Project to control a gasoline motor powered electric generator:

ESP\_MCU Folders & files structure:

* /src:
  + main.c
  + main.h
  + /energy\_data: //reading\_energy() thread
    - energy\_meter.c
    - energy\_meter.h
  + /motor\_data: //reading\_motor() thread
    - motor\_data.c
    - motor\_data.h
  + /motor\_control: //Write or read to DEMVE server
    - motor\_control\_srvc.c
    - motor\_control\_srvc.h
    - transmission.c
    - transmission.h
* /pi\_spi\_bcm2835: //Raspberry Pi SPI driver
  + pi\_spi\_bcm2835.c
  + pi\_spi\_bcm2835.h
  + /bcm2835:
    - bcm2835.c
    - bcm2835.h
* pi\_m16c\_spi: // M16C62P SPI driver and middware
  + definitions.h
  + sensors.h
  + m16c\_endianness.h
  + m16c\_spi.c
  + m16c\_spi.h
  + pi\_m16c\_spi.c
  + pi\_m16c\_spi.h
* pi\_ade7880 // Energy meter driver and middware
  + pi\_ade7880\_config.h
  + /src:
    - ade7880\_driver.h
    - pi\_ade7880\_drv.c
    - pi\_ade7880\_drv.h
    - pi\_ade7880\_gpio.c
    - /ade7880:
      * ade7880\_configuration.h
      * ade7880\_registers.h
      * preamble.h
      * ade7880\_spi\_protocol.c
      * ade7880\_spi\_protocol.h
      * ade7880\_srv\_cmd\_handler.c
      * ade7880\_srv\_cmd\_handler.h
* /libIEC61850\_server:
  + Description\_on\_motorcontrollerIED.xls
  + motorcontrollerIED.icd
  + iec\_server.cfg
  + static\_model.c
  + static\_model.h
  + /lib:
    - libiec61850.a
    - libiec61850\_wrapper.a
  + /include
    - All libIEC61850 related include files.
* /Documentation

Files content

* main.h:

To avoid "includes" doubled defined and the headache that they may carry, general and common 'includes' that belong to the files under /src folder are all declared once and here in this file, such in a way that the source files must refer to this 'main.h' to get them. In exceptional cases that a particular header file is required uniquely by an internal source file, this may be included only in the related source file. This idea is similar to the applied by Labrosse in his system, µC/OS

Includes as well, the set of definitions of a new type of variable to hold the readings from the two different sources. The ADE7880 energy meter and the M16C/62P µController that reads and writes to the motor. By using them, one may access the read values easily. They are similar to those defined inside the drivers folders but those are optimized for tx rv purposes inside the driver, and they may not be accessed from outside of each environment. Follow this order:

* server\_data\_st
  + mc\_data\_st motor
    - uint8\_t command;
    - uint8\_t cmd\_addr;
    - mc\_tval\_st torque;
    - mc\_tval\_st suction\_temp;
    - mc\_tval\_st coolant\_in\_temp;
    - mc\_tval\_st coolant\_return\_temp;
    - mc\_tval\_st block\_temp;
    - mc\_tval\_st exhaust\_temp;
    - mc\_tval\_st oil\_temp;
    - mc\_tval\_st crankShaft\_rpm;
    - mc\_tval\_st fuel\_in;
    - mc\_tval\_st fuel\_return;
    - mc\_tval\_st suction\_flow;
    - mc\_bval\_st bin\_input;
      * mc\_bin\_\_ut
        + uint16\_t bin\_all;
        + bit\_s

uint16\_t oil\_pressure\_OK :1;

uint16\_t emergency\_stop :1;

uint16\_t min\_fuel\_lvl :1;

uint16\_t annealing\_ON :1;

uint16\_t battery\_charging :1;

uint16\_t motor\_on :1;

uint16\_t motor\_off :1;

uint16\_t reserved :9;

* + em\_data\_st energy
    - em\_phase\_t phase\_a;
    - em\_phase\_t phase\_b;
    - em\_phase\_t phase\_c;
    - em\_neutral\_t neutral\_line;
      * em\_phase\_t
        + em\_tval\_st IRMS;
        + em\_tval\_st VRMS;
        + em\_tval\_st WH;
        + em\_tval\_st POWER;
      * em\_neutral\_t
        + em\_tval\_st IN\_IRMS;
        + em\_tval\_st INSTANT\_IN;
        + em\_tval\_st IN\_HXIRMS;
        + em\_tval\_st IN\_HYIRMS;
        + em\_tval\_st IN\_HZIRMS;
        + em\_tval\_st IN\_HXIHD;
        + em\_tval\_st IN\_HYIHD;
        + em\_tval\_st IN\_HZIHD;
        + em\_tval\_st ISUM\_HXVRMS;
        + em\_tval\_st ISUM\_HYVRMS;
        + em\_tval\_st ISUM\_HZVRMS;
        + em\_tval\_st ISUM\_HXVHD;
        + em\_tval\_st ISUM\_HYVHD;
        + em\_tval\_st ISUM\_HZVHD;

em\_tval\_st

float value;

uint64\_t timestamp;

i.e. to accesss the timestamp of the phase a instantaneos rms current in:

*server\_data\_st data; // declare an element of the variable from the header*

*uint64\_t irms\_timestamp = data.energy.phase\_a.IRMS.timestamp //use it*

* /src/main.c

Opens three new threads, one to handle the M16C readings (reading\_motor), another to the ADE7880 (reading\_energy)and one more to upload these values to the DEMVE system (motor\_control\_srvc).

A variable member of '*server\_data\_st*' is created previously, initialized and passed as a pointer to its memory address to the three threads. In this way all the threads have a common place to read from or write to.

To protect the access to this variable and accesses to the SPI driver, semaphores are created here too.

The infinite loop will catch any keyboard press and exit if 'q' is read. Each loop execution sleeps for 100ms

* /src/motor\_data/motor\_data.h:

Application unique PID

macros to convert ADC values to Flow and pulses to RPM.

* /src/motor\_data/motor\_data.c:

*reading\_motor()* thread.

In first execution initializes SPI communications if not initialized before.

Then calls M16C µController by reading all the sensors data in one call. One may call each sensor individually if required, but this is not efficient.

The received data is checked and filtered. If no error is detected, values are loaded one by one into a local variable together with a time stamp taken at that moment, to be passed altogether to the main container, the pointer to the '*server\_data\_st*' member.. This step is protected by a semaphore, as each access to it is. Time stamps are taken by the Raspberry Pi and not the M16C because this has not a realtime clock and to avoid the overload of this µController acting as a peripheral. Despite of this, the sensors are read all every microsecond in polling mode, the same for the bit values constantly checked, and flow and revolutions are measured by 16 bit timers at 750KHz and 3MHz respectively, triggering interrupts in a time gap from 0 to 87.38ms (flow) and 0 to 21.85ms(RPM). The total length of the transmission is 2 bytes in the MOSI and 33 back in the MISO, 35 bytes equals to 280bits. When the transmission speed is set to 3.8Kbps, 73ms is the total transmission time. The processing time is less than 1ms. This gives us a minimum delay of 74ms up to a maximum of 160ms. These values are transmitted at a relaxed speed of 3.8kbps that can be raised to 7.6kbps if required. Giving a minimum of 37ms up to 125ms per transmission but increasing the chance of error [[1]](#footnote-1). Tested at 15kbps the chance of error rises heavily due to the ISR overload

In the M16C. Said this, one may take the timestamp as the time the measure has arrived and, if required, improve it by adding a µController with a real time clock to set timestamps from it.

There is included a call to transmit\_measures() which is no more than a function to display the data on screen for debugging purposes. Uncomment it if required.

Then the thread sleeps for a determined time in miliseconds.

As for each source file that holds the code for an independent thread, an APP\_ID has to be defined for the thread. Initially is only used to control accesses to the SPI driver. In this file, one may find it defined as *MOTOR \_TASK\_PID* and refers to a unique 32 bits integer number, no other APP\_ID can have the same value. Calls to the drivers require this number.

* /src/energy\_data/energy\_meter.h:

Application unique PID

* /src/energy\_data/energy\_meter.c:

*reading\_energy()* thread.

First run opens 'bcm2835' library and SPI communications, if not opened jet, and proceed to configure the ADE7880 as an energy meter

The loop reads the required ADE registers. As this device has many different, one by one are read and not all at the same time as with the M16C communications. Processed in the same way as teh motor readings, the transmission speed is 1Mbps that can be raised up to 2Mbps. The speed has been tested and proved that one register read and processing takes less than half of a millisecond, reading a set of 5 registers from 2 to 3 milliseconds. Note that the test conditions were to add printouts of the timestamp of each reading in the code, function that, itself, adds a delay.

* /src/motor\_control/motor\_control\_srvc.h has nothing relevant
* /src/motor\_control/motor\_control\_srvc.c

*motor\_control\_srvc()* thread

This thread executes only once. It creates independent threads that call the *server\_writeData()* and *server\_readData()* callbacks and network connection to hold DEMVE transmissions by means of libIEC61850. These functions upload and read back data to DEMVE system.

Only server\_writeData() is implemented to upload the data to DEMVE, for security issues, we cannot access or control the motor in any way so despite of defined for a future use and included in the code, it remains empty.

Data is uploaded using the methods provided by the library after building the required IED description file.

