6 Embedded Architectures

More and more embedded architectures try to separate application software from basic software. This facilitates reuse and allows an easier use of code generation tools like Matlab Simulink. A prominent example of this separation concept is the Autosar RTE.

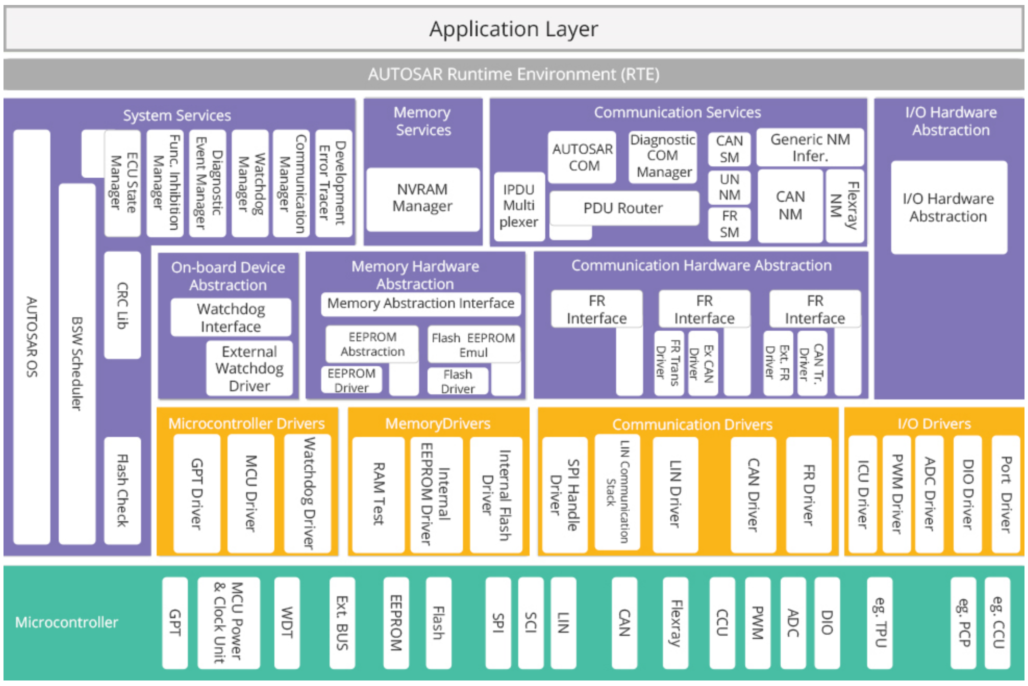

Figure 78 - Autosar RTE[7]

[7] https://www.mobiliya.com/industries/automotive/autosar

6.1 A light version of the Autosar RTE - Electronic Gaspedal

Effort: 8h	Category - B
RTE, Autosar, Software Components, Runnables, Activation Concepts	

In this exercise you will develop an ECU following the Autosar RTE concept. In the concept, the applications software and basic software (drivers and OS) are separated by an intermediate layer called RTE, which provides data containers for a message based communication between runnables and manages events (cyclic and data), which will activate the various runnables. The runnables represent the user code. They only communicate via RTE signal objects, direct hardware driver and OS calls are not allowed. This separation facilitates re-use over projects and controller boundaries.

The picture below illustrates the basic functionality of the system. Please note, that the focus of this exercise is the RTE, therefore the runnables are kept very simple.

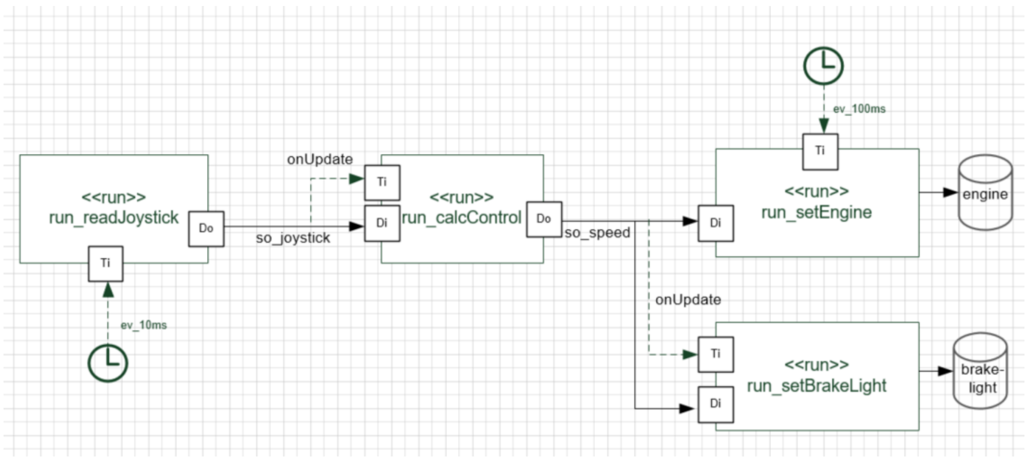

Figure 79 - Electronic Gaspedal ECU

the runnable `run_readJoystick` will be called every 10ms. This runnable will update the `joystick` signal, using the API `pullPort()`. As this signal is realized as asynchronous signal (details see below), an event will be fired upon update.

