1. **Общие сведения о системы.**

Прикладная программа в системе управления Sider (дале по тексту “скрипт”) пишется на языке высокого уровня LUA. Язык LUA интерпретируемый, это означает, что скрипт выполняется виртуальным процессорам LUA – Lua runtime. Из скрипта нет непосредственно доступа к аппаратуре, всю низкоуровневую обработку сигналов осуществляют системные драйвера, выдавая в скрипт уже подготовленные данные о состоянии дискретных входов, значении токов нагрузки на транзисторных ключа, сообщения интерфейса CAN и т.д.

Поскольку в системе есть драйвера, занимающиеся управлением аппаратурой, и скрипт, выполняющий алгоритм управления, то система многозадачная. Синхронизацией работы всех задач занимается операционная система реального времени (ОСР). ОСР осуществляет постоянное переключения между различными задачами, обеспечивая:

* Гарантированное выполнение с нужной частотой задач реального времени ( например алгоритма защиты транзисторных выходов от перегрузки)
* Гарантировании запуск остальных задач, включая выполнение скрипта, в то время, когда процессор свободен от задач реального времени.

Структурно система представлена на рис. 1.



Рис.1. Структура системы.

Таким образом скрипт, это полноценная программа на языке высокого уровня, с полным и безопасным доступом к аппаратным возможностям системы. Системное ПО переводит аппаратуру в безопасное состояние, если в скрипте возникли какие-то сбои или логические ошибки.

В обще случае, скрит представляет из себя алгоритм обработки входных данных и формирования упрвляющих сигналов, который запускается в бесконечном цикле. Перед началом нового цикла системное ПО передает в скрипт текущие значения входных данных. После окнчания цилка системоне ПО получает от скрипат новые значения управляющих сигналов.

При этом, поскольку

Рис1.

Т

1. Из хоста скрипт запускается в 2 этапа. На первом создается coroutine, в которую  загружается кусок и запускается его исполнение ( do).  На втором этапе, в цикле вызывается функция с именем main из загруженного куска . Вызов происходит через coroutine.resume, до тех пор пока resume  возвращает OK или yeild
2. Программа на lua, должна представлять из себя кусок, с обязательно объявленной функцией main.  Но функция должна быть объяывлена main = function () В этом случае, при первом запуске куска после загрузки создаятся глобальная переменная main на которую уже можно в дальнейшем ссылаться при вызове resume. А вот если объявить function main()  то работать не будет. Надо уточнить про local. Возможно срабоатет так же local function main()
3. Тело main  должно содержать рабочий бесконечный цикл, в котором будет хотя бы один вызов coroutine.yeild в начале цикла. Соответственно этому вызову передаются в качестве параметров выходные данные - логические значения выходов и принимаются вхожные данные. Пока это токи каналов,и значения входов и время в милисекундах с момента предыдущего вызова. В начале цикла это надо вызывать что бы при первом проходе у кода уже были актуальные данные. Возможно будут еще какие-то входные данные, пока не придумал. время нужно для реализации абсолютных задержек. Пока я не уверен какой реально будет цикл исполненичя lua  скрипта, делать какие-то сложные функция таймеров не вижу смысла, поскольку есть по стуи глобальная переменная тика, которую уже надо запихивать во все написанные таймера на lua. Исходя из этого в теле рабочего цикла не должно быть других вызовов coroutine.yeild, иначе собьются таймера. ( понятно что можно обойти это через доп переменную, но думаю усложнять тут не стоит) и естественно не должно быть бесконечных циков или каких-то ожиданий чего-то, что бы не порушить переключение процессов.
4. До тела рабочего цикла в функции main можно писать последовательные инициализации или ожидания каких-то событий. Ну типа ждем пока на дискретный вход не придет сигнал разрешения работы. (типа пинаем coroutine.yeild пока вход не примет нужное значение. Вот тут можно писать все что угода, и вызывать coroutine.yeild  сколько угодно, поскольку это блок исполняется один раз перед рабочим циклом.
5. За пределами функции main должны быть расположены объявления глобальных переменных - входа, выхода, токи, таймер ( все с чем работает coroutine.yeild ) вызовы функций ядра для инициализации всего железа (настройки параметров каналов и вот это все) и запихнуты библиотечные функции и любые пользовтельские функции и псевдо-классы.
6. Передачу frame can можно достаточно просто реализовать через вызов Си функции ( уже работает), сложнее с приемом. Я пока еще не вкруил как это лучше сделать, ибо поскольку прямой обмен данными межлу си хостом и lua у нас в одону строну. В смысле что lua вызывает функции Си, а Си вызывает только функцию main из Lua. Соответственно если асинхронный пример, условный майлбокс то это дожно выгдятьи как-то так. Сначала где-то в инициализации вызвать функция настройки фильтра Can на нужный ID. А далее нужно как-то запрашивать у хоста, пришли ли данные или нет? А если пришли, то их забртаь в lua. Это можно сделать либо через Си функцию, гдеч она принимает таблицу как параметр. Собственно обратно флаг ready и в таблицу запихиваются из Си новые данные, или флаг что данных нет и тбалица тогда не обновлялась. Либо сделать отдельные функции на стату буфера - пришле нужный пакет или нет, и отдельную фунецию забрать данные, если они уже пришли. А вот опрос и обновление написать на lua.
7. Общие принципы работы PDM.
   1. Силовые выхода.

