Tauria OÜ

GraDisLib v-1.0.2 User Guide

Gradis ver 1.0.2

Introduction

GraDisLib is library made for easy and fast development of GraDis device. GraDis device is microcontroller systm that is ready to use in end products. It does include Graphical dot-matrix display, ARM Cortex M3 processor, programmable peripheral system.

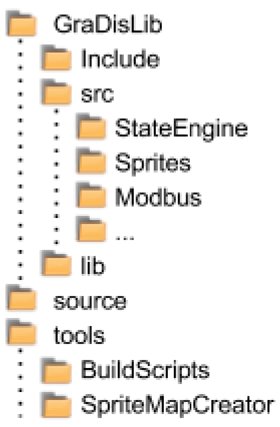
This document does describe how to use GraDisLib on your application.

It is described the design patterns and how the system does work. You will get introduction into state engine, managing system power, how to start new project based on template project and more.

Table of Contents

Template Project Organisation

Part of the template project organisation is depicted on the picture right side of this chapter. The folder structure is made for easy using it inside Eclipse IDE project. The **Include** folder contains header files of different modules for inclusion in projects. The **lib** folder contains prebuilt object files that can be linked with the target. The **src** folder can be absent if you do not have license to see source code of the modules.

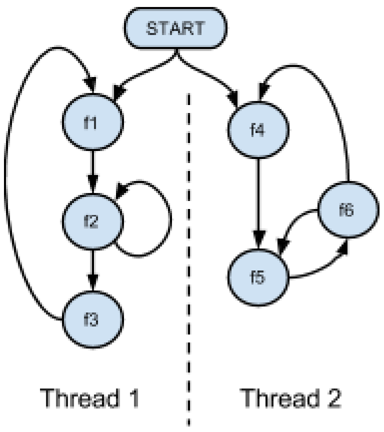


Under the **tools** folder are some important applications. Like application for creation of sprite map for displaying graphical icons, characters, sprites and drawings on the screen. BuildScripts contains useful scripts for building the project. For instance StateEngine does have prebuild script that will easify state declaration writing and counting of number of threads.

State Engine

Introduction

State Engine implements cooperative multitasking system. It helps to create several parallel running threads that are executed in round-robin style: after one thread finishes its function execution, the other thread's function is called. Usually in embedded firmware, there is needed to wait behind hardware and the computing time is expensive. Waiting in dead loop is not acceptable.



Idea behind the state engine is not to wait in dead loop but return from the function and run another functions in between. On the illustratory flow diagram left side all the circles depict functions (f1)...(f6) that do something. The arrow lines show which function will be executed next time in the thread. If there is needed to wait behind other party when will it complete its function, the hardware status is monitored at the beginning of the function (f2) and if there is no information the function returns. Next time when the (f2) discovers that the information has arrived, it will process it and if needed will change the function to be run next time (f3). When thread 1 function returns, then thread 2 function will be executed and then thread 1 function again. There is no limit defined for the number of threads.

Why Co-operative Scheduling

There are lots of operating systems available for embedded systems. Most of them implement pre-emptive task scheduling. The pre-emption will eliminate the risk of buggy deadlock in one thread that will stop entire system. In our framework we expect that developers must know the system, it must be simple and understandable. ***Developers must avoid deadlocks*** in all steps using the State Engine's pattern. It must be made sure that one function does not take too much time (time meaning comes from specific application). It is not acceptable of not knowing all the corners of the firmware. A very clear advantage of cooperative scheduling is that there is ***single stack space***. In preemptive operating system, each task sitting in its own deadlock must have dedicated stack space. This will dramatically reduce RAM availability.

State Engine architecture does not eliminate pre-emption. Interrupt routines still preempt the state engine's functions. The state engine is designed to run inside idle thread but can be executed inside interrupt routine. The architecture allows multiple state engines in the system too although it might not give any clear advantage of doing so.

State Engine will ***increase the code readability*** by separating different system states into different functions. Usually behind the function or state is deeper understanding about the physical system where the firmware is used. It helps to reduce cryptic conditional statements between many variables to perform an operation. The condition is separated into specified function to run next time that performs the operation and validates that the state is still corresponding to physical system.

When doing co-operative scheduling with single core of CPU, there is ***much less worry about race conditions between threads***. It is warranted that another thread is not going to modify the memory field at the same time. Although care must be taken that interrupt routines do not interfere.

