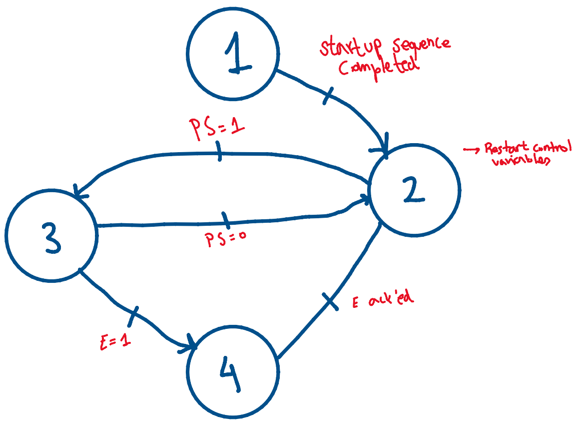
Class design SW

Tuesday, February 19, 2019 4:20 PM

SystemManager

The system manager is the component governing the system. Its goals are to read the digital user inputs, handle the user references and to allow all the components to know what status (set of tasks ans features) has the system. The status will be implemented by a FSM, as described in the diagram. Whenever the error manager detects an error, the system manager state will change to error, whenever this error is acknowledged by the user, the system will switch to standby, restarting all the control variables. Further information can be found in the diagram.



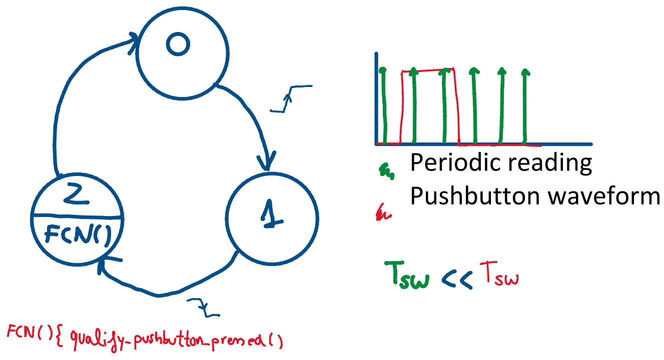
System manager FSM state	systemState

Allow global read of system state	readSystemState()
System manager main function including FSM	manageSystem()

States: 1. Start-up 2. Stand-by 3. Running 4. Error	Startup sequence is performing all the necessary initializations PS = Power switch E = Error detected E_ACK'ed = Error acknowledged by some kind of button or human interaction

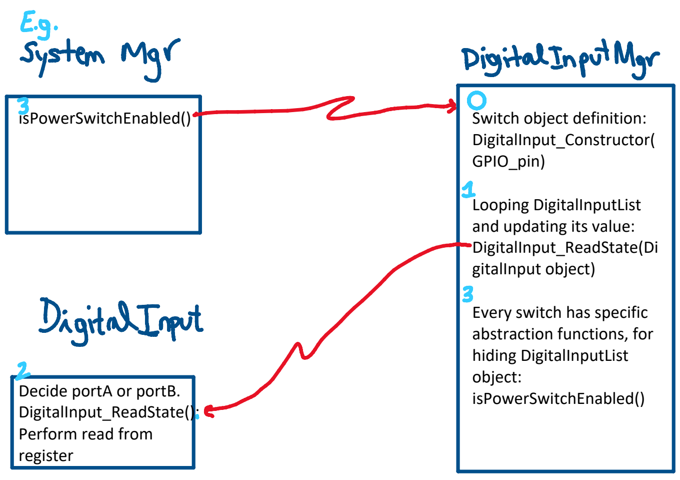
pushButtonManager

This module is an abstraction layer higher than digitalInputMgr. Its goal is to qualify whether a pushbutton is on or not. It is based on a FSM which will qualify a 'pressed' event whenever a pushbutton is pushed AND released. The pressed event will hold a '1' value until an external software component reads it. At that moment, the value will be reset to 0. This implies risks in case that two software components use the same button, but that will never be allowed.



digitalInputManager

This module provides the interface for knowing the state of diferent digital inputs.
The object DigitalInputList is a struct with all the digital inputs. Every digital input is made by a digitalInput object, which has two attributes, the GPIO pin(e.g. GPIO 34) and the pin status(HIGH or LOW).



