POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

A Model-Based Design Embedded Software Development Methodology for an OSEK-Compliant RTOS

Relatore:
Prof. Massimo Violante

Candidato:
Filippo Cottone

Luglio 2019

Acknowledgments

To my family, that supported me in hardest periods of my life like last year. To my lovely girlfriend, that believes in me day by day, dreaming about a better future. To my crew, which without that the university years would have been less fun.

In the last years, the automotive sector has increasingly focused on control electronics: it is just sufficient to think that a common car has hundreds of MCU-based Embedded Systems, suitable for obtaining a more reliable, precise and avant-garde finished product. This has led to an increase of complexity in development processes. Due to the continuous demand of higher quality and limitations on time to market, the concept of Model-Based Software Design has started to be more and more used. In fields where a missed deadline could harm the user, Real-Time Operating Systems are a mandatory choice.

Based on results of process analysis aimed to understand how to reduce the time spent in software development, French and German automotive companies created the OSEK/DVX consortia, giving life to a standardization of the Embedded software components that required the biggest effort during development process.

Basing of these concepts, the Embedded Software Development Methodology proposed in this Thesis aims to futher accelerate the software development process and standardize it by following rules and metrics imposed by the methodology itself to avoid the introduction of bugs due to human errors by limiting the developer to just develop the Board Support Package (if not provided), its relative Simulink library and the Simulink model. The latter will represent the Task-level application layer, that will be translated in custom OS code thanks to the developed Simulink System Target File, dealing with Inter-Process Communications, Priorities, scheduling and Tasks executions. A large variety of cases can be described by modeling software at the Task-application level.

Furthermore, dealing with a OSEK-Compliant Real-Time Operating System, the whole OS configurations are obtainable from the same model, providing an Operating System code generation process to include the necessary resources needed for the modeled situation, by obtaining the minimum OS memory footprint.

Table of contents

Chapter 1

Theoretical Background

1.1 Real-Time System and RTOS

In many embedded systems application fields *timing* is considered the most important and critical factor: in these scenarios, to execute a certain routine after some additional delay may cause unexpected events or even worse may harm the user and/or the people around it.

Let's define a **Real-Time System** as a system that, by receiving some data and processing them to compute some results, affects the environment sufficiently "in time" by respecting an imposed deadline [1]. Real-Time Systems are usually used for Safety-Critical or Mission-Critical applications, such as automotive control components, medical devices or nuclear systems. Systems of this type must guarantee a response to a certain event within a certain time frame: if the response is given later, the Real-Time processing fails.

Caring about timing-constrained execution possible issues, handling Real-Time Systems that have to execute different routines (let's call them **Tasks**) by respecting different deadlines and ensuring data integrity between them can be very challenging in complex systems; in order to deal with these issues, many Real-Time Operating Systems were created during the years.

A **Real-Time Operating System** or **RTOS** is defined as a *"a background program which controls and schedules executions and communications of multiple time constrained tasks, schedules resource sharing, and distributes the concerns among tasks"*[1].

In order to better understand how to a RTOS works and to have a clear idea on what it has to deal with, it is needed to consider the following definitions:

- **Preemption**: operation through which a Task is temporarily removed from the *running* state and moves to the *waiting* state, due to the necessity of the CPU to execute another Task or routines;

- **Scheduling**: Operating System component that implements a scheduling algorithm starting from the Task states, their priorities and the available resources. The scheduling algorithm will take care of choosing a Task to be executed from the list of ready Tasks depending on the algorithm metrics. Scheduler helps to improve and optimizing the CPU utilization by organizing the Task executions and exploiting the Task waiting states by executing other ones from the ready queue.

  A scheduling algorithm is said to be **preemptive** if preemption is allowed, due to different kind of necessities, as for example in dealing with Tasks with different priorities. Preemptive scheduling algorithms are more complex to implement, due to the CPU context swtich mechanism to move from a Task execution to another one.

  Non-preemptive scheduling algorithms do not allow a CPU context switch from a Task to another as preemption, so they are easier to implement.

  Real-Time systems need to ensure that the Task deadlines are respected, in order to avoid failures that may lead to harmful situations for the system user. Starting from this concept and understanding what defined before, many different scheduling algorithm can be implemented, depending on the situation and how much critical is the application.

1.2 OSEK/VDX Standards

During the years, the quantity of microcontrollers used in the automotive industry has grown very fast: nowadays, the most common cars have up to hundreds of Real-Time Systems based on such components, due to the high quality requirements, more secure and efficient vehicles specifications.

A race to higher quality Electronic-based vehicles began. In order to improve the Software development productivity and as consequence to speed up this process without affecting the software quality neither the Real-Time specifications, an analysis based in Germany and France highlighted the most effort-required points during the project roadmap. What was discovered was that the major part of the effort was spent in the development and debug phase of the Operating System, I/O interfaces and Network Communications.

From this analysis, two consortia started to grow: the french car manufacturers made the so called **VDX**, that stay for *Vehicle Distributed eXecutive*; in the meanwhile, the german autovehicle houses created the **OSEK**, that comes from the german acronym "*Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug*", which means "*Open Systems and their Corresponding Interfaces for Automotive Controllers*".

As defined in [2], "*OSEK is a standards body that has produced specifications for an embedded operating system, a communications stack, and a network management protocol for automotive embedded systems. It has also produced other related specifications. OSEK was designed to provide a standard software architecture for the various electronic control units (ECUs) throughout a car*".

So these consortia started growing basing their ideas on creating standard development processes. Later on, they merged together to give life to the OSEK/VDX open standards.

The main goal was to develop a standard API to reduce the amount of effort and increase the code reutilization within the vehicle systems. As result, the ISO 17356 standard was born. Many principal parts are included OSEK standard:

- *Operating System* (**OS**): included in ISO 17356-1:2005, it expresses the concept of multitasking automotive Real-Time Operating System and introduces the OS APIs; moreover, the OS specifications model an environment in which

resources can be efficiently used for automotive control unit application software. The OS is meant as a single processor operating system for distributed ECUs [3].

- *Communication* (**COM**): defines communication interfaces for internal ECU software modules communications and for external nodes, aimed to improve their portability [4];

- *Network Management* (**NM**): a set of node-monitoring services are defined; they include specifications for the internal interfaces between the internal components, the algorithm to access the low power **sleep mode**, the interface to interact with the APIs and the procedures for node monitoring [5];

Even if they were developed starting from automotive purposes, the specifications can easily meet the requirements of a small system with interprocessor communications [6].

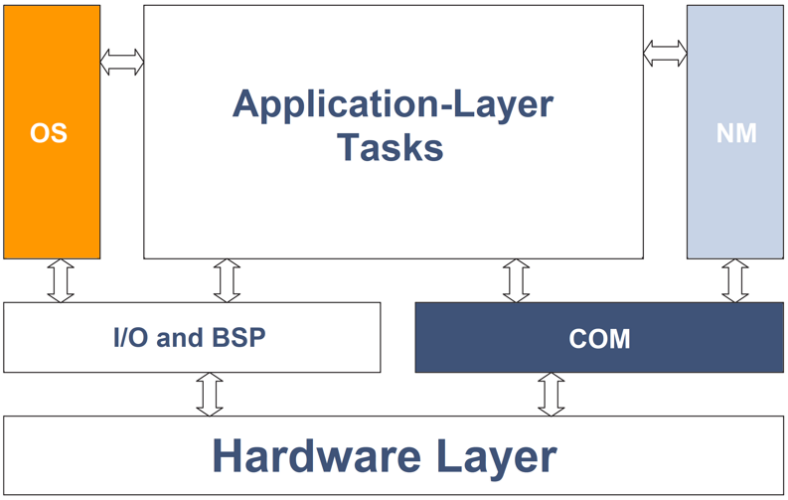


Figure 1.1: *OSEK/VDX Application Diagram.*

As can be seen in Figure 7.1, the application level is surrounded by the OSEK standard interfaces except for the I/O and BSP: the developer effort will be most spent in writing application code and, if not provided, the Board Support Package as interface to the Hardware referring to the Device Drivers.

1.2.1 Tasks Management

The OSEK/VDX Standard provides the definition of Task: it is intended as a part of control software divided by the others due to its Real-Time requirements [7]. Please remember that, in single core architectures, only a Task can run at once.

The OS scheduler allows a flexible Task management, providing concurrent, synchronous and asynchronous execution.

There exists two different Task types:

1. **Basic Tasks**: this is the most simple implementation of the task concept; they start to be executed and terminate. A basic Task release the CPU only if it terminates its execution, if the OS scheduler decides to switch the active Task to a higher priority one, or if a ISR has to be executed. Other Task switching mechanisms such as synchronizations and mutex are not supported. So, The Task will switch its status among the running, ready and suspended state:

   - *Running state*: The Task code is running on the CPU;
   - *Ready state*: The Task is ready to move to the running state, it is waiting for the scheduler to be chosen;
   - *Suspended state*: the Task reaches this state through a system service.

   All the possible transitions are represented in the follwing state diagram.

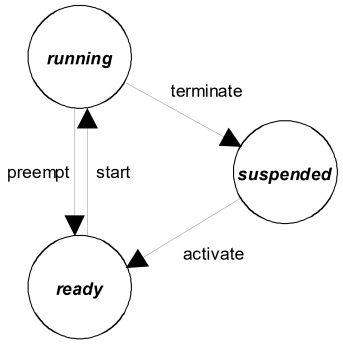


Figure 1.2: *Basic Task Transition Diagram [7].*

2. **Extended Tasks**: As said, basic Tasks cannot synchronize each others, neither waiting for other Tasks to be executed and then continuing. Features like these are introduced in the extended Tasks.

   The *waiting state* is now added to the state diagram; Tasks will reach it from the running state and they will remain in this state until an *Event* for which the Task is waiting for rises. If a Task waits for an *Event* that has already rose, it remains in the running state: it means that the events has to be cleared in order to be re-evaluated. An extended Task terminates only if it self-terminates, leading to a lower OS complexity.

   Definition of the *Event* OS object is found in section 1.2.2

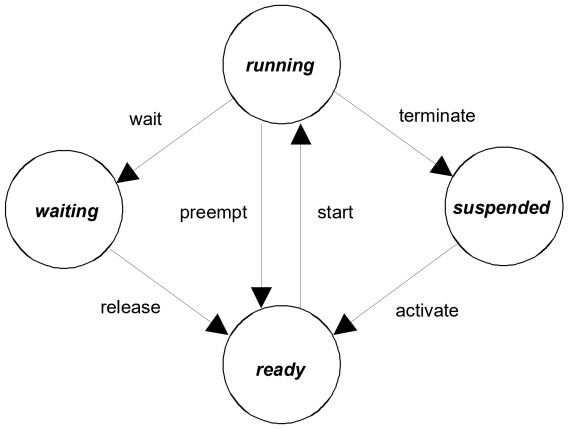


Figure 1.3: *Extended Task Transition Diagram [7].*

Managin the *waiting* state increases the complexity of the OS that has to use more complex resources with respect to using just basic Tasks, as for example *Semaphores*.

1.2.2 Events, Counters and Alarms

The OSEK standard provides as well the possibility to define, declare and manage **Events**. Each regular application specific triggers can rise particular events: a timer overflow, for example, can be considered an event; a Task that terminates is potentially considered as an event too. These kind of OS objects can be logically connected to the extended Tasks conformance classes *ECC1* and *ECC2*, that can handle them in different manners (wait for an event, set an event).

Tipically, common events are related to **Counters**: thinking about a Task, in order to implement a periodic execution, a counter can be exploited to count up to the corresponding period in clock ticks generating an event each time the value is reached.

What the standard OS offers is to define how the counters advance and the **Alarms** related to such counters. When Alarm expires, the OS services can be used to perform different actions, such as activate a Task, call a callback routine, set a particular Event, and so on.

In summary, Events can be seen as correlation between Tasks and the OS timing, by recurring on Alarms, or between Tasks that want to communicate to each other, by synchronizing their execution or to let a Task to be run before another one.

In full-preemptive OS implementations, the Events trigger the scheduler, in order to evaluate which Task has the right to run by considering that particular Event.