В модуле PDM есть 8 каналов на номинальный ток до 20А и 12 каналов на номинальный ток до 8А. Номинальный ток для каждого канала может быть задан программно. В связи с отсутствием в устройстве термокомпенсации измерений, в процессе работы устройства под высокой нагрузкой, реальная гарантированная точность измерения тока составляет ….

1. **Написание пользовательских программ.**
   1. **Исполняемые скрипты. Основные тезисы**

Пользовательские алгоритмы в системе PDM-Sider пишутся на высокоуровневом языке программирования LUA (версия 5.4.4, описание языка и всю дополнительно информацию по нему, в том числе на русском языке, можно найти на http://www.lua.org/).). Для пользователя доступен весь функционал и синтаксис языка, а сложность программы ограничена только размерами оперативной и FLASH памяти конкретной модели PDM. Программа для системы PDM-Sider может быть исполнена в любой другой среде LUA, например на персональном компьютере (на автономной LUA машине, файлы для разных операционных систем доступны на сайте http://www.lua.org/). Эту особенность можно использовать для отладки сложных алгоритмов, поскольку можно написать на LUA не только саму программу для PDM, но и скрипт для тестирования этой PDM. Подробнее об этом в разделе тестирования.

Структура программы на LUA для системы PDM-Sider должна соответствовать правилам, описанным в разделе 2.2. Весь алгоритм работы PDM описываются в виде кода в блоке рабочего цикла. Рабочий цикл, как следует из названия, постоянно исполняется ядром системы. После каждого рабочего цикла ядро производит обработку данных. В

* 1. **Правила написания скрипта.**

Структурно скрипт LUA для системы PDM-Sider должен состоять из определённых блоков, расположенных в следующем порядке:

**[блок конфигурации]**

**[блок пользовательских функций]**

**[блок рабочего цикла]**

* + - 1. **Блок рабочего цикла программы**

В общем случае шаблон пользовательского кода должен выглядеть следeющим образом:

**main = function ()** --функция main. Ее вызывает ядро PDM-Сайдер после старта.

**function stop() In1 = coroutine.yield(Out20,Out19…0ut1) end –**

**while true do** -- бесконечный цикл

**stop()** --обмен данными с ядром

**[Пользовательский код]**

**end** -- конец бесконечного цикла

**end** – конец функции main

Пользовательский код должен быть размещена внутри бесконечного цикла в функции main. Для обмена данными с ядром необходимо вызывать функцию stop внутри бесконечного цикла, перед началом пользовательского кода.

*Функция main – это предопределенная точка входа в пользовательскую программу, т.е. после старта системы и загрузки скрипта, ядро PDM-Сайдер пытается запустить из скрипта функцию c именем main. Можно писать любое количество сервисных функций, однако имя main зарезервировано для осинового рабочего цикла.*

***NB! LUA позволяет объявить функцию и таким образом function main(), это тоже правильный синтаксис, однако при таком написании из-за особенностей языка LUA, PDM не сможет ее увидеть в скрипте и запустить.***

*Непосредственно алгоритм работы устройства должен быть помещен в любой бесконечный цикл внутри функции main. В шаблоне это цикл “while true do”. Это необходимо именно потому, что пользовательская программа полноправный процесс, и если код не помещён в бесконечный цикл, то он будет выполнен только один раз.*

***NB! Внутри функции main можно вставлять любый циклы в любое место, однако они не должны быть бесконечными. В противном случае программ никогда не сможет из них выйти, что приведет к перезагрузке системы по watchdog.***

*Внутри системы PDM-Sider пользовательская программа реализована виде потока LUA (thead). Поэтому, что бы процессор PDM выполнял не только пользовательские задачи, но и запускал внутренние системные процессы ( например контроля тока через выходные ключи, обмена по CAN) пользовательская программа должна приостанавливать периодический свое исполнение. Это делается через специальную функцию LUA coroutine.yield. Тут происходит обмен данными с PDM. Когда LUA программа доходит до этого вызова, она передает в PDM управляющие данные для выходов, а при возобновлении работы получает из PDM значения токов и другие данные.*

*Для удобства работы, в шаблоне вызов coroutine.yield оформлен в виде локальной функции stop(). Резюмирая, пользовательский алгоритм должен быть размещен внутри бесконечного цикла в функции main.*

* + - 1. **Блок конфигурации.**

**В общем случае блок конциграции должен выглядить следующим образом**

**Out1**

После старта PDM, ядро настраивает все параметры аппаратуры по умолчанию. Если параметры по умолчанию соответствуют необходимым, то в конфигурации конкретно этих параметров необходимости нет. К тому же, явная конфигурация из LUA скрипта увеличивает его размер. Однако, для повышения надежности всей системы, явная конфигурация в LUA скрипте предпочтительна. Это позволит избежать проблем, при последующих изменениях LUA кода.