Initialize GPIO as Input.	configureGPIO
Create the objects, specifying the pin. This function calls DigitalInput_Constructor().	initInputs()
Read using polling method digitalInput array	readDigitalInputs()
Create higher abstraction functions linking the state of a specific input to its function. There will be as many functions as buttons.	E.g. isPowerSwitchEnabled()

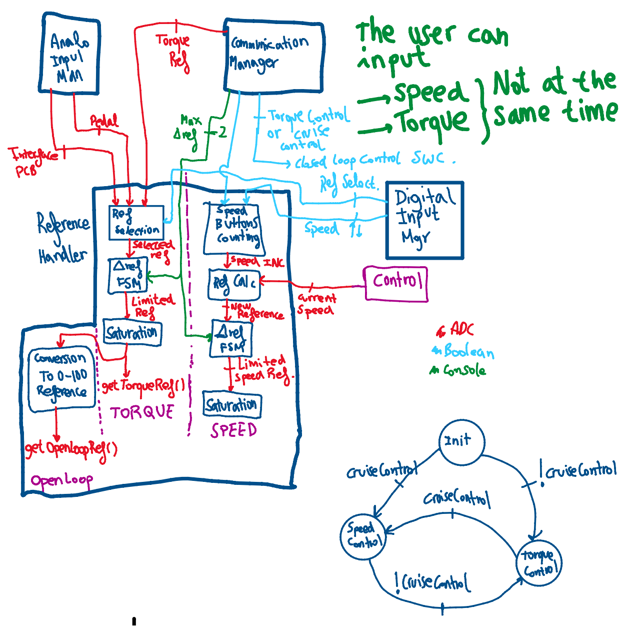
digitalInput

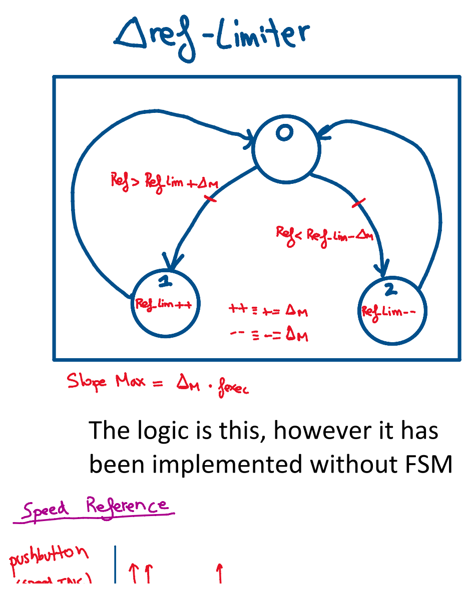
PIN
Value

Allow the setting of the pin	DigitalInput_Constructor()
Read value	DigitalInput_ReadState()
Detect whether the pin is in port A or port B	pinIsInPortA() pinIsInPortB()

referenceHandler

The reference handler must check what reference source is selected by the user. When the source is known, the value of that reference must be obtained. This value must be provided to the control algorithm by means of an interface (getTorqueReference() or getSpeedReference()).
It is not desired to have large reference changes, then a maximum slope is set by the user and cann be changed from the graphical user interface (To be implemented in GUI).
In order to improve system's safety the reference source change will not directly change the reference source but will drive systemState to STANDBY, then it will wait until the rotor speed is 0 and will start over with the new reference source.





Derivative of allowed reference	referenceSlope
Maximum value allowed for reference	maxReference
Actual value of the reference to be fed Into the control object	reference

Analyse the state of SW1 and GUI to determine whether the Reference should be obtained from	getReferenceSource()

Analyse the state of SW1 and GUI to determine whether the Reference should be obtained from Pedal, GUI or interface PCB.	getReferenceSource()
Change the slope's value	setSlope()
Interface for reading reference	getTorqueReference() or getSpeedReference()
Perform the calculation of the reference To be fed into the control object (FSM?). Include limiting to maxReference.	calculateReference()

controlTask

The control task is a component that performs the periodic execution of all the features included in the control pipeline. This starts by obtaining the ADC variables and finishes by setting the corresponding duty cycles.
It also must be considered the error mode, where the duty cycles are driven to 0% and all the control variables are restarted. The control variables restart function will be called by the system manager component whenever the system is in STANDBY state.