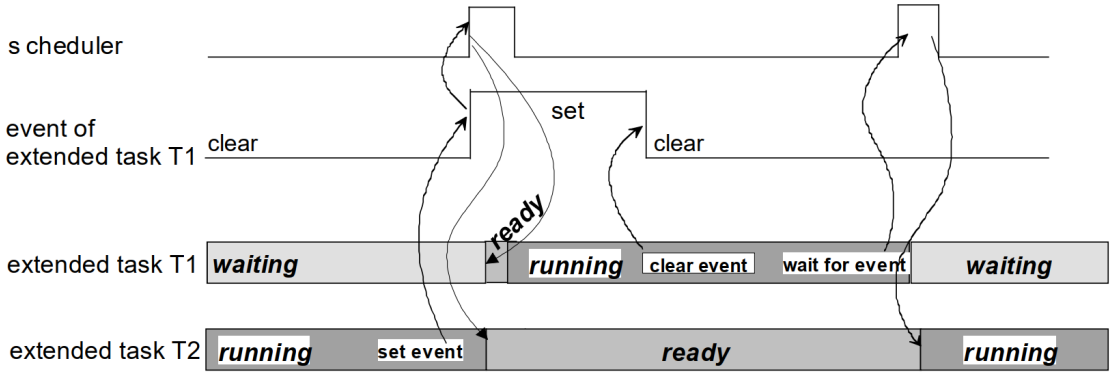


Figure 1.4: *Event-based Tasks Synchronization [7].*

1.2.3 OIL: OSEK Implementation Language

The 6th part of the standard (ISO 17356-6:2006) specifies the **OSEK Implementation Language** (OIL), i.e. the configuration language used to describe the application in a way to ensure the software portability defined by the OSEK standard. The OIL configuration file configures the OSEK application for a specified CPU; so, each CPU will have its own OIL description, with its fixed specified Task list and custom configurations. Each OSEK application is composed by different OIL objects such as Tasks, Alarms, Events, Counters, and all the resources and configurations needed to let the application works correctly. Particular Extended Conformance Class objects such as Semaphores have to be specified here too.

Dealing with a standard, additional custom objects cannot be created.

In particular, the OIL file contains two different parts: the system *implementation definition*, in which the CPU information need to be provided, and the *application definition*, describing the attributes of the application objects [8].

The configuration file will be used to generate the Operating System, by defining the proper Tasks and includes the needed OS software modules. This generation process ensure the composition of a very compact kernel, allowing a well OS memory size optimization.

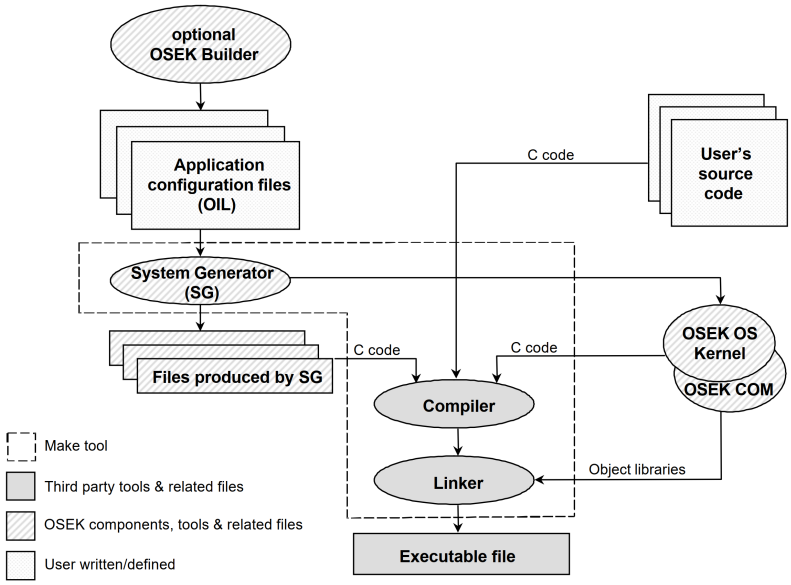


Figure 1.5: *OSEK/VDX OS Development Process [8].*

Thanks to the OSEK generation process, the developer will have to write only the OIL configuration file and the application code, by specifying the application functions and the Task bodies.

1.3 ERIKA Enterprise RTOS

The involved Real-Time Operating System is a made in Italy open source RTOS implementation of the OSEK/VDX API; it is compatible with many different well used microcontrollers such as the ARM M and R family, Infineon TriCore, Arduino, Altera Nios II, and so on. Being based on the OSEK/VDX standard, it supports all the previously mentioned features about OS objects such as Tasks, Events, Counters, Alarms, and so on.

Figure 1.6: *ERIKA Enterprise logo.*

With the help of this OS, the potential power of the multicore architecture can be exploited, being fully compatible and configurable for Real-Time parallel code running on architectures like these. The RTOS Reference Manual presents it by saying that *"Erika Enterprise offers the availability of a real-time scheduler and resource managers allowing the full exploitation of the power of new generation micro controllers and multicore platforms while guaranteeing predictable real-time performance and retaining the programming model of conventional single processor architectures"*. [9]

Being a Real-Time Operating System, it supports:

- Preemptive and non-preemtive multitasking management;
- Fixed Priority Scheduling. Priority levels are set by the developer into the OIL file;
- Shared Resources, by implementing the **Immediate Priority Ceiling** (IPC) to solve any possible Priority Inversion Problems: without its usage, a higher

priority Task can be preempted by another lower priority Task while waiting for a Task to releasing the shared resource. The IPC imposes that a Task holding a shared resouce acquires the higest priority between the Tasks sharing that resource;

- **Stack Sharing** between Tasks, as SRAM optimization;
- Hook functions before and after each context switch;

As mentioned above, it is a fully OSEK/VDX-based OS; to reduce the overall kernel footprint, 4 different conformance classes are provided in Erika Enterprise: they allow the OS to support just Basic Tasks (BCC1 and BCC2 conformance classes) or Extended Tasks (ECC1 and ECC2). The latter support all the additional features introduced in part 1.2.1 talking about extended Tasks, as for example the Synchronization and Events management primitive(up to n per Task, where n is the Microcontroller Architecture parallelism). These feature will be well used in the proposed software development approach, to periodically activate Tasks or to synchronize two dependent Tasks, one producing useful data to the other.

Feature	BCC1	BCC2	ECC1	ECC2
Multiple requesting of task activation	no	yes	no	BT: yes; ET: no
Number of tasks which are not in the *suspended* state	at least 255		at least 255 (any combination of BT/ET)	
More than one task per priority	yes	yes	yes	yes
Number of events per task	-		*nbits*	
Number of task priorities	*nbits*	16	*nbits*	16
Resources	$2^{nbits} - 1$ (including RES_SCHEDULER)			
Internal Resources	no limit (they are automatically computed by the OIL Compiler)			
Alarms	$2^{nbits} - 1$			
Application modes	$2^{nbits} - 1$			

Figure 1.7: *Conformance Classes Supported Features [9].*

Conformance classes ending with 1 (BCC1, ECC1) cannot store pending Task activations; instead, the ones ending with 2 (BCC2, ECC2) can store pending Tasks activations, and the maximum capacity is specified on the *ACTIVATION* field on the OIL Task specification. The number of allowed different Task priorities in BCC2 are 8; in ECC2, up to 16 different ones can be handled. By using Conformance classes ending with 1, n different priorities can exist, where n is the Microcontroller Architecture parallelism. Other kind of features are treated as integer numbers. For each Task, there can exist up to 2^n -1 Alarms, Application modes and Resources.

1.3.1 RT-Druid Development Environment

A full-custom Eclipse-based development environment for ERIKA Enterprise RTOS is also provided. RT-Druid allows to wrie, compile, run and analyze OS code and application code in a user-friendly manner. Being Eclipse-based, it is composed by different plugins. What is taken in consideration is the RT-Druid Code Generator: it allows to generate the whole Operating System code structure by simply analyzing the developed OIL RTOS configuration file or the AUTOSAR XML file; during this process, many different ERIKA Enterprise code generation routines are used, and the Operating System will include the needed resources to handle the custom application code. The developer has just to know how to include in the OIL file the needed resources and how to use them in the application code.

RT-Druid can also integrate the Cygwin development environment, in order to improve the functionalities allowing scripting features.

One of the most important features of RT-Druid is the ease of adaptation when changing architecture from single core to multi core or viceversa: the application code won't be affected by any changes, it's sufficient just to make some easy modifications on the configuration files and re-launch the build process. The application code design will result independent from multi-core issues, accelerating the development phase.

1.4 Target Board: NXP s32k144EVB-Q100

What was developed is a Model-Based Design development methodology for Task-level Erika Enterprise RTOS application code.

The target hardware used for evaluation tests and as base for developing the requested Firmware was the **S32K144EVB-Q100** board from NXP Semiconductors.

The board is shown in the following figure:

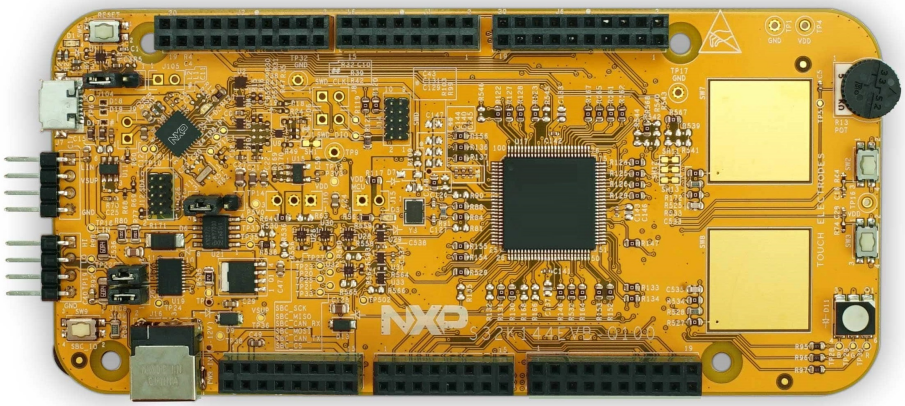


Figure 1.8: *NXP S32K144EVB-Q100 Evaluation Board.*

Being an prototyping Evaluation Board, it has many built-in interaction components such as users buttons, touch electrodes, a potentiometer and a RGB LED. An external power supply connector is present too. With these components the developers can easly simulate situations and events that can commonly rise to the applications: pushing a button can be intented as an unexpected sporadic event or an external Interrupt; turning on a LED can be intented as activate a electromechanical component, and so on.

The S32K144 SoC is thought for embedded automotive applications, so the evaluation board contains a CAN interface and a LIN interface as well: they are two of the mainly used serial communication protocols in the automotive field, ensuring a good level of reliability and connection dinstance. For example, the CAN protocol can be used for master/slave communications up to 1 Km, by paying as price a decrease in the data rate.

Many jumper inputs are also available, that can be used as GPIO, as connection

for the peripheral IOs or channels, or simply as Power Supply/Ground pin. The pinout mapping allows the board to be Arduino-compatible.

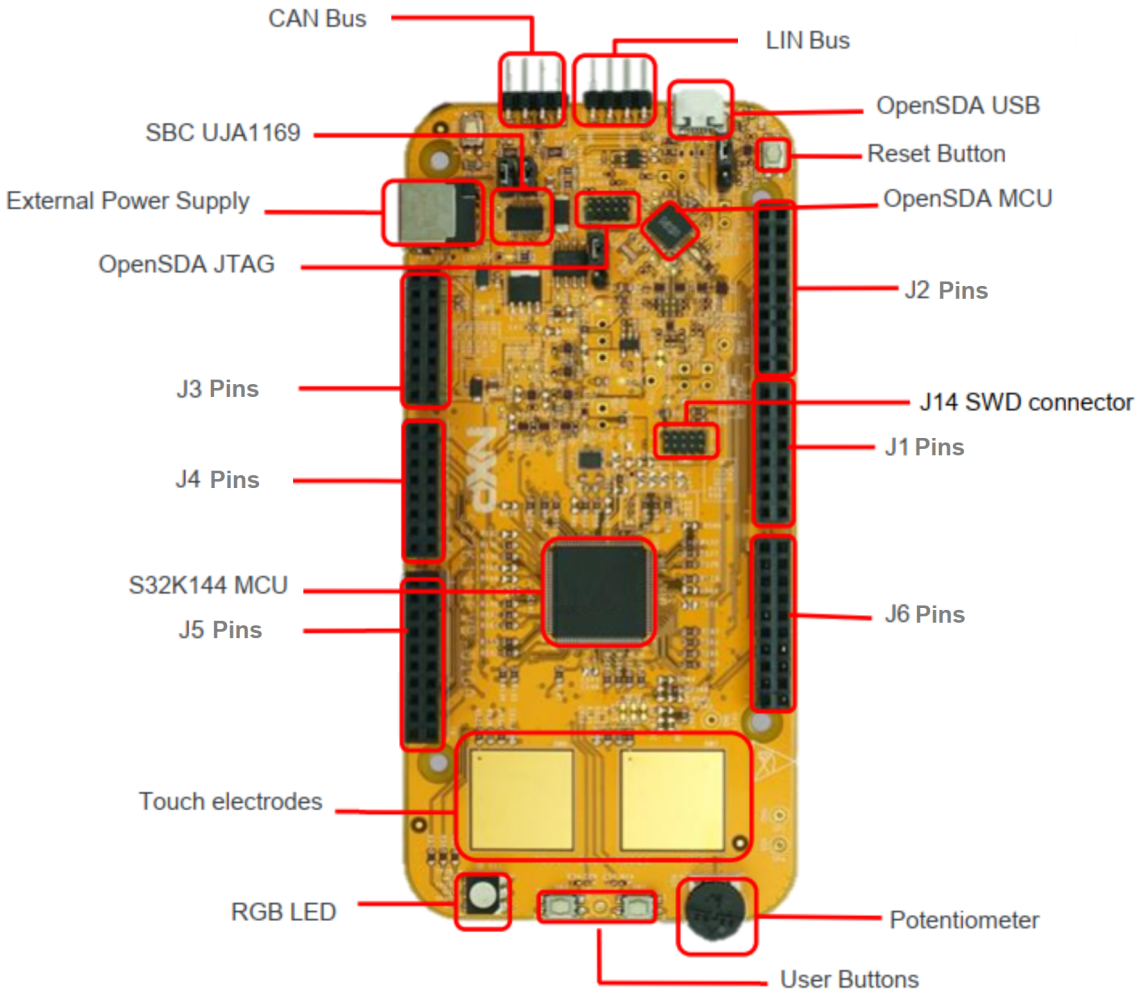


Figure 1.9: *Board Main Components.*

This SoC is based on the Arm Cortex-M4 MCU, a commonly used architecture in embedded system applications. It also contains many system peripherals that can be used for various different purposes; for example, they allow the device to support communication protocols like I2C, UART and SPI, to generate PWM signals, to perform Analog-to-Digital conversions, to transfer data without affecting the CPU usage.

