# Design document AVR

The design documentation found in this document serves as an introduction. No attempt is made to provide a full reference manual. More information is provided in the source code.

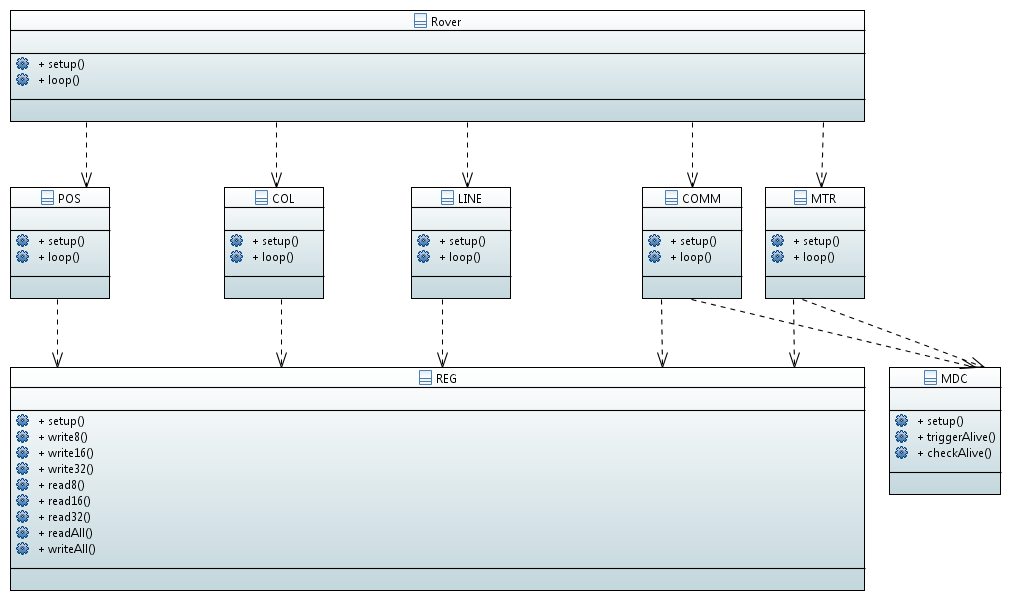
# Introduction

The program executed on the AVR has the responsibility to:

* Control the PWM outputs
* Capture the infrared sensors (collision and line)
* Machine damage (stop motors when connection with Pi terminates).

This is done by interacting periodically with the Pi via SPI.

The application is built using Arduino. This means that the architecture recognizes a ‘setup’ function that is executed during a startup and a ‘loop’ function that is executed inside a while(true) loop (as fast as possible). The application consists of a number of C modules (represented by C classes in the diagram below).



The Rover’s setup operation simply delegates to the ‘setup’ operations of the other modules. The same holds for the ‘loop’ operations.

The state of the rover is maintained in the REG module. It defines a struct that represents the state of the Rover. This struct is called the register map (holding a collection of registers). Interaction with the state is done via read/write operations. The number indicates the size of the data to be written or stored. Other modules either read or write the registers. For example, the loop operation of the MTR (motor) module will read the dutycycle and direction registers and effectuate their values on the corresponding outputs of the AVR. The POS (position sensors) module will, in it’s loop operation, inspect the current position and write it into the corresponding position registers.

The COMM(unication) module is the one that takes care of interaction with the Raspberry Pi. It checks whether the Pi intends to communicate and if so, it sends the current values of the registers to the Pi and it receives new values for registers. During this communication is blocking in the sense that no other loop operations are executed. Interrupts however are not blocked so the interrupts that decode the encoder signals are still being handled.

The MDC (machine damage) module is responsible of stopping the Rover in case interaction with the Pi gets lost. Note that it does not autonomously stop the Rover in case the Collision sensors see something.