- The whole control pipeline is not called by the system manager but by system's scheduler.
- See Lajos email from 22/02 on scheduling frequency

Perform ADC measurements	acquireAnalogSignals()
Decide whether it is desired to run in closed or open loop and execute the proper control algorithm	executeControl()
If system manager state is Error, the system must be returned to safe state: duty cycles = 0. And executeControl() should not be called. This disableDutyCycles() is directly called from the system manager.	disableDutyCycles()
Monitor SW3 to activate open loop or closed loop control	isOpenLoopControlSelected() isClosedLoopControlSelected()

analogAcqMgr

The ADC signals to be measured are defined in here --> The struct of every signal is created here.
The ADC must take as few time as possible to be performed. Some ways of achieving this is DMA or interruption based sampling, explained in example 'adc_soc'. The goal is to trigger all the sampling at the same time (simultaneous sampling).
Maybe use CLA for filtering with rocket performance, example cla_adc or cla_adc_fir.
The module is called inside the controlScheduler. The task will be executed periodically.

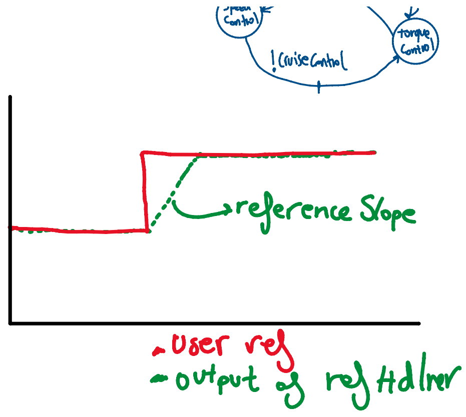
Array of all the analog signals to monitor	signalList

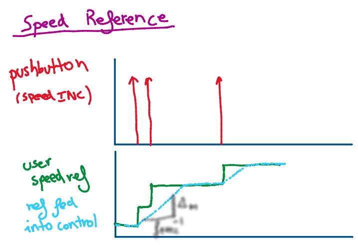
Initialize ADC, calculate filter parameters	initADC()
Perform read of all desired ADCs DMA?	readSignals()
Perform the filtering calculations with the newly obtained values	filterSignals()
Creating all analog signals from the AnalogSignalList	configureSignals()
Link the analog signals to a specific ADC channel.	configureADCs()

Signal

Type of filter HPF LPF	type
Order of the filter	order
Parameter a of filter	filterParamA
Parameter b of filter	filterParamB
Value after filtering	filteredValue
Filter cut-off frequency	cutoffFreq
Last ADC read	lastObtainedValue
Read value from ADC lag 1. Used for filtering signal.	oldValue
ADC channel where the actual signal is mapped	adcChannel
Every ADC signal has a threshold in which the signal is expected to be under normal conditions. If the signal is contained outside this threshold, the AdcMonitor from ErrorManager will trigger an error. It consists of an array of two values, minimum and maximum.	threshold

Object constructor	signal()
Calculate filter parameters	filterParameters()



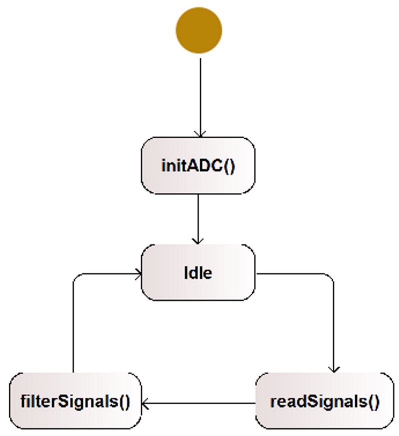


The ADC measurements are made in the interrupt, every time new measurements are available. So this is not necessary, you just have to take the values from the struct when you need them.

I am imagining a switch case of FSM changing with isClosedLoopControlSelected(). And then call runClosedLoop/calculatedDutyCycle

I guess this should look at some error flag and then just shut down. Maybe consider toggle the enable signal instead.

I would do this in one method and then return true or false depending the switch position.



initADC() calls all methods needed for configureing the analog signals and ADCs. This includes:

InitADC()	Standard method
ADCSelfCalibrate()	Standard method
configureSignals()	
configureADCs()	

readSignals() reads digital value of all ADC channels initialized.

filterSignals() calculates the voltage and performs the filtering of ditto.