The device is Functional safety compliant with ISO26262, including self-check

peripherals as Watchdog timer, voltage monitors, memory protection, clock monitors and cyclic redundancy checking.

NXP Semiconductors provides also a Design Studio useful to write bare metal C code to load on the board; in this case, it will not be intended as application software for a Real-Time Operating System running on the device.

In order to develop the Firmware used as base for the reference Simulink Library, the Device Reference Manual was taken as main consulting document, to better understand the peripheral Hardware structure, how they work and what are the configuration and control signals needed to manage to exploit the peripheral as desiderd.

Moreover, it also includes the OpenSDA Low-Cost Debug and Programming Interface. As its User's Guide says, "OpenSDA is an open-standard serial and debug adapter. It bridges serial and debug communications between a USB host and an embedded target processor"[10].

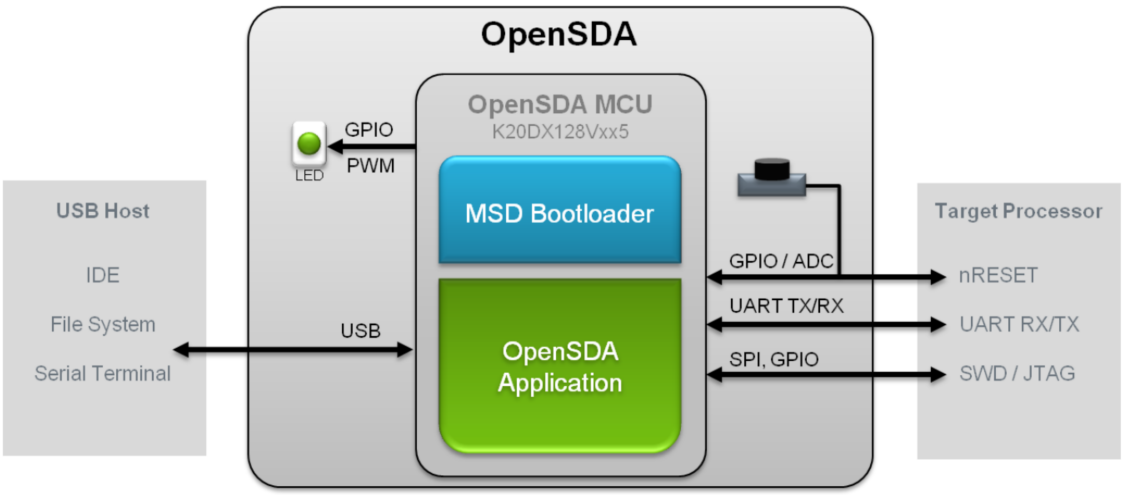


Figure 1.10: *OpenSDA Block Diagram [10].*

Chapter 2

Model-Based Software Design Background

In order to being able to go deeper in what was developed, some Model-Based Design concepts have to be introduced.

"***Model-Based Design*** *is a mathematical and visual approach for the development of complex control systems. It is systematic use of models throughout the development process for design, analysis, simulation, automatic code generation and verification. It is broadly used in motion control, industrial equipment, aerospace, and automotive applications*" [11].

It provides a block-based approach for software development, providing the possibility to design, simulate and validate full-custom models representing different kind of modules, as for example Control Units for electromechanical components; starting from the model, the code generation can be performed, in order to use the resulting code to build an executable that can run on the target device.

Embedded Software developers can use this approach to simulate some software parts to understand whether algorithms works fine or not. If satisfied from the obtained results, they can choose to generate the code and integrate it in the main project.

The main advantages of Model-Based Design are:

- **Accelerating development times**: Code Generation can lead to a non negligible time saving during the development process. In time-constraint projects, MBSD can be very helpful;
- **Reducing prototyping costs**: Simulation tools as Simulink can match the hardware performance requirements, so by simulating instead of periodically prototyping allows to save huge quantities of money and prototyping time;

- **Model Reutilization**: The same model (or submodel) can be easily reused and integrated in other models in more than one project; this allows the developer to save more time during his whole worklife;

- **Reducing SW Bugs**: with the help of a well developed Code Generation flow, the resulting code is conform to a kind of "standardization", i.e. it allows the Model-Based Designer to exploit the specific programming skills and knowledge of the developer who wrote the code generation templates, metrics and guidelines for the target device. This will lead to a huge reduction of software bugs on the generated code and reduces the difficulty of developing such software.

2.1 Using Simulink as Model-Based Design Tool

One of the most used tools for MBSD is Simulink, a modeling software for design and simuliation of dynamic systems. It is fully integrated in MATLAB, so its powerful resources can be exploited to model such high quality and precise models.

Moreover, Simulink allows code generation starting from models by using **Embedded Coder**. With this process, C, C++, optimized MEX functions and HDL code can be easily generated.

Dealing with C code generation, three main group of functions are generated:

- **Initialize function**: runs only once, it aims to setup the model and allocate the memory used to run the model step function;

- **Step function**: contains the model related C code; it will be periodically executed. Many applications use a fixed execution period, many others a variable one. It can be set in Simulink configurations;

- **Terminate function**: runs only once when the model software is not needed anymore, it aims to free the memory used by the model step function. If the step function will run forever, this function will be never called.

2.1.1 S-functions

In order to use some full-custom C code as for example and hand-written C algorithm or internal developed function calls, Simulink supports **S-functions**, i.e. a computer language description of a Simulink Block written in the target language. They are compiled as MEX files using the *mex* utility, to let MATLAB execution engine load and execute them [12].

Thanks to S-functions, the developer has the opportunity to add its custom code and mixing it to the models, by making Simulink a very flexible Model-Based development environment. In summary, they represent custom code packed in a Simulink block, supporting input and output ports, and **custom masks** to let the block user choosing as many functions parameters as the S-function developer wants.

2.1.2 Target Language Compiler and Real-Time Workshop

Even if Simulink offers a very huge potential for simulations, without code generation the models would remain just simulation objects. In order to perform the code generation process, Simulink includes a tool formerly called **Real-Time Workshop**, best known as *Simulink Coder*. This tool provides for the creation of model-based Software intended for real-time control of target-specific Hardware[13].

By analyzing the model structure, Simulink is able to generate the desidered code useful to emulate the model behavior in software. This process requires that each block is translated to the target language code. For this reason, the **Target Language Compiler** Tool, or TLC, is included in Simulink, allowing the developer to customize the generated code accordingly to the requested platform starting from any model. It's used to convert the model into C code. In order to proceed with this process, as first step of the RTW, the model is translated in a written form file, called *model*.rtw, then the TLC will analyze it as a text processor to compose the opportune generated code. Optionally, after the code generation, a build process can start, to directly produce an executable program via a cross-compile operation. In this last step, different compilers can be used, depending on the target device that has to execute the program. For example, ARM-based devices will need the *.elf* executable file. The executable can be automatically loaded into the target Hardware.

The TLC needs many different files to work correctly. All of them compose the **TLC program**. They are used to interpret the Simulink *.rtw* file and create the target code; they allow the system to produce code depending on where the code has to run, by following some metrics and templates dictated by the TLC code directives. There exist two kind of TLC files:

- **System Target Files**: They contain general information as for example the code language to be generated, the TLC templates to include, the name used to find it on the Simulink STF browser or setting inerithance from other STF. It can be thought as the main TLC file in the file collection.
- **Block Target Files**: They impose the code generation related to a block type when a block of that particular type is found when analyzing the rtw file. Each block will have the corresponding *block*.tlc file.

Figure 2.1 shows the complete Real-Time Workshop flow.

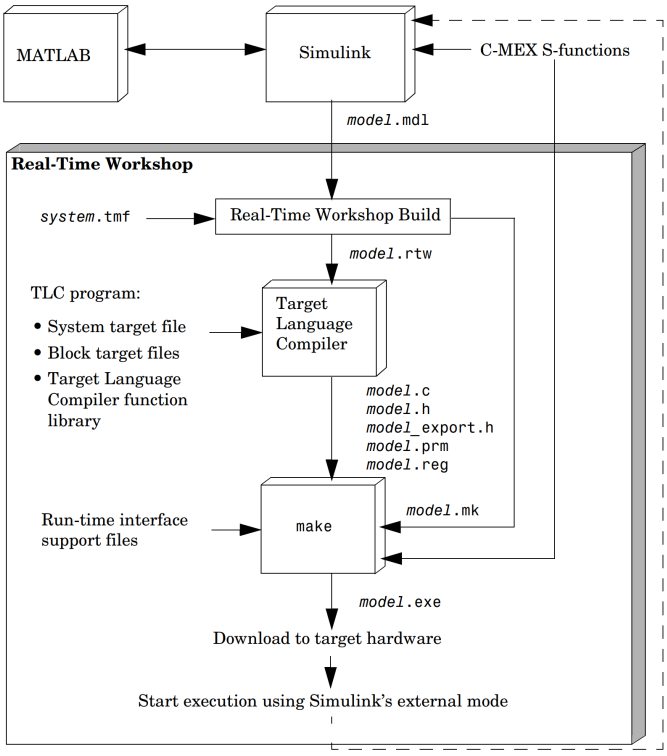


Figure 2.1: *Code Generation flow with RTW [14].*

It's common to do not use the whole flow, but just stopping at the code generation step; in cases like this, the Model-Based Designer will manually integrate the code in its project. It may happen when other software parts are already developed or in project in which the whole software is not only what it has to develop, as for example in situations with different development teams.

Chapter 3

BSP Extentions

Erika Enterprise OS is independent of the particular target used board: as seen, it can run in many different architectures and SoCs. The board-specific code used as interface from the Operating System (and application code) to the Hardware is called **Board Support Package**, or simply BSP.

The following layered scheme will show its position in the Software-to-Hardware hierarchy.

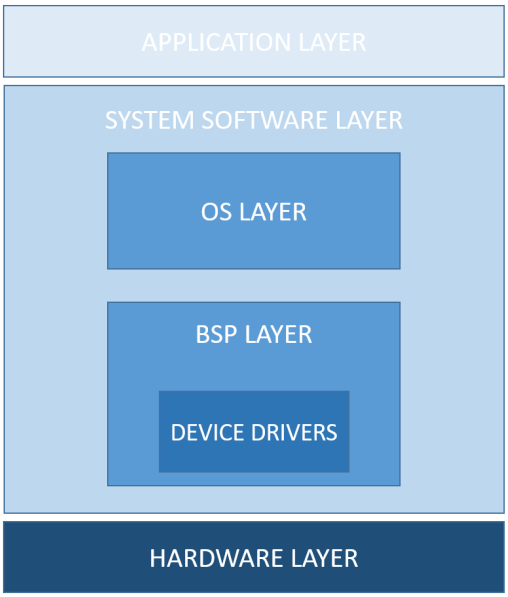


Figure 3.1: *SW-to-HW Hierarchy.*

The BSP provides a standard way for initializing, using and deinitializing the device's Hardware. It is a higher abstraction of the driver software concept. The first step needed to create the Simulink library aimed for Code Generation was to enhance the BSP of the RTOS for the target board.