This event will trigger the runnable `run_calcControl`. This runnable will check the value of the `joystick` signal (datatype `sint8_t`). If the value is bigger than 0, the `speed` signal (uint8_t) will be set to 2 x the `joystick` value. If it is below or equal to 0, the `engine` value will be 0.

Once this value is updated, the `run_setBrakeLight` will be triggered.

`run_setEngine` will be called every 100ms and copy the `speed` signal to the `engine` signal (if it is not too old) and call the driver (`pushPort`). In case of a too old speed signal, the value 0 shall be written to the engine.

`run_setBrakeLight` will also check the `speed` value when it is updated. In case the `speed` value is 0, the signal `brakeLight` will be turned on, in case of a value bigger than 0 it will be turned off. The `brakeLight` signal then is also send to the hardware.

6.1.1 Iteration 1 - Configuration of the RTE

Download the RTE Generator from the Moodle page. You need Python to run it - https://www.python.org/downloads/ . Please also check the help index.html file provided in the package.

Use the Python Editor to create the required artifacts. Please leave the naming conventions as default. Select the output folders as needed. It is recommended to store a design folder next to the asw, bsw and rte folders (as shown in the lecture)

Based on the model description above, enter the required data in the tool. Once you are done, run the code generator. You will have to localize the generator python script the first time you run it. Please check the help file for this.

Go to the defined `\out` folder and analyze the generated files! Check the provided comments to understand the task of every file.

6.1.2 Iteration 2 - Add the RTE to your project

Create a new Erika project and add a new folder `\rte` to your source folder. Copy the generated files to this folder.

Create an Erika project having the following resources:

- tsk_Init()
- tsk_Background()
- Application tasks as required
    - you can either create a single cyclic and event task for all runnables. In this case you omit the risk for races, but long running / low priority runnables cannot be interrupted. In this case, you need to create a manual tick evenbt for the tasks – check the details in the RTE Generator help
    - or you create a separate cyclic and event driven task (simple, but not really good from the functional structure)
    - or you create an io and a control task (better functional structure)
    - and for the age, you will need an additional system task for the age incrementation
- Add Events as defined in the model
- Add Alarms as defined in the model
    - In case you have more than one cyclic trigger, you can also create a single alarm which is firing a callback. Inside the callback you manually activate the tasks / fire the events as needed.

- The will allow you to use less (expensive) OS ressources.

6.1.3 Iteration 3 - Getting it compilable

Complete the declaration in the `*_type.h` files and add some skinny sheep code to the corresponding source files.

6.1.4 Iteration 4 - Getting it running

In the code for the `joystick` driver, read the `joystick ADC value` using the driver implemented in exercise 2.1. I.e. the middle position represents the value 0, the right and upper point the value 127 and the lower and left point the value -128. Alternatively, you can also use the ADC driver created by PSOC creator.

The `pwm` driver should write the engine value to the RGB LED, using the green channel.

The `gpio` driver should write a ON signal to the red LED in case the `so_brakeLight` is on (TRUE), or an OFF signal in case the signal is off (FALSE).

Now let's add the code to the runnables as described in the initial chapter of this exercise.

It is recommended to start with the joystick runnable and to use e.g. the `UART_LOG port` to create some verbosity.

6.1.5 Iteration 5 – Error Handling

You might have noticed that the RTE offers a variety of error signaling options which can and should be used to detect different error conditions.

- The driver commands `pushPort` and `pullPort` return a possible error code of the driver.

In addition to this driver error code, the signal itself contain explicit and implicit error codes

- The signal status identifies any error condition identified in the RTE, e.g. an error code returned from a driver – this allows the application to verify that valid data is being used.
- The signal age, which measures the time since the last valid update. It is up to the application to decide which age still is acceptable. For a short period, it might e.g. ok to use old data, if you can expect, that the next update will work again.

Besides the data, you should also supervise the activation of runnables – please check the next exercise for this.