Set filteredValue	setValue()
Read filteredValue	readValue()

openLoopControlManager

This module has two main goals. It can set the duties to a constant value according to a reference input or it can also implement VF control according to the reference.
The function setConstantDutyFromReference() sets a constant duty cycle in all the legs.
The function openLoopVFControl() sets constant frequency sine waves in the three legs. This sines have a phase difference of 120º between one another. Also, it adjusts the amplitud of the waves according to VF control.
The VF control function should be called in every switching cycle to update the signal accordingly.

Constant duty cycles on all legs.	setConstantDutyFromReference()
Generate VF control at referenced freq.	openLoopVFControl()

closedLoopControlManager

This software component must perform an organized call of all control functions used in the closed loop control and also the signals measurement.	
A function for restarting control variables must be provided. This function will be called by the system manager.	restartControl Variables()

Abc2dq and dq2abc

This component runs the closed loop control algorithm. Abc2dq and dq2abc This modules perform the calculation for changing the reference frame.
dqSignal object is used.
Consider CLA.

controller

The controller is the component performing the reference calculation. Its input are the reference and the measured variables and the otput is the voltage referene that will be fed into the SV modulator.
There are two main tasks, calculating the current reference from torque and flux references and calculate voltage reference from current references.
The controller uses the PI object. The input to the PI object is an array of errors with the current and previous values of error. The length of the error array must be tuned. The error array calculation is performed in this module.
Consider CLA.

Calculate current references implementing 'UFO module controller' module.	calculateCurrentReference()
Calculate error, keep it using appendError() and finally calculate the reference using the PI object.	calculateVoltageReference()

PI object

The PI object will be the frame for implementing the PI controller. It has proportional and integral parameters. For the integral component calculation, the previous value of error is necessary. The input must be an error vector with previous values (at least one).
Consider CLA.

$$Output = KP \cdot e + KI \cdot \sum_{n=0}^{\infty} e(n)$$

Proportional parameter	KP
Integral parameter	KI

Calculate output	PiCalculation()

epsilonArray

As many control errors are calculated in the system (those will later be fed into the PI controllers) and as we are using integral action, an array (history) of errors is used. This array of errors changes every control execution cycle. To unify all this tasks and features an error object is created. It will be important to have the append() function in order to add the last calculated error to the array. In order to achieve that, a double linked list object will be used. Info can be found at https://www.learn-c.org/en/Linked_lists
The error array object will be directly feed into the PI object.
As in the system there are error objects and an error array for monitoring and safety purposes, instead of naming the difference between the reference andd the measured value 'error' it is named 'epsilon'.

Allow appending error values to the array.	appendEpsilon()

SVModulator

The input is a voltage reference and the output is 3 duty cycles duty cycles, not boolean but a value between 0 and 100. This value is then fed into dutyCycle object for setting the appropriate duty cycle signal.
The calculation of the duty cycles is a big task, literature for performing it mighht be found in SemesterProjectPED2\Software\Documents\SVM.

Consider CLA.

All 3 duty cycle values are calculated from the voltage reference.	calculateDutyCycles()
Set the calculated duty cycle values to the duty cycle object using setDutyCycleX()	setDutyCycles()

dutyCycle

The dutyCycle module creates the PWM signal when a reference duty is set. It can also output the current value of the duty cycle.
At initialization the setup of the PWM in done in the file PWMconfig.c.
The counter register (TBCTR) increases its value at the dsp clock frequency (90MHz) and when it's value equals the counter compare register (CMPA) the output changes it's value. This means that the precision of the PWM signal decreases linearly with the switching frequency.
The carrier selected has been a symetrical triangular wave to adjust to the control design.
During operation, function setDutyX(duty) will set the duty of the PWM signal in the selected branch. Another function setAllDuties(duty) will update all three branches at the same time.
Also, the function readDutyX() will output the current value of the duty.

Set duty cycle value	setDutyX(duty)
Set all duties to same ref	setAllDuties(duty)
Read duty cycle value	float = readDutyX()

Internally, the file manages two object types.