In particular, the improvements were focused on two of the most important

peripherals: the Analog-to-Digital Converter and the FlexTimer Module. This last can be used for many different purposes, so the BSP was developed in order to use all of them. GPIO functions were also developed.

3.1 ADC functions

The Analog-to-Digital Converter functions allow the programmer to choose between one of the two ADCs modules and one of the channel for the conversion, from 0 to 15. It is also possible to choose the opportune conversion resolution, depending on the purpose of the conversion. The Conversion is SW-triggered.

As can be seen in the device Reference Manual, all of the ADCs Single Ended Channels are connected to the Alternate Mode 0 of the related pin, that is the default one when the device is turned on, so the functions will not care about setting it to a proper value.

- ***FUNC(void, APPL_CODE)*** *adc_setup(uint32_t instance)*: Activates the ADC clock and initializes the ADC registers given an instance; after having called this function, the ADC is ready to work, it just has to be triggered to begin a conversion. In particular:
  - The ADC Peripheral Clock Control Register *PCC_ADCx* is written to enable the peripheral *Clock*, where *x* is the Converter instance, i.e. ADC*0* or *1*. Here, the Peripheral Clock Source Select PCS is set to option 2 and after the Clock Gate Control CGC bit is set;
  - The ADC Status Control Register 3 *ADCx_SC3* is written to enable the conversion *Hardware Averaging*, in order to have more precise converted values. The amount of samples for this operation is specified in this register too, so it's fixed to the maximum one (32 samples averaged);
  - The ADC Status Control Register 3 *ADCx_SC3* is also written to start the *calibration* sequence; by setting the CAL bit, the sequence starts and cannot be interrupted. For calibration, it is mandatory to use averaging with the maximum samples number;

- ***FUNC(uint16_t, APPL_CODE)*** *adc_get_value(uint32_t instance, uint8_t chanIndex, uint8_t resolution)*: Starts the ADC conversion by using the given ADC and channel. The resolution has to be specified when this function is called.

  – A constant structure is taken as reference, containing the ADC default values about clock divisor, resolution, clock source, trigger mode, and other meaningful parameters needed to let the ADC peripheral working correctly;

  – The specified conversion resolution is used to update the structure field in order to let the drivers set the *MODE* field in the ADC Configuration Register 1 *ADCx_CFG1*;

  – In order to implement a SW-triggered conversion, a change on the *ADCH* Input channel select field of the ADC Status and Control Register 1 A *ADCx_SC1A* (corresponding to the channel 0 SC1 register) has to occur: the new contained value must correspond to the input ADC Single Ended Channel. In order to allow adjacent conversions from the same channel source, after each conversion an idle value is written on the register, in order to ensure a data change during the next conversion;

  – At the end, before reading the converted data, the function will wait until a conversion is completed, by waiting for the "Convertion Active" bit (ACAT field) in the *ADCx_SC2* register to be cleared.

  – The result is stored in the ADC Data Result Register *ADCx_Rn*, where n represent the channel identifier (A represents channel 0, B represents channel 1 and so on). The value is so returned.

- ***FUNC(void, APPL_CODE)*** *adc_unset(uint32_t instance)*: Deactivates the ADC clock and deinitializes the ADC registers given an instance.

  – All of the previous set configuration and status registers are cleared;

  – The ADC Clock is disabled by clearing the CGC field in the *PCC_ADCx* register;

3.2 FlexTimer functions

Timers are well used in many embedded applications, in different manners. With the FlexTimer Module, or FTM, it is possible to count up to a certain number of clock ticks starting a Interrupt Service Routine, or ISR, after, to generate Pulse Width Modulation signals to drive particular loads or to count how many clock ticks is long an input square wave. The new BSP functions allow the Operating System and the programmer to use in a very flexible way the timers FTM0, FTM1, FTM2 and FTM3; each of them has its own channels that can be used for the above mentioned different purposes, without restrictions. Each FTM Channel can be directly connected to a on-board Port configured in input, as for example in capture mode, or output direction, as for example to generate PWM signals: information about channels mapping are contained in the device description input multiplexing excel file that can be found as attachment of the Device Reference Manual.

3.2.1 Counting mode functions

- ***FUNC(void, APPL_CODE)*** *ftm_setup(uint32_t instance, uint8_t clock_div)*: Activates the FTM clock and initializes the FTM registers given an instance. To do so, a default configuration structure was created, containing the timer settings needed to let it works properly: the structure clock prescaler field is immediately updated with the user-defined function parameter; the other custom fields will be updated with the other function calls such as the PWM generation, the counting mode or the input capture.
  - The peripheral Clock is activated, by setting in the *PCC_FTMx* register the CGC bit;
  - The Features Model Selection register *FTMx_MODE* is updated in order to enable the free running Counter mode, by setting the *FTMEN* bit;
  - The structure fields are written in the Timer configuration Registers. The specified clock prescaler is written on the Status and Control register *FTMx_SC*, in the 3-bit field Prescaler PS; this value allows a 7-bit counter to reduce the FTM clock frequency, dividing the FTM source clock by a power of 2 up to 128. Figure 3.2 shows its behavior.

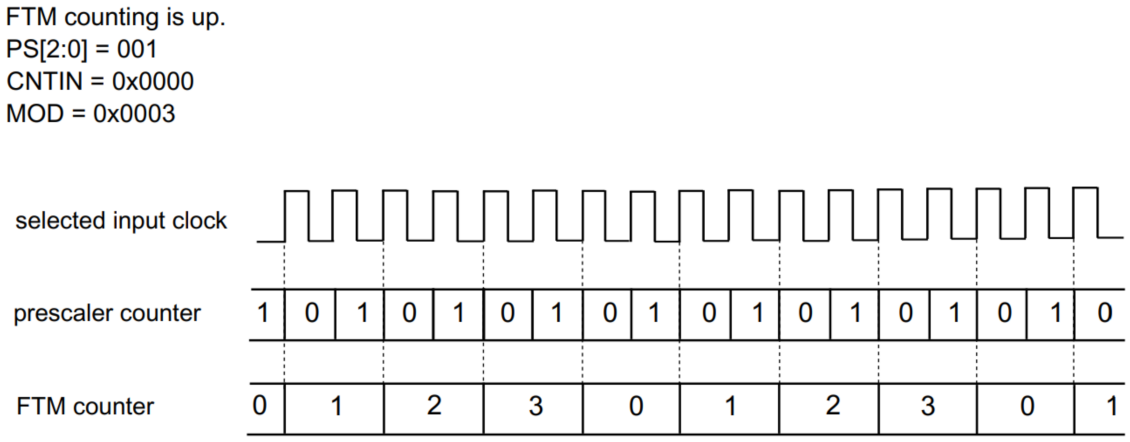


Figure 3.2: *FTM Prescaler behavior [15].*

- ***FUNC(void, APPL_CODE)*** *ftm_count(uint32_t instance, uint16_t initial_value, uint16_t final_value)*: Starts upcounting from the specified *initial_value* to *final_value*. At the end of the counting phase, the Interrupt Service Routine is called; a custom *FTMn_Ovf_Reload_IRQHandler()* has to be written by the programmer, where n is the instance number of the FTM. In particular:

  – The initial value from which the counter start to count and the final value that has to be reached by counting are passed by the programmer, and they are written respectively into the Counter Initial Value *FTMx_CNTIN* and Modulo *FTMx_MOD* Timer Registers;

  – After this, the Counter Clock Source is connected to the counter by setting the opportune bits of the field *CLKS* into the Status and Control Register *FTMx_SC*, so it will start counting.

  When reaching the MOD value, the Timer Overflow Flag bit *TOF* of the *FTMx_SC* register is Hardware-set and an **Timer Overflow Interrupt** rises. In order to manage other future interrupts, it has to be cleared, otherwise they will be ignored. A common developer choice is to clear the Overflow flag inside the Interrupt Service Routine.

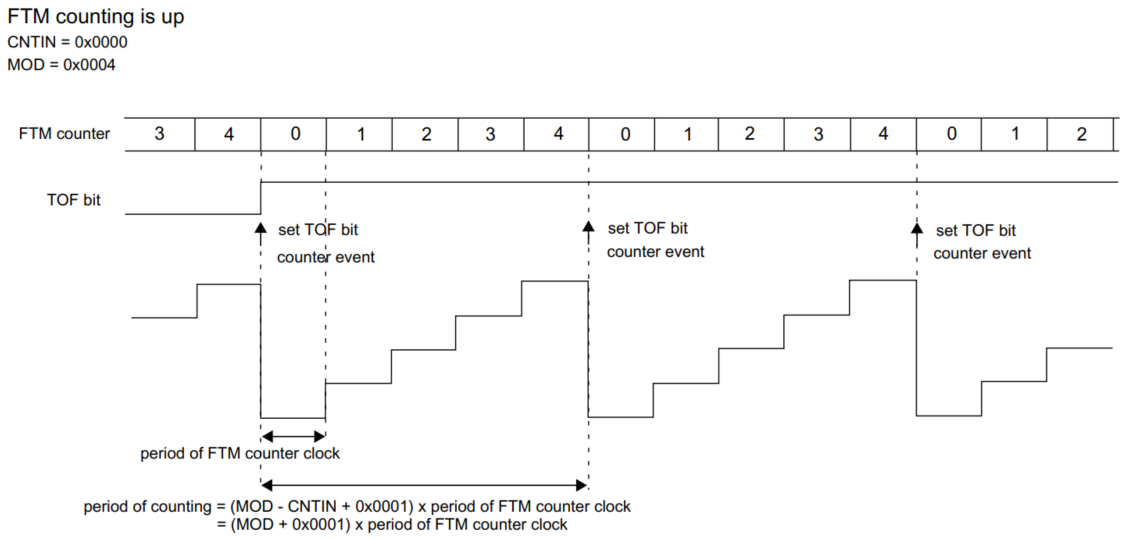


Figure 3.3: *FTM Counting mode behavior [15].*

- ***FUNC(void, APPL_CODE)*** *ftm_unset(uint32_t instance)*: Deactivates the FTM clock and deinitializes the FTM registers given an instance;

3.2.2 PWM generation functions

- ***FUNC(void, APPL_CODE)*** *ftm_pwm_setup(uint32_t instance, uint8_t chanIndex, uint8_t clock_div, uint16_t duty_cicle, uint32_t frequency)*: Initializes the FTM to generate an Edge-aligned PWM signal by using a certain channel as output. Before this function returns, the PWM signal starts to be generated by using the specified FTM channel. The Pulse frequency and duty cycle have to be specified when calling this function; the programmer can also set the opportune clock divisor. Getting more in details:

  – a default structure is instantiated, containing the basic configurations to let the PWM be generated, as for example the pin direction, a default pin port and index, and so on;

  – The structure fields are so modified accordingly to the function parameters; many values are derived from the channel parameter due to the dependency: it's used as index to read an array of structure fields such

as the pin port, the port index and the alternate muxing mode. This allows the programmer to use all of the timer channels by ignoring to know many hardware configurations that have to be performed. The channel index is chosen among the values of an enum definition, allowing also a more user-friendly way to choose the FTM channel.

- The function *ftm_setup* is called to initialize the FTM as explained above;
- Some of the structure parameters are used to configure the pin in output mode and connecting it to the opportune FTM channel. To do so, a bit corresponding to the port index into the opportune Port Data Direction Register *GPIOn_PDDR*, where *n* is the port letter from A to E, is set. To use the chosen pin as Timer channel, the alternate mode will be written into the Pin Control Register *PORT_PCRn*, on the *MUX* field; this field defines which alternate mode use, but they are chip-specific, as specified in the reference manual;
- In order to develop a Hardware-independent BSP interface, The Duty Cycle is passed as a paramater from 0 to 100, expressing the percentage of high period of a square wave in its complete period (high + low period), defined by the wave frequency. However, the register configurations allow a maximum duty cycle value of *0x8000h*: this was done to have a greater Duty Cycle resolution, hence to get a better control on the average value of the PWM wave. The user specified parameter is so normalized with respect to the maximum allowed value corresponding to the 100% of Duty Cycle.

  After the normalization, the value is used by the PWM drivers to compute the counter value corresponding to time in which the generated wave has to perform a falling edge: this computation also involves the developer specified frequency (in Hz), that will be also converted to a period value in timer clock ticks. In this way, it is more easy to understand how much clock ticks correspond to the specified Duty Cycle. As result of all of these computations, the first edge value is written into the Channel Value Register *FTMx_CnV*, where n is the channel index, and will be used as match value. The firmware imposes this mechanism by setting into the

Channel n Status and Control *FTMx_CnSC* the Edge or Level Select A and B fields, *ELSA* and *ELSB*, to 0 and 1. In this way, the channel output is forced high when a counter overflow occurs and the counter restarts counting, and it's set low when the counter value matches the previously computed value, corresponding to the Duty Cycle in clock ticks.