All of this leads to ***minimal memory footprint*** and less time spending on managing the threads.

Basic Usage

In order to get State engine running, at least the following must be done in your project:

main.c

```
#include "state_engine.h"

state_engine_t *my_engine = 0;

int main()
{
    // setup the idle thread engine structure
    // if you have more than one engine, you must init all the engines before ...setup()
    state_engine_init( &my_engine );
    // and initialise all threads of all engines and hardware, calls SE_INIT constructors
    state_engine_setup();
    // make the interrupts running e.t.c.
    system_hardware_init();
    // enable global interrupts
    CyGlobalIntEnable;

    for(;;)
    {
        // check if the system should power off part of its functions or hibernate
        state_engine_powersave(
            // run all threads of the idle once when they are timed out
            state_engine_run( my_engine )
        );
    }
}
```

The *my_engine* is holding state engine's state variables. This way you can have many engines. You need to init the hardware, state engine's variables and let it know where is the array of threads. Threads will be initiated. The threads are running when the run method gets called. In every call of run method the engine will call one function from every thread.

example_thread.c

```
#include "state_engine.h"

SE_INIT( exampleThread_init, 5 )
{
    state_engine_append_thread( my_engine, exampleState_1, 0 );
}

SE_STATE( exampleState_1 )
{
    state_engine_set(se_thread, exampleState_2);
}

SE_STATE( exampleState_2 )
{
    state_engine_set(se_thread, exampleState_1);
}
```

The *preprocess.sh* script will make prototypes for functions *exampleThread_1* and *exampleThread_2* for you. So you do not need to worry about in which order are the functions defined. The declarations are included through *state_engine.h*. *SE_INIT* and *SE_STATE* are macros for defining and declaring corresponding functions. *SE_INIT* is kind of constructor. In the linking phase of the application, all *SE_INIT* functions will be pointed in ordered manner. The ordering is made according to second argument of SE_INIT. All the init function will then be called inside *state_engine_setup*. In the init function you can append current thread into engine. With this you are going to tell to the specific engine the function it will execute next and the last parameter is the thread's context pointer. With help of the context pointer, you can create software where you can clone the same thread. Method *state_engineset* is used to tell the state engine, which function it must execute next time. With this function you define state transitions. If you do not specify state transition, the same function will be called next time by the engine. The variable *se_thread* is passed as an argument to the state function. There is also argument called *se_context*, which points to the memory of context structure specified during init of the thread.

Advanced Usage

Additional to the basics, you can place the thread to sleep. This means that during minimum of specified period of time, the thread's function will not be executed. The time is usually longer and depends how much time other threads use for execution. You can place the thread into infinite sleep- in this state the thread never wakes up. You can drive other threads, when you know the other thread se_thread pointer, you can change its states, change its sleepiness. So you can have a thread that you use as sub-thread. There is macro that simplifies verification of the state of a thread or of the subthread.

Hardware Functions

The hardware dependent functions like system clock reading is placed into hardware dedicated file named . In that file, state engine itself will know if it does have nothing to do. There it is possible to decide whether system must go to sleep for power saving as there are no states to run and only a interrupt can start something useful, or as you wish go entirely into hibernation.

The hardware dedicated functions are:

```
uint32_t state_engine_current_time_us( void );
```

is function to receive current value from system timer in microseconds.

```
void state_engine_system_sleep_us( uint32_t delay);
```

is function that state engine will call this method when there is no functions to execute. Depending from the delay you can decide wether to sleep or not.

```
void state_engine_sleep_wakeup_handler( void );
```

is function that will be called from hardware timer interrupt to wake up the system from sleep.

```
void state_engine_sleep_release( void );
```

is function will be called by state-engine itself when a state is changed. It is possible that state engine is sleeping and from another interrupt routine you want that the engine will wake up.

```
void state_engine_hardware_init( void );
```

is function that is used to initialise the hardware functions, like microsecond timers and others.

```
void state_engine_critical( bool critical );
```