DutyCycle	duyValue	float with ref value
	dutyCompare	Uint16 including value to input in the compare register
DutyCycleList	legA	DutyCycle struct corresponding to leg A
	legB	DutyCycle struct corresponding to leg B
	legC	DutyCycle struct corresponding to leg C
	minDutyValue	Minimum value of input duty (float = 0)
	maxDutyValue	Maximum value of input duty (float = 100)
	minDutyCompare	Minimum value of comparator (Uint16 = 0)
	maxDutyCompare	Maximum value of comparator (Uint16 = CLKfreq/(2*SWfreq))

positionCalculator

The position calculator is the software component in charge of computing the flux angle. The way is to add the rotor position, coming from the encoder, to an estimation of the angle difference produced by slip. (To be confirmed that is it like that).

Compute the position of the magnetic flux	computeFluxAngle()
Create an interface to allow the reference frame transformation modules reading the flux position	readAnglePosition()

encoderManager

The system must be able to measure absolute (3 signal encoder) position, speed of the rotor from an encoder. Check eqep_pos_speed example.
QEPA, QEPB and eQEPI inputs are used.
Read: AA Enhanced Quadrature Encoder Pulse (eQEP) Module Reference Guide.pdf
The encoder is not called by any module but is periodically and automatically run.
Consider Encoder internal watchdog.
The angle found from the encoder is the rotor mechanical angle, but I think that the control needs the stator electrical angle, then the slip frequency and number of poles must be considered. Check with Stef how to find the stator angle.

Maybe compute and read functions are simple enough to mix them into one.

Initialize eqep	initializeEncoder()
Compute position	computePosition()
Compute speed	computeSpeed()
Make variables publicly available to read	readSpeed() readPosition()

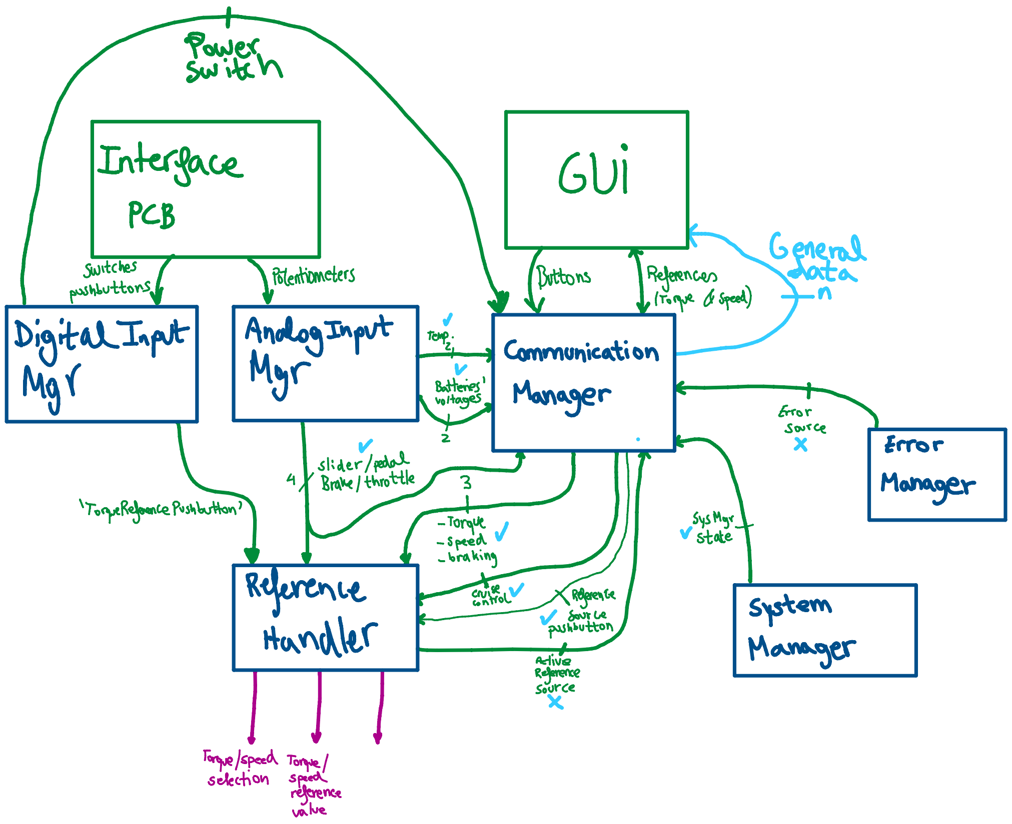
slipAngleCalculator

The goal of this component is to calculate the flux angle coming from integrating slip frequency. The control name of this feature is CFO.