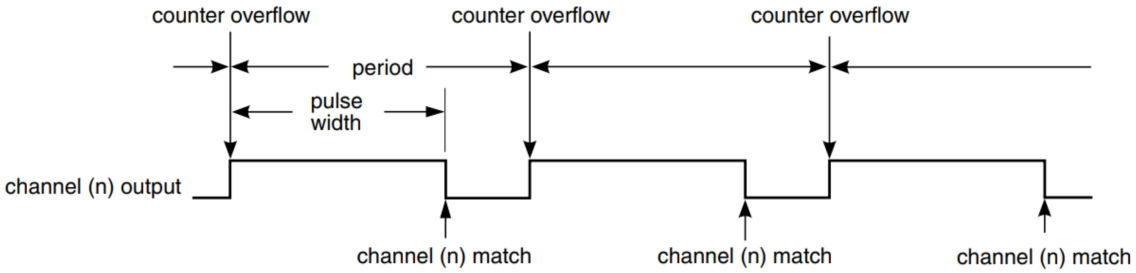


Figure 3.4: *FTM Edge-aligned PWM Generation [15].*

- ***FUNC(void, APPL_CODE)*** *ftm_pwm_unset(uint32_t instance)*: Stops the counter, deactivates the FTM clock and deinitializes the registers used for the PWM configuration;

3.2.3 PWM Input Capture mode functions

- ***FUNC(void, APPL_CODE)*** *ftm_capture_setup(uint32_t instance, uint8_t chanIndex, uint8_t clock_div)*: Initializes the FTM in capture mode.
  - As for the PWM mode, many of the parameters depend on the FTM channel to be used, so the same array of structure fields is read by using the developer-defined channel parameter; so the same configurations will be done for the port configuration and alternate mode muxing, except for the port direction that has to be set to input for the capture purpose;
  - The function *ftm_setup* is called to initialize the FTM as explained above;
  - A default structure is taken as reference, containing the necessary configuration values to perform a **Rising-Edge Aligned *Period-On Capture Signal Measurement***, i.e. to count how much clock ticks are contained between a rising edge and the corresponding falling edge. In

order to allow the maximum precision, the maximum Counter value is set to maximum allowed value on 16 bits, *0xFFFF* = 65535. The measurement will act in continuous mode; in this way, the capture value will be continuously updated when detecting a rising edge and a falling one.

  - The maximum Counter value is written into the *FTMx_MOD* Modulo Register; the continuous mode is imposed by opportunely setting the Mode Selection A and B fields, *MSA* and *MSB*, into Channel *n* Status and Control Register *FTMx_CnSC*;
  - Before enabling the clock, the Interrupt request for the channel is enabled; it indicates that the capture measurement is completed, and the Channel *n* Value Register *FTMx_CnV* is updated with the captured data;
  - At the end, the FTM Clock Source is set, and the peripheral will start to be ready to capture a signal;

- ***FUNC(uint16_t, APPL_CODE)*** *ftm_capture_get_value(uint32_t instance, uint8_t chanIndex)*: Returns a 16-bit value that indicates how much clock ticks are contained in the square wave recevied as input from the specified channel. Being the FTM configured in continuous capture mode, the function reads the last captured 16-bit value from the *FTMx_CnV* register;

- ***FUNC(void, APPL_CODE)*** *ftm_capture_unset(uint32_t instance, uint8_t chanIndex)*: Deinitializes the the Capture configuration for the specified FTM channel;

3.3 GPIO functions

Some functions related to the General Purpose IO usage were included in the BSP extention. Board built-in LEDs and buttons are directly connected to the GPIO registers.

- ***FUNC (void, APPL_CODE)*** *GPIO_port_config_input(uint8_t port, uint8_t index)*: Configures the opportune port in GPIO mode, in input direction;

- ***FUNC (void, APPL_CODE)*** *GPIO_port_config_output(uint8_t port, uint8_t index)*: Configures the opportune port in GPIO mode, in output direction;
- ***FUNC (void, APPL_CODE)*** *GPIO_write(uint8_t port, uint8_t index, uint8_t val)*: Writes the given digital value *val* on a GPIO; the on-board RGB LED is directly connected to the GPIOs, with an inverted logic (0 means LED on, 1 means LED off). In order to set or clear the GPIO digital value, the opportune Port Data Output Register *GPIOn_PDOR* bit is set or cleared; there is a bit for each GPIO pin;
- ***FUNC (uint8_t, APPL_CODE)*** *GPIO_read(uint8_t port, uint8_t index)*: Reads the digital value from a GPIO. The on-board buttons and the touch electrodes are directly connected to the GPIOs. The function returns the digital value (0 or 1), that wil be read from the Port Data Input Register *PDIR*, by reading the opportune bit.

Chapter 4

Simulink Library for MBSD

After the extention of the BSP for the target board, a Simulink Library was created starting from the new BSP functions. It consists in a group of Simulink blocks that can be instantiated on the models in order generate custom C code by using the new BSP functions. All of them were created by using the MATLAB Legacy Code Tool, in order to create the S-functions used to generate the opportune code with Embedded Coder in Simulink. The following procedure was executed just once for each block to be created, and so for each new function. After creating all blocks, the whole collection composes the library, so this process is not intended to be performed each time a block has to be used.

4.1 Block Generation Process

For sake of explanation, let's analyze step by step the procedure to create the block that implements the function "adc_get_value":

1. Using the MATLAB command line or by creating a script, execute the following code:

```
%% adc_get_value function
def0 = legacy_code('initialize');
def0.HeaderFiles = {'hal.h'};
def0.SFunctionName = 'adc_get_value';
def0.OutputFcnSpec = 'uint16 y1 = adc_get_value(uint32 p1, uint8 p2, uint8 p3)';
legacy_code('sfcn_cmex_generate', def0);
legacy_code('sfcn_tlc_generate', def0);
legacy_code('slblock_generate', def0);
```

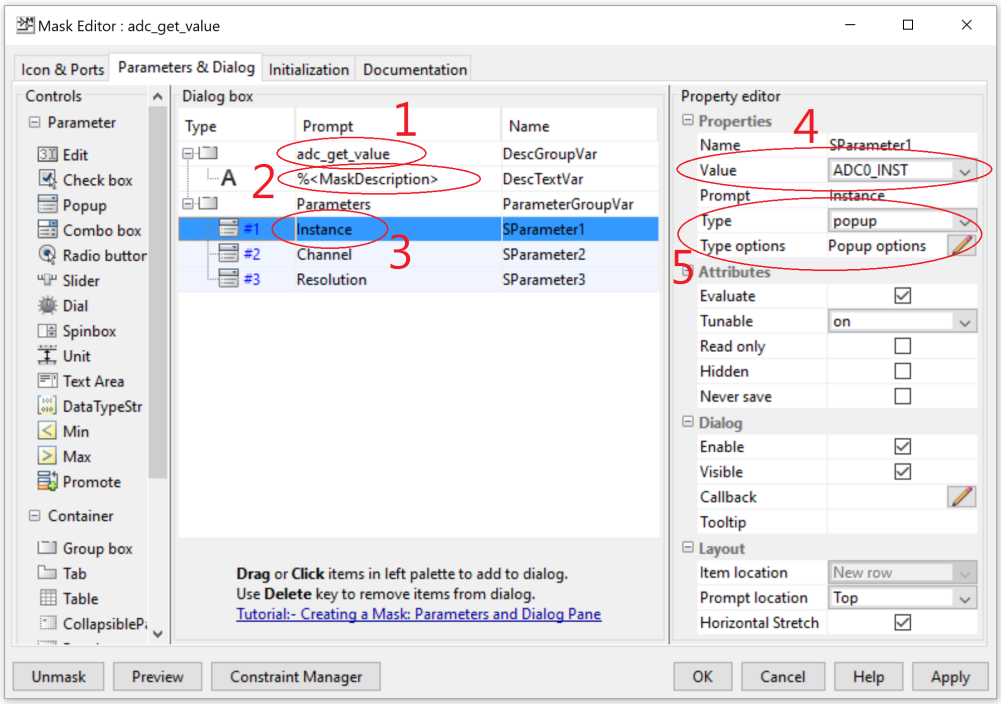
   - ***def0 = legacy_code('initialize')***: creates and initializes the structure *def0*, that will be filled with the needed useful information;

- ***def0.HeaderFiles = {'hal.h'}***: specifies the header file containing the function declaration.
- ***def0.SFunctionName = 'adc_get_value'***: specifies the function name;
- ***def0.OutputFcnSpec = 'uint16 y1 = adc_get_value(uint32 p1, uint8 p2, uint8 p3)'***: specifies the function usage; yi is the ith block output port, pi the ith function parameter (specified by using the mask) and, if present, ui refers to the ith block input port.
- ***legacy_code('sfcn_cmex_generate', def0)***: Generates the source C file containing the new created S-function;
- ***legacy_code('sfcn_tlc_generate', def0)***: Generates the TLC file, needed to recognize the blocks of *adc_get_value* type from Embedded Coder during the Code Generation Process;
- ***legacy_code('slblock_generate', def0)***: Opens a new Simulink window containing the new created block;
- ***mex 'adc_get_value.c'***: Compile the S-function; if this step is skipped, Simulink will not find the opportune S-function for the block of this type.

  For the purpose of simple code generation, the source file containing the S-function is not important; it is useful just to complete the code generation process.

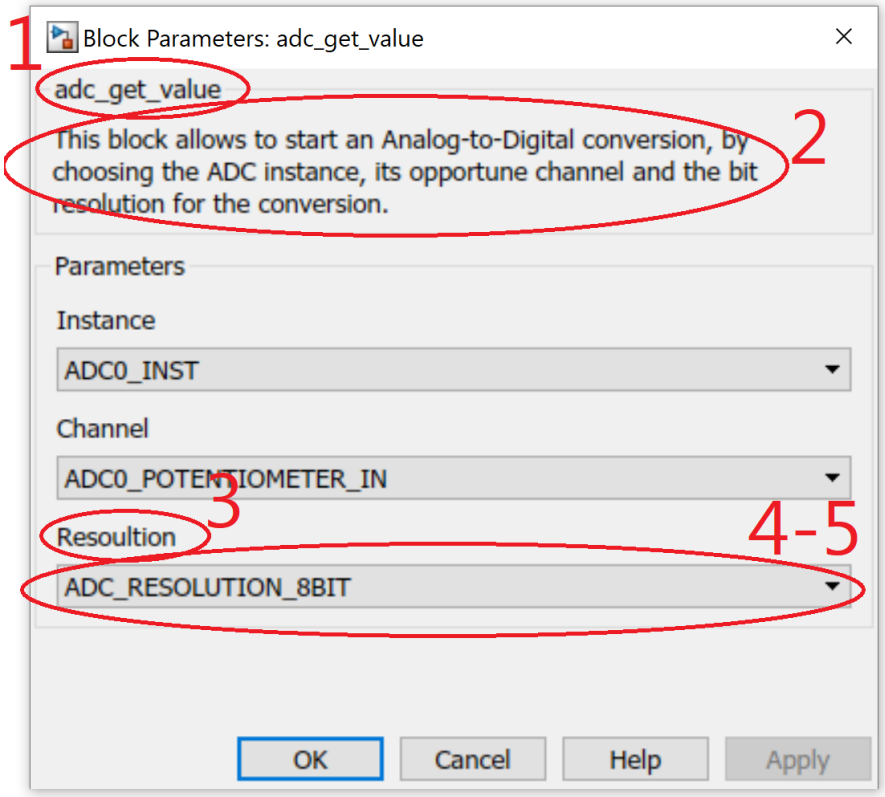
2. Once the block is created, the library user would be able to set the function parameters in a user-friendly manner. So, a mask for the new block is needed.

   - By right clicking on the block go to *Mask* → *Edit Mask*. A new window appears;
   - Click on the "Parameters & Dialog" tab. Here, you can specify all the parameters configurations; some of the most important ones are presented in the following:



In this window, you can specify the mask name (1), a reference to the mask description (2), change the parameter mask names (3), choose the default value of the parameters (4) and set the parameter configuration type (5). This last allows you to leave to the user as many freedom as you want. For example, if the parameter has no fixed assumable values, the type can be set to "edit"; instead, if the parameter has some fixed assumable values, the best choice is the popup, that allows the user to choose among the elements of a non-editable list, treated in front end as strings, and in back end (i.e. during the code generation) as incremental integer value. However, there are many other useful selectable types (combobox, radiobox, ...) that can be used depending on the application. The mask description that is referenced in this tab has to be written in the Documentation Tab.