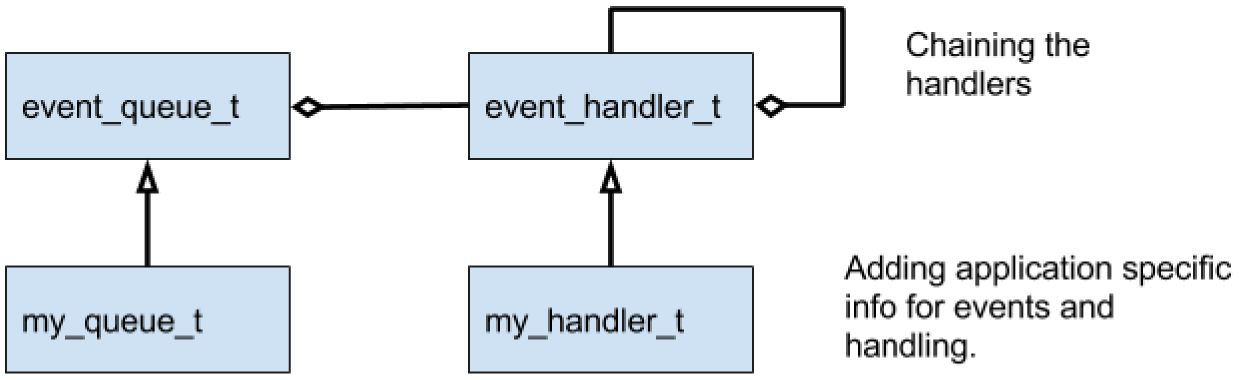
is function that is used to specify actions what to do when state engine does indicate it is doing something critical. Usually you want to disable and enable interrupts here.

Event Queue

Introduction

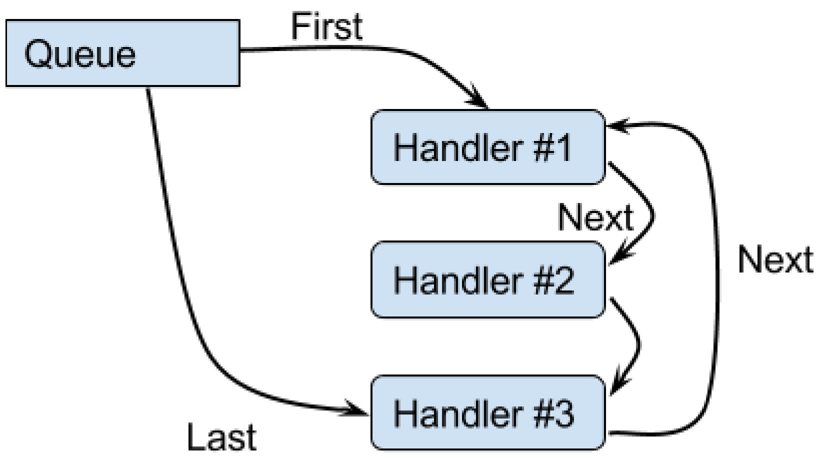
Event queue is made for inter-process communication. It is useful for handling requests to perform very short actions or sending data between threads when the data does change. Event Queue is queue of handlers. Same thing can be viewed also as queue of actions. Sometimes the requests happen in short period of time but handling takes much more time and must be made in series. So, the requests can be placed into queue and handled one-by-one. For example EEPROM handler requests are placed into queue and handled one at a time.

Following class diagram shows how the system is arranged.



My_queue_t will hold event source side of data. When calling the handler, the handler function is getting pointer to this data. My_handler_t will hold the handler side of data. This data is also presented to the handler function. Depending on the specific application, event source side may analyse the handler data and decide what to do with the handler.

Sample of the objects in queue:



Usage Overview

First, the event caller must create event queue object and manage it. If there is no specific data needed the default type *event_queue_t* may be used. The fact of event call itself will carry one bit of information about the situation.

```
event_queue_t   my_queue;

// inside function ...
event_queue_init( &my_queue );
```

```
// calling only the first handler in the queue
event_queue_handle( &my_queue );

// or calling all the handlers
event_queue_handle_all( &my_queue );
```

As shown, you may want to control how to call the handlers by calling them one by one. In simple event cases you want to call them all. It will stop calling the handlers when one handler blocks and newly inserted handlers are not executed.