## REG

The main responsibility of the REG module is to main the Rovers state in the so called register map. REG.h defines a struct (similar) like:

typedef struct

{

uint32\_t MICROS;

uint32\_t MILLIS;

uint8\_t LEFTDIR;

uint8\_t LEFTDC;

uint8\_t RIGHTDIR;

uint8\_t RIGHTDC;

uint8\_t COLLISION;

uint8\_t LINE;

int32\_t LEFTPOS;

int32\_t RIGHTPOS;

uint16\_t AMBOFFSET;

uint16\_t AMB\_LINE\_NE;

uint16\_t AMB\_LINE\_EN;

…

uint16\_t IR\_LINE\_SW;

uint16\_t IR\_LINE\_WS;

uint16\_t IR\_LINE\_WN;

uint16\_t IR\_LINE\_NW;

uint16\_t IR\_COL\_NE;

uint16\_t IR\_COL\_SE;

uint16\_t IR\_COL\_SW;

uint16\_t IR\_COL\_NW;

} REG\_map;

Reading and writing is done by operations that require a register ID and a value or destination. The register ID’s are #defined in REG.h as well. As reading and writing is frequently done, this operation is worthwhile optimizing. This is done by, during setup, filling an array of pointer to destination addresses in the register map. By this, reading/writing a register simply boils down to dereferencing the array with the register ID in order to get the address of the value in the map. This address is used to either read or write the corresponding register/value.

extern void REG\_setup()

{

reg\_address[REG\_MICROS] = (uint8\_t\*) &reg\_map.MICROS;

reg\_address[REG\_MILLIS] = (uint8\_t\*) &reg\_map.MILLIS;

reg\_address[REG\_LEFTDIR] = (uint8\_t\*) &reg\_map.LEFTDIR;

…

reg\_address[REG\_IR\_LINE\_WN] = (uint8\_t\*) &reg\_map.IR\_LINE\_WN;

reg\_address[REG\_IR\_LINE\_NW] = (uint8\_t\*) &reg\_map.IR\_LINE\_NW;

}

extern void REG\_write8(int id, uint8\_t val)

{

uint8\_t\* dst = (uint8\_t\*)(reg\_address[id]);

\*dst=val;

}

## LINE/COL

The responsibility of the LINE and COL modules is very similar. Both translate the input of the IR sensors into a digital signal that indicates whether they see a line or colliding object or not. In order to ensure sufficient contrast, an IR led is used to illuminate the environment. Light is reflected by a surface then is read by the IR sensors. As there is always an amount of ambient IR light, a correction is needed. This is done by first making a measurement without illumination by the IR led. Then, the IR led is turned on and another measurement is made. The first measurement represents the amount of ambient IR light. This value is subtracted from the second measurement. If the result of the subtraction exceeds a certain threshold the digitized output will be set to 1, otherwise it will be set to 0.

Besides registers holding the analog measurement result of the ambient and active readings, also a register is defined that holds a bit mask of the digitized results.

## POS

The responsibility of the POS module is to maintain the distance travelled by the Rover’s wheels. Note that due to slip, this is not equivalent to the distance the Rover travelled.

Rotation of the wheels is captured by quadrature encoding. This means that two block signals, phase shifted by 90 degrees, are provided by the Rover’s mechanics/electronics to the AVR microcontroller. It is important not to miss any state change as this leads to measurement errors. Therefore, state changes on the corresponding input pins of the microcontroller are handled by ISR’s. Each state change leads to a position increment/decrement stored in a local variable.

In the loop function, the values of the local position variables are copied to the corresponding registers in the register map.

## MTR

The responsibility of the MTR module is control the motors of the Rover. The Rover contains two motors each actuating a track. Controlling the speed of the motors results in the Rover moving forward, backwards or even making turns.

The motors are controlled by a direction signal (forward/backward) and a duty cycle of a PWM signal. The register map contains corresponding registers. The MTR’s loop function reads the values in the register map and effectuates them on the microcontroller’s outputs.