- After having set everything, the mask is ready to be used, so the generated code will depend on the parameter as expected. The effects of the previous points are highlighted in the following mask window:



Even if the popup choice seems to be the most reasonable because it limits the freedom of the user to the programmer's choices, MATLAB treat every choice as a incremental integer number starting from 1. So, the first choice on the popup will generate 1, the second will generate 2, and so on. For this reason, the previously developed BSP function were adapted to the MATLAB enumerations: if **MATLAB_GEN_CODE** is defined into *hal.h*, the parameter inputs will be properly adjusted; otherwise, if it is not the case to deal with generated code, by not defining it the developer could do not mind about this offset adjustments and proceed with its task development coding by using the BSP enumerations, that allows to write a more readable code.

When the parameter values have to be specified by the user (like for example by using an edit parameter cofiguration type), in order to allow the user to insert a reasonable value for the specified parameters some constraint about the input was necessary. Keep in mind that not all the configuration types are constraintable.

To create a constraint:

1. Go to the mask editor (Right click on the block, cick on Mask → Edit Mask...);
2. Click on the parameter that has to be constrained;
3. in the property editor, under the Attributes voice, click on the Constraint popup and click on "Add new constraint"; a new window opens;
4. Insert a name for the new constraint on the opportune edit field;
5. Set the opportune constraints on the minimum and/or maximum accepted values, on the sign, on the type and so on. If needed, a custom MATLAB expression can be used as constraint;

The created constraint can also be applied to other configuration parameters; when created, it will appear on the above mentioned constraint popup so it can be chosen.

4.2 Library Deployment

When all the blocks for each new function of the BSP and their corresponding masks were created, the complete library was saved and deployed, in order to find it in the Simulink Browser. This process is also useful to easly find the block by double clicking on the model and writing one of the created library block name.

The following MATLAB function was developed and executed, and the deployment succeeded.

```
function blkStruct = slblocks
        % This function allows the library to appear
        % in the Library Browser and to be cached in
        % the browser repository

        %Specify the library Simulink file name
        Browser.Library = 'S32K144_erika_RTOS_lib';

        %Specify the name that will appear in the Library browser
        Browser.Name = 'Erika Enterprise RTOS BSP Library for NXP S32K144EVB';

        blkStruct.Browser = Browser;
```

Figure 4.1: *MATLAB code for Simulink Library integration*

After the deployment, by refreshing the Simulink browser, the deployed library will appear and the Designer can easly access to it. Remember that once deployed the new Simulink Library will be always visible, even after restarting MATLAB; so, the previously block creating method will not be needed anymore:

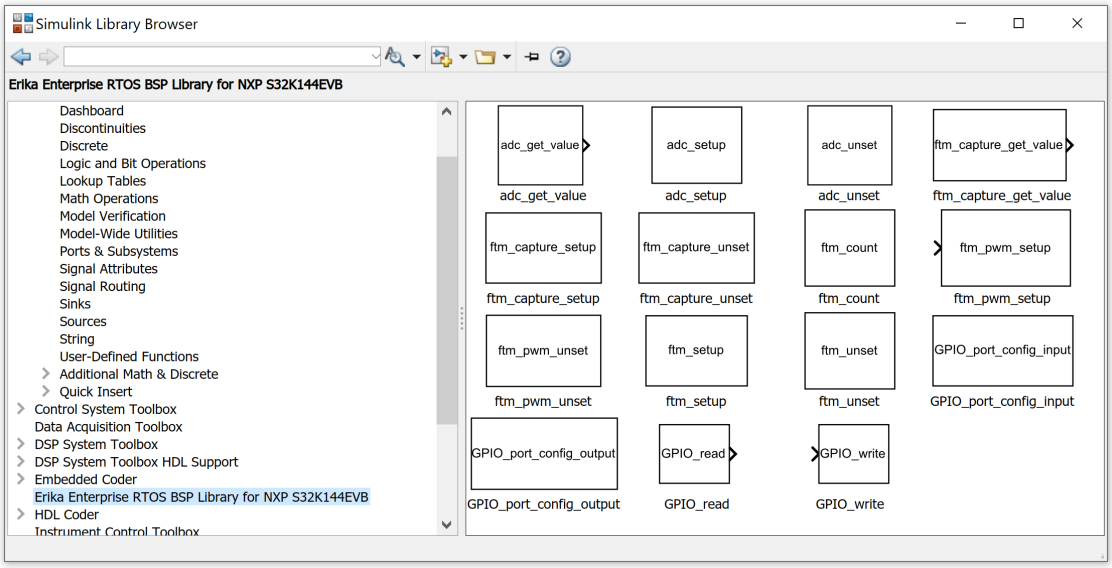


Figure 4.2: *Developed library in the Simulink Library Browser*

Chapter 5

System Target File for RTW Code Generation

Simulink should be able to generate Erika RTOS-compatbile code starting from the model. A System Target File aimed to create the Task C file and the OIL OS configuration file was created.

5.1 MATLAB Model Analysis Function Library

The following MATLAB functions were developed to be called during the RTW by the TLC files: they aim to easily provide information about the Model having as additional result a more readable TLC code.

- *get_task_period(model, task_name)*: given a model and a task name, it returns the task sample time in milliseconds;

```
function r = get_task_period(model,task_name)
%Get the Task period in ms

    load_system(model);
    blocks = find_system(model, 'SearchDepth','1');
    bsize = size(blocks);
    r = 0;
    if(bsize(1) > 1)
        %For each block in the top level...
        for i = 2: bsize(1)
            tmpname = get_param(blocks{i},'Name');
            %... If the block name is equal to the given Task name...
            if (strcmp(tmpname,task_name))
                %... Return its Sample Time.
                tr = get_param(blocks(i),'SystemSampleTime');
                r = str2num(tr{1}) * 1000;
                return
            end
        end
    else
        disp("Empty Model!")
    end
end
```

- *get_input_task_from_RT(model,rt_name)*: given a model and a Rate Transition block name, this function returns the Rate Transition writer Task name;

```
function r = get_input_task_from_RT(model,rt_name)
%Given a Rate Transition block name,returns the writer Task name

    load_system(model);
    blocks = find_system(model, 'SearchDepth','1');
    bsize = size(blocks);
    r = "";
    if(bsize(1) > 1)
        %For each block in the top level...
        for i = 2: bsize(1)
            ports = get_param(blocks{i},'PortConnectivity');
            ss = size(ports);
            if (ss(1) > 0)
                for j = 1 : ss(1)
                    %... Search for the block that has the RT as Dst. ...
                    dst = get_param(ports(j).DstBlock, 'Name');
                    if(strcmp(dst,rt_name))
                        %... And return the Task name.
                        r = get_param(blocks{i},'Name');
                        return
                    end
                end
            end
        end
    else
        disp("Empty Model!")
    end
end
```

- *get_output_task_from_RT(model,rt_name)*: given a model and a Rate Transition block name, this function returns the Rate Transition reader Task name;

```
function r = get_output_task_from_RT(model,rt_name)
%Given a Rate Transition block name,returns the reader Task name

    load_system(model);
    blocks = find_system(model, 'SearchDepth','1');
    bsize = size(blocks);
    r = "";
    if(bsize(1) > 1)
        %For each block in the top level...
        for i = 2: bsize(1)
            ports = get_param(blocks{i},'PortConnectivity');
            ss = size(ports);
            if (ss(1) > 0)
                for j = 1 : ss(1)
                    %... Search for the block that has the RT as Src ...
                    src = get_param(ports(j).SrcBlock, 'Name');
                    if(strcmp(src,rt_name))
                        %... And return the Task name.
                        r = get_param(blocks{i},'Name');
                        return
                    end
                end
            end
        end
    else
        disp("Empty Model!")
    end
end
```

- *get_RT_input(model,rt_name)*: given a model and a Rate Transition block name, this function returns the Rate Transition input variable name;

```
function r = get_RT_input(model,rt_name)
%Get the RT input variable name

    load_system(model);
    blocks = find_system(model, 'SearchDepth','1');
    bsize = size(blocks);
    r = 0;
    if(bsize(1) > 1)
        %For each block in the top level...
        for i = 2: bsize(1)
            tmpname = get_param(blocks{i},'Name');
            %... If the block name is equal to the given RT name...
            if (strcmp(tmpname,rt_name))
                tr = get_param(blocks{i},'PortHandles');
                %... Return the input arrow label.
                r  = get_param(tr.Inport,'Name');
                return
            end
        end
    else
        disp("Empty Model!")
    end
end
```

- *get_RT_output(model,rt_name)*: given a model and a Rate Transition block name, this function returns the Rate Transition output variable name;

```
function r = get_RT_output(model,rt_name)
%Get the RT output variable name

    load_system(model);
    blocks = find_system(model, 'SearchDepth','1');
    bsize = size(blocks);
    r = 0;
    if(bsize(1) > 1)
        %For each block in the top level...
        for i = 2: bsize(1)
            tmpname = get_param(blocks{i},'Name');
            %... If the block name is equal to the given RT name...
            if (strcmp(tmpname,rt_name))
                tr = get_param(blocks{i},'PortHandles');
                %... Return the output arrow label
                r  = get_param(tr.Outport,'Name');
                return
            end
        end
    else
        disp("Empty Model!")
    end
end
```

- *get_task_priority(model,taskname)*: given a model and a Task name, this function returns the Task priority, following the Rate Monotonic rules;

```
function pr = get_task_priority(model,task_name)
%Get the Task period in ms

    load_system(model);
    blocks = find_system(model, 'SearchDepth','1');
    bsize = size(blocks);
    r = 0;
    count = 0;
    if(bsize(1) > 1)

        %Count how much Tasks are contained into the model
        for i = 2: bsize(1)
            if(strcmp(get_param(blocks{i},'BlockType'),'SubSystem'))
                count = count + 1;
            end
        end

        periods = zeros(1,count);
        k = 1;

        %For each block in the top level...
        for i = 2: bsize(1)
            %... If the block is a Task...
            if(strcmp(get_param(blocks{i},'BlockType'),'SubSystem'))
                %... Save its Sample Time in ms.
                tr = get_param(blocks(i),'SystemSampleTime');
                r = str2num(tr{1}) * 1000;
                periods(k) = r;
                k = k + 1;

                %if the analyzed task is what we need...
                if(strcmp(get_param(blocks{i},'Name'),task_name))
                   ...save its period to find in the future its priority
                   taskperiod = r;
                end
            end
        end

        %array of unique periods in ascending order
        periods = unique(periods);

        %return the task period index -> priority
        pr = find(periods==taskperiod);
        return
    end
```

In TLC coding, it is easy to get the Rate Transition name list, but it is not to get the connected Subsystems neither the variable names; so, for each Rate Transition Block found, these functions will be called in order to get the opportune information needed for code generation.

5.2 TLC Files for Custom Code Generation

The proposed System Target File enables Simulink to generate Tasks running at different rates and managing Inter-Process Communications. Moreover, it imposes Tasks priorities following the **Rate Monotonic** Scheduling Algorithm metrics: by doing a static analysis during the code generation process, Tasks with minor cycle duration will have assigned a higher priority. Please keep in mind that in OSEK OS the higest priority level is 1.

The System Target File provides also the OS Oil Configuration File generation, by defining the OS objects needed to let the generated Tasks working opportunely as described in the Model.

In particular, different TLC files with different purposes were created:

- ***mbd_s32k14_erika_rtos.tlc***: System Target File. It contains the header needed to be recognized by Simulink as TLC file in the model Configuration Parameters; many internal parameters such as the code format, the model name and the Language are set here. The other TLC files are included in this one.

  Moreover it contains the specification through which this STF becomes derived from the standard "ert.tlc", so all of the ERT options and code generation styles are inherited. The suffix for the generated directory name is also specified here;

- ***mbd_oil_erika_rtos.tlc***: This is the configuration oil file template; it works similarly to the task template, but what is generated is the RTOS configuration file, by defining the opportune tasks and, by analyzing their sample time, alarms and events that allow to set the periodicity of each task and other needed OS objects.