The handler side must declare handler object that will be placed into event queue. Following does show handler with specific data.

```
typedef struct {
  event_handler_t  parent; // the base of event handler's information
  uint32_t  my_handler_info; // specifics what the event handler is looking for
  //...
} my_handler_t;

// define the object
Static my_handler_t the_handler;

// define the handler function
Static evh_return_t the_callback( my_handler_t *h_data, event_queue_t *e_data )
{
  // The return value determines how event hadling continues:
  //  CONTINUE - next handler in queue will be called
  //  BLOCK - next handler will not be called this time
  //          and this handler will be called again next time
  //  REMOVE - next handler in quee will be called
  //          and this handler will be removed from queue
  return CONTINUE;
}

// inside initialisation function ...
event_handler_init( (event_handler_t*)&the_handler, (event_callback_t*)the_callback );
event_register( &my_queue, (event_handler_t*)&the_handler );
```

Handler’s return value will determine how the current event handling continues and wether the handler will receive next event too.

Return Value	Handling continues this time	On next event
CONTINUE	yes	Is called as usual
BLOCK	no	Is called as first handler
REMOVE	yes	Will not be called

System events

The GraDisLib does define some specific events.

se_event_ask_hibernation

This event will be called by the state engine when it needs to decide whether the system needs to go into sleep or into hibernation. Inside sleep only cpu will halt, inside hibernation all power consumption must be reduced. This event side will define special boolean variable that must be set to false when the hibernation is not permitted.

se_event_do_hibernation

This event is called when it has been decided to enter into hibernation. All threads that are turning on some hardware need to listen it and turn the hardware off.

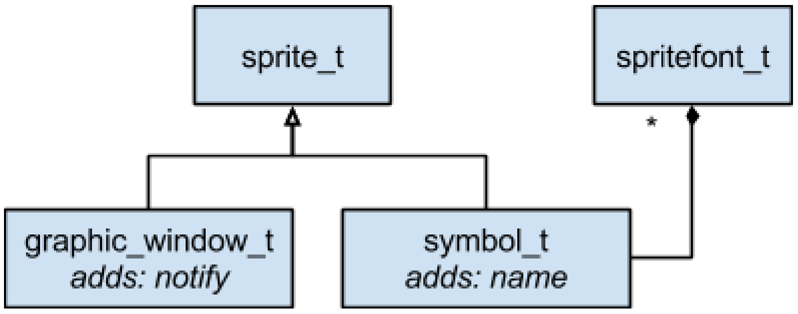
se_event_wake_hibernation

This event is called after the system has been waking up from the hibernation. All threads that need some hardware to be powered on, must turn the power on during handling this event.

Sprites

Introduction

Sprites are dot-matrix pictures that can be drawn into another sprite and transferred onto the screen of dot-matrix display.



Sprite, Window, Symbol, Font

sprite_t

is base class for dot-matrix picture. It keeps record, where the image is stored in memory, what are the image sizes in pixels and how to modify cursor after drawing the sprite into another sprite.

graphic_window_t

adds method to notify the sprite owner that the sprite has been renewed. This way, if someone modifies the sprite, the owner will get to know that it is changed. For example, the LCD driver will start refreshing the LCD display with new data.

symbol_t

adds 4 byte name to sprite. The name can be handled as integer number for fast searching and comparison.

spritefont_t

collects symbols into array. This array must be ordered for binary search. The sprite map creator is aligning the symbols into ordered list of sprites in the font array.

Symbol and Font Preparation

Symbol is named sprite. In the sprite all the bits are packed so that the sprite would use minimal space. In the font all bits of all symbols are packed together without separation to save space. Also generated symbols are kept in flash memory area and are not loaded into RAM to save ram space.

Symbol and font creation is made simpler with tool named SpritemapCreator. This is little utility that is able to read pixels from XPM file and write them into C code file. Compiler is then compiling all of it into program code.

There are rules that must be followed in order to be successful with SpritemapCreator. This simple tool complains a little on errors and by not following the rules you end up with no items in font. There is 2 input files for map creation: the XPM file and a TXT file. Inside the TXT file the font name, symbol name and how to find the sprite from xpm file are written. The files must be named the same with different extension. You must understand that compiler is going to create also the object file with same name, and some structures into the object file with names written into txt file. So, make sure the names will not collide with the other names in the code.

XPM File Rules

Inside the XPM file, there is allowed only few colors. The amount of colors must be as small that the xpm file identifies the pixel only with one character. Predefined color codes are used to identify sprite boundary and dots.

