriver**

**21.1 Назначение**

Драйвер ManageCanDriver предназначен для использования компонентом InterChannel.

**21.2 Состав**

Драйвер ManageCanDriver состоит из одноименного модуля.

**21.3 Описание**

Драйвер обеспечивает интерфейс типа *ArrayIoDriver* модуля iodrv и управление одним драйвером *EcanDriver*. Драйвер *ManageCanDriver* повышает надежность обмена пакетов данных по шине CAN следующими способами:

1. Повтор неудачной передачи пакета данных;
2. Отсечение повторяющихся пакетов данных при приеме;
3. Запрет передачи пакета данных в конце главного цикла.

Эти способы реализуются в функции ***ManageCanDriver\_run()***.

Пакет данных состоит из данных (задаются при обращению к драйверу) и номера пакета (формируется драйвером). Номер пакета наращивается на 1 при запуске передачи, при повторе передачи номер пакета не изменяется. При приеме пакеты с повторяющимся номером пакета игнорируются.

Попытка передачи является неудачной при:

* 1. Получении от драйвера *EcanDriver* кода отказа;
  2. Передача пакета драйвером *EcanDriver* превышает время, равное 10 вызовам функции ***ManageCanDriver\_run()***.

Количество попыток передачи равно 4. При исчерпании всех попыток передачи драйвер перейдет в ЗС.

Запуск передачи проводится в том случае, если до конца главного цикла остается время, равное времени двух кратной передачи пакета. В противном случае запуск откладывается до начало следующего цикла.

**Применение**

Для использования этого драйвера требуется:

1. включить его в проект;
2. организовать инициализацию драйвера, обычно проводится в файле MainInitial.c;
3. Вызвать на выполнение функцию ***InterChannel\_runDrv()*** с частотой 32 кКц. Эта функция вызывает ***ManageCanDriver\_run()***.

**22 Модуль readInstructionCode**

Модуль *readInstructionCode* предназначен для использования компонентом Testing при подсчете КС ПЗУ.

Модуль *readInstructionCode* состоит из одноименного модуля и содержит одноименную функцию. Функция возвращает значение типа uint32, где код команды содержится в 24 младших битах, 8 старших бит содержат 0.