  Let's start analyzing how it is composed:
  In the first lines, some basic information about the target device and CPU are provided, such as the operating frequency, the CPU, SoC and Board model and so on; due to the fact that the development board was chosen at the beginning, these information are fixed, so there are no dependencies between the template and the model:

```
36  CPU mySystem {
37
38      OS myOs {
39          EE_OPT = "OS_EE_APPL_BUILD_DEBUG";
40          EE_OPT = "OS_EE_BUILD_DEBUG";
41
42          USERESSCHEDULER = FALSE;
43          CPU_DATA = CORTEX_M {
44              MODEL = M4;
45              MULTI_STACK = TRUE;
46              IDLEHOOK = TRUE {
47                  HOOKNAME = "idle_hook";
48              };
49              CPU_CLOCK = 48.00;
50              EXECUTE_FROM_RAM = TRUE;
51          };
52
53          MCU_DATA = S32K1XX {
54              MODEL = S32K144;
55          };
56
57
58          BOARD_DATA = S32K144EVB_Q100;
59
```

Figure 5.1: *TLC Oil file - Target device and CPU information.*

Moreover, the additional hook functions have to be specified here. The proposed oil file specifies just an idle hook function.

A special resource, the **RES_SCHEDULER**, is also supported by Erika Enterprise RTOS. It has a ceiling equal to the highest priority. So, a task locking RES_SCHEDULER becomes non-preemptive. If needed, the **USERESSCHEDULER** parameter has to be set to TRUE [9]. The proposed configuration set it to FALSE.

In case of depending tasks with different rates, the semaphores have to be defined in the "*USEEXTENSIONAPI*" section, contained inside the "*OS*" section. The used naming rule will impose the composition of the semaphore name to be dependent to both the writer and reader Task, so it will be **{*writer_task_name*}_{*reader_task_name*}_sem**:

```
%%IN CASE SEMAPHORES ARE NEEDED, THEY ARE ADDED HERE
%foreach j = ::CompiledModel.BlockHierarchyMap.NumSubsystems
    %if ::CompiledModel.BlockHierarchyMap.Subsystem[j].Type == "root"
        %assign tmp = 0

        %%FOR EACH BLOCK IN THE ROOT LEVEL...
        %foreach k = ::CompiledModel.BlockHierarchyMap.Subsystem[j].NumBlocks
            %assign btype = ::CompiledModel.BlockHierarchyMap.Subsystem[j].Block[k].Type
            %assign bname = ::CompiledModel.BlockHierarchyMap.Subsystem[j].Block[k].SLName

            %%...IF THE BLOCK IS A RATE TRANSITION...
            %if btype == "RateTransition"
                %assign input_task =  FEVAL("get_input_task_from_RT",model_name,bname)
                %assign output_task = FEVAL("get_output_task_from_RT",model_name,bname)

                %if tmp == 0
    USEEXTENSIONAPI = TRUE {
                    %assign tmp = 1
                %endif

                %%...SPECIFIES THE NAME OF THE SEMAPHORE NEEDED TO MANAGE THE DATA DETERMINISM BETWEEN THE TASKS.
                /* Semaphore for Rate Transition between Tasks %<input_task> and %<output_task> */
        SEMAPHORE = DEFAULT { NAME = "%<input_task>_%<output_task>_sem"; COUNT=0; };

            %endif
        %endforeach
        %if tmp == 1
            };
        %endif
        %break
    %endif
%endforeach
```

Figure 5.2: *TLC Oil file - Semaphore specifications.*

In order to easily get the writer and the reader task name, the developed library functions *get_input_task_from_RT* and *get_output_task_from_RT* are used.

Other fixed specifications are included in the oil template; the following lines will specify the library for the target device. As said in part 1.2.2, in order to exploit alarms and events to impose the task periodicity, the conformance class has to be specified in the Oil file; being a fixed information, the specification will not depend on the model:

```
97          LIB = S32_SDK {
98              /* Used to select Board: S32K144EVB-Q100 */
99              BOARD = S32K144EVB_Q100;
100             /* Used to select library version. */
101             VERSION = "0.8.6 EAR";
102             /* Create libs32sdk.a */
103             STAND_ALONE = TRUE;
104         };
105
106
107
108         STATUS = EXTENDED;
109
110         KERNEL_TYPE = OSEK {
111             %%ECCx TO CREATE TASK PERIODICITY WITH EVENTS AND ALARMS
112             CLASS = ECC1;
113         };
114
115     };
```

Figure 5.3: *TLC Oil file - Library and conformance class specifications.*

Application source files, such as the file containing the task definitions, the file containing the main function that will start the operating system, the BSP and the generated model source file, have to be specified under the **APPDATA** field. The model source file name will depend on the model name.

```
117     APPDATA myApp {
118         APP_SRC  = "hal.c";
119         APP_SRC  = "code.c";
120         APP_SRC  = "task.c";
121         %%INCLUDE THE MODEL SOURCE TO THE APPLICATION CODE
122         APP_SRC  = "%<FcnMdlName()>.c";
123     };
124
125
126     COUNTER SystemTimer {
127         MINCYCLE = 1;                /* min value for cycle, typically 1 */
128         MAXALLOWEDVALUE = 65535;     /* once reached the counter wraps */
129         TICKSPERBASE = 1;            /* how many tick hw are need to obtain a sw tick */
130         TYPE = HARDWARE {            /* device that we are using */
131             SYSTEM_TIMER = TRUE;
132             PRIORITY     = 1;          /* ISR2 Priority of the systick timer */
133             DEVICE       = "SYSTICK";
134         };
135         SECONDSPERTICK = 0.001;      /* period of the system timer in seconds */
136     };
137
```

Figure 5.4: *TLC Oil file - Application files and System Timer specifications.*

A counter is also included in the specifications: it will be used as **System Timer**, and will use as hardware source the *Systick* device timer, to avoid using general purpose timers (FTM) that can be used for different application purposes.

Before going on with the Task specifications, in order to avoid the Priority Inversion Problem between Tasks an Erika Enterprise OS object, the **Resource**, is used: for each Rate Transition Block connected between a lower priority Task and a higher priority one, a Resource will be declared; the used nomenclature is ***{writer_task_name}_{reader_task_name}_res.***

```
%%RESOURCE DECLARATIONS: USED TO AVOID PRIORITY INVERSION PROBLEM
%foreach j = ::CompiledModel.BlockHierarchyMap.NumSubsystems
    %if ::CompiledModel.BlockHierarchyMap.Subsystem[j].Type == "root"
        %assign tmp = 0

        %%FOR EACH BLOCK IN THE ROOT LEVEL...
        %foreach k = ::CompiledModel.BlockHierarchyMap.Subsystem[j].NumBlocks
            %assign btype = ::CompiledModel.BlockHierarchyMap.Subsystem[j].Block[k].Type
            %assign bname = ::CompiledModel.BlockHierarchyMap.Subsystem[j].Block[k].SLName

            %%...IF THE BLOCK IS A RATE TRANSITION...
            %if btype == "RateTransition"
                %assign input_task =  FEVAL("get_input_task_from_RT",model_name,bname)
                %assign output_task = FEVAL("get_output_task_from_RT",model_name,bname)
                %assign input_priority  = FEVAL("get_task_priority",model_name,input_task)
                %assign output_priority = FEVAL("get_task_priority",model_name,output_task)

                %%...IF THE READER TASK HAS A HIGHER PRIORITY (1 = HIGHEST PRIORITY)...
                %if output_priority < input_priority

                    %%...DECLARE A RESOURCE TO AVOID PRIORITY INVERSION
                    /* Resource declaration: avoid priority inversions in critical sections */
                    RESOURCE %<input_task>_%<output_task>_res { RESOURCEPROPERTY = STANDARD; };

                %endif
            %endif
        %endforeach
        %break
    %endif
%endforeach
```

Figure 5.5: *TLC Oil file - Resource declarations.*

During the Real-Time Workshop, the template reads the generated model RTW file and finds any non-virtual subsystems at the depth 1 starting from the model root, that corresponds to the top level of the module. In this way, the name of each Task is easily reachable. In order to set the task cycle period, Erika Enterprise allows the tasks to be Event-sensitive. The events can be periodically set by the OS configurable alarms. To exploit these OS objects, the alarm will inherit the period from the opportune Simulink Subsystem. So, each Task will wait for its "**wake up**" event generated at the same period of the task on the design by the periodc alarm; after waiting, the event is cleared and the task execution starts. The whole described code is contained in a endless loop. So, the last part of the Oil template will contain the task, events and alarms specifications. The declared task names, alarm names and event names depend on the Subsystem names, in accordance with the C generated

code. As can be seen in Figure 5.6, for each Task in the model, its name is extracted from the model itself, the sample time is easily obtained by calling the developed MATLAB function *get_task_period*.

Moreover, the Task priority is also defined here: it will depend on the Task periodicity, accordingly to the **Rate Monotonic** scheduling algorithm. For this purpose, the developed library function *get_task_priority* is called.

```
%%TASK DECLARATIONS

%foreach j = ::CompiledModel.System[GetBaseSystemIdx()].NumTotalBlocks

    %%FOR EACH TASK FOUND IN THE ROOT LEVEL...
    %if ::CompiledModel.System[GetBaseSystemIdx()].Block[j].Type == "SubSystem"

        %%GET THE TASK NAME
        %assign taskname = ::CompiledModel.System[GetBaseSystemIdx()].Block[j].Identifier
        %assign tname = ::CompiledModel.System[GetBaseSystemIdx()].Block[j].Name

        %%GET THE TASK SAMPLE TIME
        %assign tasksampletime = CAST("Number",FEVAL("get_task_period",model_name,taskname))

        %%GET THE TASK PRIORITY
        %assign taskpriority = CAST("Number",FEVAL("get_task_priority",model_name,taskname))


        %%PRINT INFOS ON THE DIAGNOSTIC VIEWER
        %matlab disp ("Task: " + taskname)
        %matlab disp("Sample Time (ms): " + STRING(tasksampletime))
        %matlab disp("_____")



        %%...DEFINE THE TASK INTO THE OIL FILE...
        TASK %<taskname> {
            PRIORITY = %<taskpriority>;
            ACTIVATION = 1;
            SCHEDULE = FULL;
            AUTOSTART = TRUE;
            STACK = PRIVATE {
                SIZE = 512;
            };

```

Figure 5.6: *TLC Oil file - Data getting and Task specification.*

The TLC code contains some print lines for the Diagnostic Viewer; printing strings is a very good way to debug the TLC file templates during the development process.

In case the Task has a higher priority with respect to other Tasks that generate data for its execution, the resources must be specified in its specification, so for each Rate Transition connected to the Task, if it has lower priority writer Tasks, the opportune resource will be added to the Task specifications accordingly to the nomenclature imposed in figure 5.5

```
209             %foreach j = ::CompiledModel.BlockHierarchyMap.NumSubsystems
210                 %if ::CompiledModel.BlockHierarchyMap.Subsystem[j].Type == "root"
211                     %%FOR EACH BLOCK IN THE ROOT LEVEL...
212                     %foreach k = ::CompiledModel.BlockHierarchyMap.Subsystem[j].NumBlocks
213                         %assign btype = ::CompiledModel.BlockHierarchyMap.Subsystem[j].Block[k].Type
214                         %assign bname = ::CompiledModel.BlockHierarchyMap.Subsystem[j].Block[k].SLName
215 
216                         %%...IF THE BLOCK IS A RATE TRANSITION...
217                         %if btype == "RateTransition"
218 
219                             %assign input_task =  FEVAL("get_input_task_from_RT",model_name,bname)
220                             %assign output_task = FEVAL("get_output_task_from_RT",model_name,bname)
221 
222                             %%...IF THE TASK IS INVOLVED IN THE RATE TRANSITION...
223                             %if input_task == taskname || output_task == taskname
224 
225                                 %assign input_priority  = FEVAL("get_task_priority",model_name,input_task)
226                                 %assign output_priority = FEVAL("get_task_priority",model_name,output_task)
227 
228                                 %%...IF THE READER TASK HAS A HIGHER PRIORITY...
229                                 %if output_priority < input_priority
230                                     %%...DECLARE A RESOURCE TO AVOID PRIORITY INVERSION
231                                     /* Resource definition to avoid Priority Inversion */
232                                     RESOURCE = %<input_task>_%<output_task>_res;
233                                 %endif
234                             %endif
235                         %endif
236                     %endforeach
237                     %break
238                 %endif
239             %endforeach
```