The **clear bit** color is white, “**#FFFFFF**” and the value must be exact.

The **set bit** color is black, “**#000000**” and the value must be exact.

Any other color is non-sprite color and there must be no set or clear bit between sprites. In other words, the sprite must be rounded with other color with at least one pixel width that is not set or clear. The sprite area surrounded by other coloured bits must be rectangular. If it is not rectangular, then in many cases error will be printed and map creation will fail. It is not guaranteed to fail.

TXT File Rules

The text file is organized into columns which are separated with one or more space character. One symbol is defined in one line.

#	Name	Description
1	Font Name	Must be formatted in US ASCII characters following also rules for gcc literal standards. All the symbols in one font must be together. It is not permitted to have one row in with FONT1 then row with FONT2 and then row with FONT1 again. Sprite Creator will treat them as different fonts and compiler will fail later on while producing the binary.
2	Symbol Name	Must be formatted in UTF-8 characters. Space and \ character are special. \ character is used to escape next 2 characters which format hexadecimal number for character value. To have \ character itself as part of name, you must write \5c, where 5c is hexadecimal number for character code for \. Space can be entered as \20. Up to 4 characters is supported.
3	X	The point X coordinate on the xpm picture. Must be somewhere inside the spite boundary area. The top left corner will be searched by creator.
4	Y	The point Y coordinate on the xpm picture. Must be somewhere inside the spite boundary area. The top left corner will be searched by creator.
5	dX	In range -15..15. The increment of X coordinate after drawing symbol additional to sprite width. When drawing the symbol, the drawing function will provide next X coordinate for next sprite drawing. This simplifies drawing text. This field helps to

		design drawing by leaving empty space between symbols when > 0. Also for condensing those for example when writing italic symbols when < 0. Then next symbol will be drawn littlebit on top of previous.

Graphical Functions

GraDisLib does have several useful graphical functions for drawing box, frame, line, point, sprite into another sprite and utf8 encoded string symbols into sprite. All graphic function either draw points or erase the points. The last boolean argument specifies whether to draw or erase. This way you can draw one gesture on top of another. If you want clean background then draw a box with the flag of erase dots. The graphic functions are optimised to draw many dots at once if possible.

Display Driver

Most importantly for user, the display driver defines a window (derived from sprite) that is named **window** and can be accessed as sprite with help of macro **MAINWIN**. The next macro **WINDOWNOTIFY** helps to notify the window owner about change inside the window. Display driver starts to send the window sprite into LCD display inside background interrupt routine after notification. The macros are defined in file graphic.h.

Modbus

Introduction

Inside GraDisLib are methods and macros that help to solve the ModbusRTU protocol handling task. In Modbus, the communication works as reading and writing registers. There is 2 register types: coils and registers. Coils are 1 bit width registers and registers are 16 bit width registers. To organize different type of registers into the system one must combine the primitive types. Register specification must be handed to clients on printed or electronic document that is human readable. The protocol itself does not include means of defining how the data is organized.

Architecture Overview

On the level of communication the reading and writing is done on the primitive register level. I.e. 1 bit coils are read and written or 16bit registers are read and written. The application level of both communicating parts must know exactly how the data is structured into the primitive registers. Read from the Organising Values and Structures about how GraDisLib helps making the system into high quality. GraDisLib helps to avoid misaligned read and writes into registers and will help to return error response to another part.

Integration with Communication

Modbus does expect response from slave device. Slave devices do not initiate the messaging on its own. So, the device will get over channel a packet and must create response to this packet. For this, GraDisLib does have a function to call:

```
void modbus_pdu_handle_request( modbus_pdu_be_packet_t *in, modbus_pdu_be_packet_t *out );
```

The packet in was received from communication channel and the packet out is where the response will be formatted. The function will process the request inside in packet and always fill the resulting out without sleeping of any kind. The packets you can initiate like this:

```
static uint8_t buffer[TELEGRAM_MAX_LEN];

static modbus_pdu_be_packet_t in_packet =
{
        TELEGRAM_MAX_LEN,
        (modbus_pdu_generic_t*)buffer
};

uint8_t responsebuf[TELEGRAM_MAX_LEN];

static modbus_pdu_be_packet_t out_packet =
{
        TELEGRAM_MAX_LEN,
        (modbus_pdu_generic_t*)responsebuf
};
```

The sleeping part is partially true, it depends on the data model implementation, there are functions that must validate the request. You must write those functions and if you will write blocking cycle there, all state engines will stop. Read on how to write the data model implementation.