## COMM/MDC

The responsibility of the COMM modules is to interact with the Raspberry Pi. The MDC module needs to keep track of machine safety. Machine safety in this case means that the Rover is expected to stop moving as soon as interaction with the Pi terminates.

### Communication

Communication between Pi and AVR is done via SPI. The Pi is considered to be the master meaning that it is the Pi who initiates the communication. To give the AVR some time to prepare, additional signals are used to control the communication protocol. These signals are called:

* REQEXC: request to exchange (send from Pi to AVR)
* ACKEXC: acknowledge exchange request (replied by AVR to Pi)

Whenever the Pi wants to exchange the register map, the REQEXC is set HIGH. The AVR has to react on this signal by making any required preparations. As soon as it is ready to communicate, it needs to acknowledge by setting ACKEXC HIGH. The Pi then starts the SPI transfer (while keeping REQEXC high). After exchange of the full register map, the PI set REQEXC to LOW. Then it has to wait until the AVR sets ACKEXC LOW indicating completion of the transfer.

The COMM module contains two variables capable of holding a copy of the register map. One is used to hold the data being sent to the Pi; the send buffer. The other one will store the data that is received by the AVR; the receive buffer.

As soon as the Pi raises REQEXC, the AVR copies the current values of the register map into the send buffer. Then it raises ACKEXC after which the data is being exchanged. The data in the send buffer is send to the Pi and the data received from the Pi is put into the receive buffer. In order to avoid corrupting data in the register map, only registers which are meant to be written are actually stored in the register map. After this, the AVR waits until REQEXC is LOW before lowering ACKEXC.

### Safety

The MDC module has two functions that regularly need to be invoked: triggerAlive and checkAlive. The triggerAlive function stores the current time, the time that it is invoked, into a local variable. The checkAlive function subtracts this value from the current time (resulting in the amount of time that passed between invoking triggerAlive and checkAlive). If this period exceeds a certain threshold, then the output on the PWM is set to zero.

In order to avoid the MTR module to simply set a new DC, it also invokes checkAlive before setting a non zero DC on the outputs.

It is obvious that the COMM module needs to invoke triggerAlive as soon as a communication succeeds. However, inside the communication protocol two moments exist where the AVR needs to wait for the PI to react. Also in these waiting loops, checkAlive needs to be invoked in order to avoid the Rover running indefinitely while waiting for the Pi to react.

### SPIcological

The registermap is about 80 bytes large. The SPI clock signal can be set to 4 Mhz at the AVR. 4 Mhz implies a communication speed of 4 Mb/s or 0.5MB/s, or 0.5KB/ms. A register map of 80 bytes should be communicatable in 0.16ms. So far theory.

In practice, the AVR has a one place buffer to exchange the bytes to be communicated. That means that whenever one byte has been transmitted, another one needs to be put in the transmit buffer as soon as possible. Luckily, it is allowed to first write into the transmit register before reading the received data.

The PI turns out to leave only a marginal amount of time between the consecutive bytes, about two SPI clock cycles. A SPI clock of 2Mhz requires 1 us for 2 clock cycles. 1 us is about 16 clock ticks of the 16 Mhz AVR clock.

Whenever a byte is received, an interrupt service routine is executed at the AVR. While handling this interrupt:

* the next byte needs to be written
* the received byte needs to be retrieved
* the cursor pointing to the next byte to be written needs to be incremented
* the cursor pointing to the next locatino where to put received bytes needs to be incremented and
* it needs to be tested whether the full message has been received.

This can not all be done within the 16 clock ticks. The solution is not to let the PI transfer the message as one block, but to transfer the map byte after byte. After each byte has been send, the Pi waits a little while before sending the next one.

In fact, a delay of about 10us between two bytes would suffice. However, it turns out that an invocation of usleep(1) on the raspberry pi requires at least 75 us. The alternative is to use some kind of a busy waiting loop. We need to balance the load on the pi vs. an error in the communication.