Read the angle difference produced by slip	readSlipPosition()
Compute the angle difference produced by slip How often should that be triggered What inputs does it need?	TBD

communicationManager

The goal is to create an interface between the DSP and the user. Information like speed, references or temperatures are shown. It also has the task of supporting the debugging of the code, as the user is able to easily see the state of many variables at the same time. The interface will be developed using GUI Composer, which is a TI's sotware for the intended use.
The GUI Composer project was created following https://dev.ti.com/gc/designer/help/UsersGuide/index.html and https://dev.ti.com/gc/designer/help/Tutorials/GettingStarted/index.html. However following that steps wasn't enough as the GUI Commposer variables wouldn't be linked to the actual code variables. Finally, a project from the Gallery (A place where people share their projects) was imported and then by configuring that project with our board settings, it started working. A TI's employee answer: https://e2e.ti.com/support/tools/ccs/f/81/p/599945/2216781#2216781 was useful for general guidelines. It's stated that it's convenient that the variables are volatile and the variables must be global.
XDS connection is used.
According to testing, the update period shouldn't be shorter than 0.25 s.
This constraints shape the communication manager. In our system, global variables are avoided in order to achieve a more robust and safer code. Then the communication manager will have defined all the global variables that are to be viewed in the interface. For acquiring the values, the interfaces provided by every submodule will be used.
Another concern is how often will the data be updated. As the user can only get a limited amount of information for amount of time, the data shouldn't be updated at frequency f >> 1 Hz. The limitation of GUI composer using XDS communication appears to be around 4 Hz.



The GUI composer showed a connection error in the verifying connection step: 'Hardware not connected. Communication with Target Failed:no response' The solution was to create a new project. In order to copy your previous work, you can create a template with the previous and when creating the new project, use the template.

Structure containing variables from GUI to be read by other SWC	GUIAnalogSignals

Allow the read of the structure from other SWC	getGUIAnalogSignals()
Function periodically called by scheduler which collects the variables to be shown in GUI from other SWC	manageCommunications()

errorManager

The error manager is the module caring about system's safety. Every submodule oversees specific features.
The mantra in this module will be to call downstream functions expecting a boolean return. Functions' name will not be misleading on the used logic (direct or reverse).
The flags for error triggering will be of type error, which will be an enum.
The errorManager must be periodically scheduled. overcurrentMonitor performs discrete integration: be careful with scheduling times.
Error manager must react whenever an error happens. The reaction consists on driving the enable pin of the drivers to off, switch on an error LED, logging the error source and the timestamp and finally letting the system manager know that it should switch to system error, waiting for user acknowledgement.

Call downstream functions in order to check if there are errors	monitorErrorSources()
Group all safety reactions(described in next functions)	performSafetyReaction()
Switch on the appropriate LEDs, using digitalOutput object	turnOnErrorLED()
Driver enable pins (3) to off, this will be performed by directly controlling the digital output, using the digitalOutput object.	disableDriver()
The system must log the errors in order to let the user know what is the chronological order of events.	
Let system manager know there has been an error. This will be implemented by creating a function that the system manager will call, this function will return an error object. This error object will be the logic OR of the error manager error object array. Every element of the array will contain the status of a safety monitor. This single error object will be accessed by polling. The reaction of system manager is to go to system error state.	getErrorStatus()

digitalOutputManager

The digital output manager is used to keep track of all the digital outputs. This includes a struct with every digital output.
NOTE: For now only static outputs have been created, need do do the jumbers as well. When a new output is created remember to configure it in GPIOConfig.h.

Void initDigitalOutputs(void)	This function calls digitalOutput_Constructor() on every digital output that needs to be initialized.
Void setxxxLED(state)	The set functions are used to interface with the digital outputs. There will be one for every output. The

	argument is: digitalOutputStatus[OFF, ON].
Int getxxxState(void)	The get functions are used to get the state of the specific output. The return will be OFF=0 or ON=1.

digitalOutput

Used for driving LEDs among others.

Pin where the digital output is mapped	pin
Logic value of the digital output, an enum must be created digitalOutputStatus [OFF ON]	state

Void digitalOutput_Constructor(digitalOutput *, pin)	This constructor is used to initialize the digital outputs. The arguments are the spcific output and the desired GPIO number
Void setdigitalOutput(DigitalOutput *, state)	This function checks if the state are set to be high or low, and then calls the necessary function
Void setDigitalOutputHigh(digitalOutput *)	This function sets the desired output high
Void setDigitalOutputLow(digitalOutput *)	This function sets the desired output Low

Error

Error type, it consists of an enum with states, in UML is called errorEnum.