Organising Values and Structures

GraDisLib helps you to create modbus registers and structures in the way that does ensure, the reading and writing of entire multi register structures at once and not partially. So the design is to make sure client reads or writes all the registers of a structure or multi register value. When this is not accomplished, then error must be returned. This restriction is required because time-delay between reading of different registers may result of half the values to be from different state of system.

`LEN_ELEMENT(elem)`

Gives the length of element in 16bit registers. The elem must be a C variable, or structure as it is passed into sizeof operator. It must be resolvable in compile time.

`SID_ELEMENT(a,str,elem)`

Returns index of start of the element in 16bit registers array. If elem is NOT an array, then in place of the a you must write &. If elem is an array then leave a empty. str is the structure variable and element is the member name inside the structure.

`EID_ELEMENT(a,str,elem)`

Returns index of start of the next element in 16bit registers array.

`IS_ELEMENT_REG(id,a,str,elem)`

Returns true, if the index id is inside the element in 16bit registers array.

`IS_ELEMENT_START(id,a,str,ele,typ)`

Returns true, if the index id is start of the element in 16bit registers array. The typ is typename of the array element. For example you have a array of 32bit values in registers and you want to make sure the id is start index of one of the array elements. Ofcourse this macro can be used with a variable, then this is like 1 element array and in place of a you must write & sign.

Example Holding Registers

First declare the structure of the registers. According to this structure, the communication will take place. It is upt to you and the data model implementation how you will allow the reading and writing this structure.

```c
typedef struct __attribute__ ((packed))
{
    uint16_t num_of_points; // number of points in points array
    points_t points[MAX_NUM_OF_POINTS]; // the points array
    float area; // the area of the gesture inside points
}
holding_registers_t;
```

Ofcourse the points_t is also defined somewhere before.

```c
typedef struct __attribute__ ((packed))
{
    float X; // coordinate X of the point
    float Y; // coordinate Y of the point
}
points_t;
```

For example if MAX_NUM_OF_POINTS is 4, then the registers map would look like this:

Reg #	Meaning
0	num of points
1	points[0] #1,2 - X #3,4 - Y
2	
3	

4	
...	...
13	points[3] #13,14 - X #15,16 - Y
14	
15	
16	
17	area
18	

And the goal is not to allow reading or writing only register 18 for example or registers 2 and 3. Goal is to allow operation only if all the registers 1-4 are covered for points[0], or registers 17,18 are covered for area at once.

Then, you need to reserve the memory for the holding registers in global scope.

```
holding_registers_t holding_regs;
```

We must provide information to modbus handler about how to call our data model handling functions.

```
const modbus_pdu_copy_register_collection_t holding_registers_collection =
{
        REG_COLLECTION_IN_RAM(holding_regs),
        holding_done,
        REG_FUNCTIONS_START
        copy_points_element,
        copy_single_holding_register,
        copy_area_element,
        REG_FUNCTIONS_END
};

const modbus_pdu_conf_t mbpdu_data_model_conf =
{
        &input_registers_collection,
        &holding_registers_collection
};
```

From the structure of mbpdu_data_model_conf, the modbus handler will get the information where in the ram are our holding regs placed. The holding_done is a function that modbus handler will call if the packet from the network has been fully and successfully handled. REG_FUNCTIONS_START is a macro that evaluates to “{“ and REG_FUNCTIONS_END does evalueat to “0 }”. At the end of modbus_pdu_copy_register_collection_t is a variable length array of function pointers. Modbus handler is calling these functions as long as one of them will handle the request or until it does reach the 0 pointer written with REG_FUNCTIONS_END. So, real copy is made of these functions.