Describe a concept for handling the different error types:

Error type	Possible cause	How would you handle this?
Driver	- Short circuit - Broken link/ connection - Hardware failure - Wrong configuration - Wrong scaling - Wrong Protocol	- Redundant drivers - Diagnostic circuits - Software calibration
Signal Status	- Erroneous values from driver - No valid value for the signal (from software)	- Range check - Status check macros - Alive/ Deadline monitoring
Signal Age	- Improper scheduling of tasks - Starvation by other tasks - Error in signal status	- System task of higher priority to calculate age accurately. - Cease activation of related runnables - Monitoring

6.1.6 Iteration 6 - Extensions (optional)

The RTE has some limitations, which may be addressed if you want to dive a bit deeper into this topic.

More signals

Instead of only using the joystick to differentiate between driving and braking, you might want to add a brakePedal signal object, which is set by an ISR in case a button is pressed.

Critical Sections

Identify critical section and update the RTE configuration as needed. Update the `rte_types.h` file as required (check the comments).

Display task

You might want to use the TFT display to show some data of the process. As this is a rather long running process, the display runnable should be executed in an own, low priority context.

6.2 Timing Supervision

Effort: 4h	Category - B
Watchdog, hardware manuals, startup, alive monitoring, deadline monitoring	

Your task is to implement a timeout supervision concept for the PSOC. In case the system comes to a halt, e.g. caused by an endless loop or shutdown of the OS, a reboot should be initiated. This concept is called alive monitoring. As this is a pretty brute force error handling, an additional deadline monitoring shall be implemented, which supervises individual runnables.

6.2.1 Hardware Watchdog Driver

Req-Id	Description
NFR1	The system will be developed based on Erika OS.
NFR2	Use existing functions of the system as far a sensible and possible.
NFR3	Info: The watchdog is an architectural (fixed) functionality of the Cortex. I.e. there is no component available, instead you should check the architectural manual and created startup code.
FR4	Create a new software driver called watchdog which provides the functions described below.

FR5	`/**`
	` * Activate the Watchdog Trigger`
	` * \param WDT_TimeOut_t timeout     - [IN] Timeout Period`
	` * @return RC_SUCCESS`
	` */`
	`RC_t WD_Start(WDT_TimeOut_t timeout);`

FR6	Define a sensible enum value for the timing period, based on the provided hardware functionality.

FR7	`/**`
	` * Service the Watchdog Trigger`
	` * @return RC_SUCCESS`
	` */`
	`RC_t WD_Trigger();`

FR8	`/**`
	` * Checks the watchdog bit`
	` * @return TRUE if watchdog reset bit was set`
	` */`
	`boolean_t WD_CheckResetBit();`

FR9	Develop a first test framework, fulfilling the following requirements:
FR10	Initialise the watchdog trigger with the longest period.

FR11	Trigger the watchdog in the background task
FR12	By receiving an event (button or uart), call the OS shutdown function
FR13	Upon startup, show (via the UART_LOG), if the system was rebooted after a power on reset (POR) or after a watchdog reset.

6.2.2 Alive Watchdog

Req-Id	Description
NFR1	The system will be developed based on Erika OS.
NFR2	Use the driver implemented in the first exercise and integrate it into the code of the Electronic Gaspedal Exercise.
FR3	Add a global bitfield to the driver which is initiated with the value {0}
FR4	Add a function WD_Alive(uint8_t myBitPosition) to the driver, which sets the bit at the corresponding position.
FR5	This function shall be called by every runnable using a uniqe position, i.e. Runnable_0 sets bit at position 0, Runnable_1 sets bit at position 1 and so on.
FR6	In the background task, the WD_Trigger() function will be called if all bits are set, i.e. all runnables reported their alive status. Furthermore, all bits are cleared.
FR4	Add this concept to your system and implement testcases to verify the functionality.

6.2.3 Deadline Monitoring (optional)

Alive monitoring typically serves as a last line of defense, as the system can only react with a rather harsh reaction like reset.

In order to control the timing a bit less harsh, deadline monitoring can be applied, by checking the runtime of a single runnable or group of runnables.

The concept would be as follows:

- Before calling the runnables in a task, an alarm will be started.
- During normal operation, all runnables will be finished before the alarm elapses and the alarm can be cancelled.

- If one or more runnables take too long, the alarm will be fired and an error handling task will be activated, which e.g. can disable certain less critical runnables.

Try to implement this concept and supervise the runtime of the cyclic and event triggered runnables of the exercise Electronic Gaspedal.