Конфигурирование модуля PDM происходит через вызов специальных функций ядра PDM. Ядро регистрирует эти функции в среде LUA перед загрузкой скрипта. После этого, эти функции доступны для вызова из LUA программы. Для пользователя это означает, что функции конфигурации можно вызывать в любом месте программы любое кол-во раз. При этом, смена конфигурации происходит в реальном времени (т.е. в момент вызова соответствующей функции).

**NB! При запуске скрипта на автономной LUA машине необходимо использовать функции “заглушки”. Подробнее о запуске программы в автономной LUA машине см. соответствующий раздел.**

**Конфигурация мощности и режима плавного пуска.**

OutConfig(пар1,пар2,пар3,пар4)

пра1 – номер канала от 1 до 20

пар2 – номинальный ток канала. 0.1-20.0 для каналов 1-8 и 0.1-8.0 для каналов 9-20.

Значения по умолчания 20.0 для квналов 1-8 и 8.0 для каналов 9-20.

пар3 – время плавного пуска в милисекундах, от 0 до 65535. Значения по умолчания 1000 для каналов 1-8 и 0 для каналов 9-20.

пар4

1. **Работа с CAN.**

**2.1 Предопределенные функции работы с CAN**

**В системе предусмотрены системые следущие сисетмные вызовы:**

**CanSend( id, data1,….data8)**

**SetCanFrame**

**IsCanFrame**

**GetCanFrame()**

**CanSendRequest()**

**GetAnswer()**

**Данная системная функция отправляет пакет c идентификатором id в сеть CAN. Длинная пакета будет зависеть от количеством байт, которые передаются при вызове. Минимально должен быть хотя бы один байт, максимально 8. Если необходимо передать один байт данных, а потом 7 нулевых байт, то вызов должен выглядеть CanSend( ID, 0x34,0,0,0,0,0,0,0).**

**CanSendRequest( id, id\_res, data1,….,data8)**

**Данная сисетмная функция рабоатет так де, как CanSend, но перед оптрвкой настравиает интерфейс Can на прием пакета с идентефикаторм id\_res. Полечение ответа от устройства осуществлятеся с помощью функций GetCanAnswer**

Библиотека CAN

Объект CanOut

Конструктор объекта

:new( addr, time, size, d1, d2, d3, d4, d5, d6, d7, d8 )

addr – обязательный параметр, CAN ID c которым будет отправлен пакет. Поддерживает как стандартный формат идентификатора, так и раширенный. Для отправки раширенного идентефикатора неоходимо добить к адресу флаг EXT\_CAN\_ID.

Пример

MY\_CAN\_OUT = CanOut:new(0

**Библиотека PDM20**

**Yield ()**

Функция завершения работы текущего цикла скрипта. Передает в систему значения выходных переменных и приостанавливает выполнение LUA Runtime. После того, как система выполнит свои внутренние процессы, исполнение скрипта продолжиться со строчки кода, следующей за Yield() и продолжиться до следующего вызова Yield(). Перед тем, как скрипт начет новый цикл, функция Yield() загрузит в новые значение входных переменных.

**getDelay()**

Функция возвращает время в мс с момента передающего вызова функции Yield(). Данная функция используется для расчета задержек и таймаутов.

**getRPM ( ch )**

функция возвращает частоту в ГЦ на входе счетных каналов. Возвращаемое значение имеет тип float, номер канала integer и может принимать значение 1 или 2

**getOut( ch )**

функция возвращает логическое значение выходного канала ch (true или false)

**setOut( ch, data)**

Функция устанавливает новое логического значение выбранного выходного канала ch. Если необходимо установить значение на основании целочисленной переменной, рекомендуется использовать явное преобразование типа, например ***setOut ( 1, some\_var ~= 0 or true and false).***

**getOutStatus( ch )**

Возвращает статус выборного выхода ch. Статус выхода может принимать следующие значения

0x00 – выход выключен

0x01 - выход включен

0x02 - обрыв цепи

0x03 – перегрузка

**getCurrent( ch )**

Возвращает значение тока нагрузке канала ch во float

**getCurFB( ch )**

Возвращает целую часть значение тока нагрузки канала ch

**getCurSB( ch )**

Возвращает дробную часть тока нагрузки канала ch

**getCurLSB10( ch )**

Возвращает старший байт значения тока нагрузки канала ch, умноженного на 10. LUA не может проводить битовые операции с числами типа float. Поэтому передача данных по CAN возможно только в том или ином формате с фиксированной точкой. умножение

**getCurMSB10( ch )**

Возвращает младший байт значения тока нагрузки канала ch, умноженного на 10. LUA не может проводить битовые операции с числами типа float. Поэтому передача данных по CAN возможно только в том или ином формате с фиксированной точкой. умножение

**getDIN ( ch )**

**igetDIN (ch )**

Функции возвращают состония дискретого входа ch. Функция getDIN возвращает заначение true или false, функция igetDIN возвращает 1 или 0.