Function for copying the area element. This function will be called to copy values from holding_regs memory to communication buffer and otherwise. The dest and src poitners are set accordingly.

```
static bool copy_area_element(
        uint16_t *id, // the starting id, will increase on success
        uint16_t maxcount, // maximum allowed registers
        uint16_t *dest, // memory address from where copy
        uint16_t *src, // memory addres to where copy
        modbus_pdu_exception_code_t *except ) // returnable exception in case of error
{
    typedef struct __attribute__ ((packed)){ float value; } mycopy_t;
    mycopy_t *D = (mycopy_t*)dest;
    mycopy_t *S = (mycopy_t*)src;

    // verify that the *id is inside of the area element
    // note that the argument a is typed as & because holding_regs.area is not an array
    if( ! IS_ELEMENT_REG( *id, &,holding_regs,area) )
        return true; // continue handling, nothng to do for this func

    // verify that maxcount of registers covers this element
    if( LEN_ELEMENT(float) > maxcount )
    {
        *except = MBPDU_ILLEGAL_DATA_VALUE;
        return false;
    }

    // verify that the *id is index of start of the element
    if( ! IS_ELEMENT_START( *id, &,holding_regs,area,float) )
    {
        *except = MBPDU_ILLEGAL_DATA_VALUE;
        return false;
    }

    // do the actual copy from S to D with swap of byte order
    (*id) += SWAP_ORDER( D->value, S->value );
    return false;
}
```

For handing copy of points registers, we allow one array element to be copied at once.

```
static bool copy_points_element(
        uint16_t *id, // the starting id, will increase on success
        uint16_t maxcount, // maximum allowed registers
        uint16_t *dest, // memory address from where copy
        uint16_t *src, // memory addres to where copy
        modbus_pdu_exception_code_t *except ) // returnable exception in case of error
{
    points_t *D = (points_t*)dest;
    points_t *S = (points_t*)src;

    // verify that the *id is inside the array area
    // note, the a element is left empty here as points is array
    if( ! IS_ELEMENT_REG( *id, ,holding_regs,points) )
        return true; // continue handling

    // verify that the maxcount covers entire points_t element with X and Y coordinate
    if( LEN_ELEMENT(points_t) > maxcount )
    {
        *except = MBPDU_ILLEGAL_DATA_VALUE;
        return false;
    }

    // verify that the *id is start of single element in the array
    if( ! IS_ELEMENT_START( *id, ,holding_regs,points,points_t) )
    {
        *except = MBPDU_ILLEGAL_DATA_VALUE;
        return false;
    }
    // do the actual copy from S to D with swap of byte order
    // note that we copy the structure contents element by element
    (*id) += SWAP_ORDER( D->input, S->input );
    (*id) += SWAP_ORDER( D->value, S->value );
    return false;
}
```

EEPROM Driver

Introduction

EEPROM is electrically eraseable and programmable read only memory. Reading of it is generally simple: as it is mapped into memory map, you just need to read the correct place. For writing specific module is needed to be used and writing takes time. Therefore special state thread is implemented for EEPROM functions for both reading and writing.

Architecture Overview

EEPROM is implemented as separate threads of states. It does have event queue for keeping the orders in line. So, the EEPROM user must submit event into EEPROM thread for both: reading and writing.

Organising Data Storage

EEPROM storage must be organised by declaring a structure for entire application:

```
EEPTH_FILE
{
    uint8_t A;
    uint16_t B;
    uint32_t C;
} __attribute__((packed));
```

The attribute packed tells to compiler not to leave empty bytes inbetween the elements. EEPROM is expensive. Then the threads that use the values from must have event handler object:

```
eepth_action_t eeprom_b_actions; // the event handler object
uint16_t MyB; // the local copy of the EEPROM data value in RAM
// inside initialisation function...
eepth_init_action( &eeprom_b_actions, &eepth_file.B, (void*)&MyB, sizeof(MyB) );
eepth_request( &eeprom_b_actions, EEPTH_READ_REQUEST ); // read initial value from EEPROM
// when need to know if the value has been read out ...
if( eepth_action_state(&eeprom_b_actions) == EEPTH_READ_DONE )
{
  // when it is needed to read or write something ...
  eepth_request( &eeprom_b_actions, EEPTH_WRITE_REQUEST );
}
```

In the example above, the handler does work only with the value B. Some other threads are using A and C.

Related Documents

...

Document Changes

- 2015-08-22 v1.0.0 (Peeter Vois) Initial document.
- 2016-11-21 v1.0.1 (Peeter Vois) updated the state engine and event queue to latest.
- 2016-12-02 v1.0.2 (Peeter Vois) improved event queue management.