* /pi\_spi\_bcm2835/pi\_spi\_bcm2835.h

Provides definitions and prototypes for public functions

* /pi\_spi\_bcm2835/pi\_spi\_bcm2835.c

Midware driver for SPI access, makes access to the SPI reentrant but non-blocking and adds extra control over it. One must use the provided functions in here and not access the driver directly in any case. This middware makes the driver fully compatible with systems that have different applications running in different threads with calls to the same device using different configurations and speeds. It could be implemented even if having more than one different SPI bus and always warrantee that one application is using it at a time with its own configuration. Read doxygen documentation for more information.

* /pi\_spi\_bcm2835/bcm2835

This folder holds the SPI bcm2835 driver, if the driver is installed in the system the access to this folder is no longer required.

Bcm2835 library driver is the result of the work from Mike McCauley for accessing the RasPi GPIO pins v1 and v2

* /pi\_m16c\_spi/definitions.h

Use it to set the CS channel in use by the M16C µController SPI (CS1).

Set the transmission speed to 0 as the safest (3.815kHz)

Read and write command, have to be the same in M16C code. Is the first byte in the MOSI line and indicates the type of operation to perform

* / pi\_m16c\_spi/m16c\_endianness.h

M16C transmission translation at byte level. Set here the required configuration (usually little, and that is the one in M16C code). The code will do the rest, including the necessary to arrange the correct byte position.

* / pi\_m16c\_spi/sensors.h

Holds the definitions for each sensor controlled by the M16C.Required to address each sensor call. Has to be the same as in M16C code

* / pi\_m16c\_spi/pi\_m16c\_spi.h. Nothing relevant.
* / pi\_m16c\_spi/pi\_m16c\_spi.c

Midware for the M16C SPI driver. Provides the access to the SPI under control, avoiding access conflicts, it not only avoids different apps collisions but different threads of the same app as well.

Use it always whenever the M16C requires access to the SPI driver.

At this moment only *PI\_M16C\_SPI\_Write* and *PI\_M16C\_SPI\_Read* are implemented but they are enough. These methods are of the same structure like those found in the *pi\_ade7880* midware

* / pi\_m16c\_spi/m16c\_spi.h

Include definitions to build the transmission or reception buffer. It is build up as a concatenate set of structures altogether with unions, providing access to the whole transmitted string or to each individual set of information.

To pack the bytes aligned together, instead of using compiler specific code, this file provides a way to control the starting point of every new set of information by setting its relative offset to the beginning of the buffer. This is a static way to build it up but it offers a higher portability.

* / pi\_m16c\_spi/m16c\_spi.c

M16C SPI driver for Raspberry Pi. Provides only readings from the M16C since motor cannot be controlled therefore not accepting writes to its settings.

The driver provides two main methods to access the peripheral, one to read a specific sensor address content and, a second one, to read all the addresses content in a row. This latest is the only one in use since it is much more efficient. To read individual sensor addresses will not only take longer, but affect slightly the quality of the measurements since the Chip Select signal triggers an interrupt, and this has a high priority in the peripheral that might, rarely but possible, affect another interrupts like the RPMs. A word about the M16C code at the end of this document if time allows.

The calls to the driver handle the transmissions buffer byte arrangement. If the byte “endianness” has been set correctly, the content will be stored in the right order.

This file provides two methods to get or set the transmissions buffer, *M16C\_set\_tx\_buffer, M16C\_get\_tx\_buffer, M16C\_set\_single\_tx\_buffer and M16C\_get\_single\_tx\_buffer.*This methods handle the byte padding, packing the whole buffer based in the defined offsets.

* /pi\_ade7880/pi\_ade7880\_config.h

Global ADE7880 configurations, SPI channel in use by the peripheral, the SPI clock speed and a delay in µseconds while waiting access to SPI driver. One may set here the seconds to yield the current task, no further use in this code since it will always get access in time.

* /pi\_ade7880/src/ade7880\_driver.h

The file that any project should include to grant the access to the driver. Read Doxygen documentation for more information.

* /pi\_ade7880/src/pi\_ade7880\_drv.h

Definitions to hook the peripheral pins to the Raspberry Pi pins and the structure definition for storing the measured data

* / pi\_ade7880/src/pi\_ade7880\_drv.c

To keep the driver portable and system independent, required to hook the driver code to the system. Provides access from the driver to methods that have direct access to the hardware (GPIO) or other system drivers, in other words, is the way the hardware (RasPi) is hidden from the ADE7880 driver.

In order to achieve a fully independent driver, the methods that are required inside it, have to be passed from the middware as callbacks, said this, a call to PI\_ADE\_driverCallbacks() is required before calling any driver’s method.

1. These values are still unconfirmed at the moment and are theoretical values. [↑](#footnote-ref-1)