0	ERROR
1	NO_ERROR
2	IDLE

batteryMonitor

The battery monitor checks whether the battery voltage is within an appropriate threshold. The main concern is to not let the battery reach the undervoltage threshold and if that is the case: stop the system.

Check whether batteries' voltages are within threshold	areBatteriesOK()

Battery

The battery object is used to instance every actual battery. In our case we have two batterys, each one will have an undervoltage threshold associated.

Boolean function that returns true if the voltage is appropriate	isVoltageWithinThreshold()

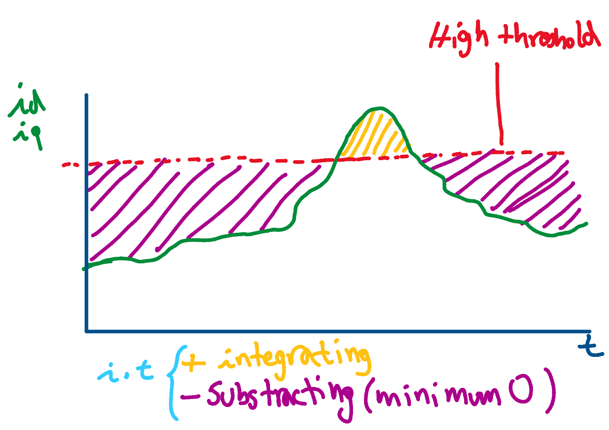
watchdogTimer

The WDT is a safety feature that restarts the MCU if an acknowledgement (ACK) is not performed within a time span. To be thought during development. The WDT might have to be an independent block to the errorManager.

Initialize watchdog timer	Done during system init
restartWatchdogTimer	TBD

overcurrentMonitor

In the case of a short circuit event, the current might damage the system. However, current over rated condition will overheat the ssytem. Then, the current must be constantly monitored in order to switch the system off if necessary. An idea of implementation might be to have a maximum i*t value and if that value is exceeded then trigger the overcurrent protection.
The current will be integrated starting from the current signal high threshold.



Consider if the calculation must be done over dq, rms, … Depending upon the decision a phase current object might have to be created.

i*t parameter to account for overcurrent in the system	chargeThreshold

Monitor whether the charge is below threshold	isChargeWithinThreshold()

controlPipelineMonitor

The control algorithm consists on several components and if any of these components has a wrong behaviour or output, the system behaviour might not be expected, then, the control pipeline will be monitored to make sure that the control is kept all the time. If that is not the case, error reaction must be performed.
TBD

AdcMonitor

In the case that a sensor or its connector is broken, the proper behaviour of the system might be in risk. Thus, the read of the ADC signals must be monitored. Also in the case of a current higher than what the hall sensor is able to measure the control should switch off the transistors.
Every signal has a minimum and maximum threshold signal.threshold[2], if the signal is outside of this threshold, the error should be triggered. Signals thresholds must be set!!

Check whether the ADC are within thresholds	areAdcReadsWithinThreshold()

If any ADC is outside threshold, an event with that information should be created.

scheduler

The scheduler is the software component in charge of calling every software component periodically.
This module will call every task when necessary. The tasks will have two possible states:
-READY: the task is ready to run and waiting for the scheduler to call it
-INACTIVE: the task has run and has been set to inactive.
At some point another part of the code might change its state to ready again, for example a timer.
The main object of the scheduler is the taskList, which is an array of TaskControlBlock(TCB) including all the tasks that the system runs. In further development, different priorities or more advanced functions might be added. The scheduler will loop around taskList and whenever a READY task is found, that function will be called. Otherwise it will loop infinitely.

Define a task object where all the tasks are listed	taskList[]

Initialize scheduler timers	InitCpuTimers(); (systemInit.c)
Loop infinitely the task list calling those which are ready to run.	scheduleTasks()
In this module the infinite loop of the system should be placed	scheduleTasks()
Initialize the taskList with all existing tasks	taskListInitialization()
Loop the task list checking whether they should be run (because the tasks must be ran periodically at whatever frequency they are defined to in taskList)	updateTasksState()