Figure 5.7: *TLC Oil file - Task Resources specifications.*

Events will be specified with the name **ScheduleEvent_*taskname***, and the alarms with the name **Alarm_*taskname***. The alarm will be periodically generated, setting the **CYCLETIME** parameter to the Task sample time.

```
243             %%EVENT RELATED TO THE TASK PERIODIC ACTIVATION
244                 /* Event managed by the Task */
245                 EVENT = ScheduleEvent_%<taskname>;
246             };
247 
248             %%...ITS TASK EVENT...
249             EVENT ScheduleEvent_%<taskname> { MASK = AUTO; };
250 
251 
252             %%...ITS ALARM...
253             ALARM Alarm_%<taskname> {
254                 COUNTER = SystemTimer;
255                 ACTION = SETEVENT {
256                     TASK = %<taskname>;
257                     EVENT = ScheduleEvent_%<taskname>;
258                 };
259                 AUTOSTART = TRUE {
260                     ALARMTIME = 250;
261 
262                     %%...IMPOSING THE RELATED SAMPLE TIME.
263                     CYCLETIME = %<tasksampletime>;
264                 };
265             };
266 
267         %endif
268     %endforeach
269 };
```

Figure 5.8: *TLC Oil file - Data getting and Task specification.*

- ***mbd_task_erika_rtos.tlc***: This is the task file template; by analyzing the model RTW file, it is possible to understand whether the Tasks are independent or not, the quantity of tasks, their sample time, priority and the data dependency between them. So, considering all of these information, the C file containing the Task implementations is generated. To better understand the content of this file, it is better to proceed step by step:

  The model-specific header files are included to the source. After this, the counters needed to manage the rate transition blocks will be declared: so, for each rate transition found in the model, a 16-bits volatile variable is declared, by following the naming rule ***writer_task_reader_task*_RT_count**, and initialized to 0, considered the moment in which a data update is needed. In case of no data dependencies between Tasks, no counters will be declared here.

```
31  /* ERIKA Enterprise. */
32  #include "ee.h"
33
34  /* Autogenerated Model Header files */
35  #include "%<FcnMdlName()>.h"
36  #include "%<FcnMdlName()>_private.h"
37
38  %closefile tmpFcnBuf
39
40  %<SLibSetModelFileAttribute(cFile, "Includes", tmpFcnBuf)>
41
42  %openfile tmpFcnBuf
43
44  /* Rate Transition counting variable declarations. */
45
46  %foreach jrt = ::CompiledModel.BlockHierarchyMap.NumSubsystems
47      %if ::CompiledModel.BlockHierarchyMap.Subsystem[jrt].Type == "root"
48          %foreach krt = ::CompiledModel.BlockHierarchyMap.Subsystem[jrt].NumBlocks
49
50              %assign btype = ::CompiledModel.BlockHierarchyMap.Subsystem[jrt].Block[krt].Type
51              %assign bname = ::CompiledModel.BlockHierarchyMap.Subsystem[jrt].Block[krt].SLName
52
53              %%FOR EACH RATE TRANTISION...
54              %if btype == "RateTransition"
55                  %assign  writer_task = FEVAL("get_input_task_from_RT",model_name,bname)
56                  %assign  reader_task = FEVAL("get_output_task_from_RT",model_name,bname)
57
58                  %%...DECLARE A COUNTING VARIABLE TO MANAGE THE SHARED VARIABLE UPDATE PROCESS
59                  uint16_t volatile %<writer_task>_%<reader_task>_RT_count = 0;
60
61              %endif
62          %endforeach
63          %break
64      %endif
65  %endforeach
66  /* End of Rate Transition counting variable declarations. */
```

Figure 5.9: *TLC Task file - Includes and RT counters.*

Each Task definition, for both independent and dependent ones, starts with a common code part: the activation mechanism. Exploiting the Extended Conformance Class, as said in part 1.2.1, the task will never terminate, it just moves its status among running, ready and waiting in an endless loop. Each time its step function is executed, it will move in waiting state, waiting for the next event to be woken up. So, in order to wait for an event, the OS API function *WaitEvent* is used: thanks to it, the task moves in waiting state; when the event comes due to the generation of the alarm, it will pass to the ready state, and when possible to the running one, clearing the event for the next task execution and executing its body:

```
70  %%SEARCHING FOR TASKS
71  %foreach j = ::CompiledModel.BlockHierarchyMap.NumSubsystems
72          %if ::CompiledModel.BlockHierarchyMap.Subsystem[j].Type == "root"
73              %foreach k = ::CompiledModel.BlockHierarchyMap.Subsystem[j].NumBlocks
74                  %assign btype = ::CompiledModel.BlockHierarchyMap.Subsystem[j].Block[k].Type
75                  %assign bname = ::CompiledModel.BlockHierarchyMap.Subsystem[j].Block[k].SLName
76
77                  %%FOR EACH TASK...
78                  %if btype == "SubSystem"
79                      %assign taskname = bname
80
81                      /* Task definition: %<taskname> */
82                      uint16_t volatile %<taskname>_count;
83
84                      TASK(%<taskname>)
85                      {
86                          EventMaskType mask;
87
88                          /* Task Body */
89                          while(OSEE_TRUE)
90                          {
91                              WaitEvent(ScheduleEvent_%<taskname>);
92                              GetEvent(%<taskname>, &mask);
93
94                              if(mask & ScheduleEvent_%<taskname>){
95                                  ClearEvent(ScheduleEvent_%<taskname>);
```

Figure 5.10: *TLC Task file - Task definition and waiting mechanism.*

In case of dependent Tasks with different rates, they will be connected through a Rate Transition in the model; with its presence, Simulink will generate a schedule function that depends on the block rates; so it alternates the tasks execution depending on their rates and guarantee the data determinism as a rate transition does. This function would be nicely used, but just in the case in which the model contains only two tasks, one dependent to the other; in different cases, as for example by adding an independent task to the model or another couple of depending tasks, the scheduling function wil contain also the step function calls to the other additional tasks, so using it to be called

in a single task would be out of the concept of Real-Time tasks execution, preemption-based scheduling and all of the benefits that a RTOS can give to the System. So, in order to mantain a general rule on code generation, the Rate Transition code generation behavior (i.e. for each dependent tasks undersand when it is time to update the data and just after this let the other task to be executed) is independently generated for each couple of dependent tasks; the updating of the data is treated as a race condition, in which the reading task will wait the writing task to complete the writing operation; for this synchronization purpose, Erika Enterprise built-in Semaphores are used.

For each task, before generating the related step function call containing the model Task generated code, the template will check whether there are passive dependencies related to the Task, i.e. situations in which the Task receives data from other Tasks (*Reader Task*). In case there exist, before launching the step function the above synchronization technique has to be implemented; it depends on the involved Task rates:

- **Lower rate Writer**: the reader will down-count how much times it was executed; when reaching 0, i.e. the time in which it expects a data updating because of the execution of the writer, it will move spontaneously to the wait state by using the semaphore defined in the OIL file aimed to manage the Rate Transition between the two tasks and the counter value is restarted to a value equal to the ratio between the involved Task sample times. So, in any cases, the writer Task will be executed before the reader, ensuring data determinism for the shared variable;
- **Higher rate Writer**: the reader will wait anytime it executes, waiting for the execution of the faster writer in order to use coherent data.

```
97                      %foreach q = ::CompiledModel.BlockHierarchyMap.Subsystem[j].NumBlocks
98                          %assign btype_rt = ::CompiledModel.BlockHierarchyMap.Subsystem[j].Block[q].Type
99                          %assign bname_rt = ::CompiledModel.BlockHierarchyMap.Subsystem[j].Block[q].SLName
100
101                         %%FOR EACH RATE TRANSITION...
102                         %if btype_rt == "RateTransition"
103
104                             %%FIND ALL THE SITUATIONS THAT INVOLVES THE TASK AS A READER
105                             %assign tmp_dst_task = FEVAL("get_output_task_from_RT",model_name,bname_rt)
106                             %if tmp_dst_task == taskname
107
108                                 %%INPUT AND OUTPUT TASK
109                                 %assign writer_task = FEVAL("get_input_task_from_RT",model_name,bname_rt)
110                                 %assign reader_task = taskname
111
112                                 %%GET THE TASK PERIODS
113                                 %assign task_period_writer  = CAST("Number",FEVAL("get_task_period",model_name,writer_task))
114                                 %assign task_period_reader = CAST("Number",FEVAL("get_task_period",model_name,reader_task))
115
116                                 %if task_period_writer > task_period_reader
117                                     %%CASE: SLOWER WRITER, FASTER READER, READER POINT OF VIEW
118
119                                     %assign tasks_time_ratio = task_period_writer / task_period_reader
120                                     /* Data dependency between %<writer_task> and %<reader_task>; Rate Transition: %<bname_rt> */
121
122                                     if(%<writer_task>_%<reader_task>_RT_count == 0)
123                                     {
124                                         %<writer_task>_%<reader_task>_RT_count = %<CAST("Number",tasks_time_ratio - 1)>;
125                                         /* Critical section: wait for any updates on data */
126                                         WaitSem(&%<writer_task>_%<reader_task>_sem);
127                                     }
128                                     else
129                                     {
130                                         %<writer_task>_%<reader_task>_RT_count--;
131                                     }
132                                 %else
133                                     %%CASE: FASTER WRITER, SLOWER READER, READER POINT OF VIEW
134                                     /* Data dependency between %<writer_task> and %<reader_task>; Rate Transition: %<bname_rt> */
135                                     WaitSem(&%<writer_task>_%<reader_task>_sem);
136                                 %endif
137                             %endif
```

Figure 5.11: *TLC Task file - Shared Variables Management, Reader point of view.*

In case of a independent Task, this template piece of code will not generate any code.

While analyzing all the Rate Transition Blocks connected to the Task, if there exist situations in which the Task generates data for an higher-priority one, in order to avoid the Priority Inversion Problem, the resource OS object is exploited to let the Task temporary changing its priority to the highest priority between Tasks sharing that resource. To do this, the function *GetResource* is used. The Operating System knows the Tasks that share that resource from the Oil file, in which each of them has its own resource list specified into the Task specification.

```
142                                 %assign writer_task = FEVAL("get_input_task_from_RT",model_name,bname_rt)
143                                 %assign reader_task = FEVAL("get_output_task_from_RT",model_name,bname_rt)
144                                 %assign task_period_writer  = CAST("Number",FEVAL("get_task_period",model_name,writer_task))
145                                 %assign task_period_reader = CAST("Number",FEVAL("get_task_period",model_name,reader_task))
146
147
148                                 %%IN CASE THE TASK IS A WRITER, LOCK THE RESOURCES FOR THE SHARED VARIABLES
149                                 %if writer_task == taskname &&  task_period_writer > task_period_reader
150                                     /* Rise the priority to the maximum priority between tasks sharing this resource */
151                                     GetResource(%<writer_task>_%<reader_task>_res);
152                                 %endif
153
154                             %endif
155                         %endforeach
```

Figure 5.12: *TLC Task file - Resource Locking for priority rising.*

In any cases, after having waited for the opportune data to be updated, getting resouces or just because an independent Task is analyzed, the step function call follows. A counter to keep track of how much times the task was executed is incremented.

```
157                     /* call MBSD auto-generated code step function */
158                     %<FcnMdlName()>_%<taskname>();
159
160                     /* Increment execution Counter */
161                     %<taskname>_count++;
```

Figure 5.13: *TLC Task file - Step function call.*

As seen, The task names and the opportune initialize, step and terminate function names will depend on the non-virtual Subsystem names and the model name.

After the step function execution, if some active dependencies involves the analyzed Task, i.e. situations in which it produces data for other Tasks (*writer Task*), these data will be updated accordingly to the relation between the sample rates:

- **Lower rate Writer**: it will update the data and then freeing the semaphore by using the *PostSem* OS API, independently on the reader rate, because it expects that the reader Task will stop its execution moving in waiting state each time the writer has to be executed. The previously locked Resource is released too, in order to restore the original Task priority;
- **Higher rate Writer**, it will take care of down-counting each time it is executed, in order to understand when to update the data, use the *PostSem* function and restarting the counter to a value that expresses the ratio between the two sample times.

```
                        %foreach q = ::CompiledModel.BlockHierarchyMap.Subsystem[j].NumBlocks
                            %assign btype_rt = ::CompiledModel.BlockHierarchyMap.Subsystem[j].Block[q].Type
                            %assign bname_rt = ::CompiledModel.BlockHierarchyMap.Subsystem[j].Block[q].SLName

                            %%FOR EACH RATE TRANSITION...
                            %if btype_rt == "RateTransition"

                                %%FIND ALL THE SITUATIONS THAT INVOLVES THE TASK AS A WRITER
                                %assign tmp_src_task = FEVAL("get_input_task_from_RT",model_name,bname_rt)
                                %if tmp_src_task == bname
                                    %%GET THE VARIABLE NAMES
                                    %assign RT_input_var  = FEVAL("get_RT_input",model_name,bname_rt)
                                    %assign RT_output_var = FEVAL("get_RT_output",model_name,bname_rt)
                                    %%INPUT AND OUTPUT TASK
                                    %assign writer_task = bname
                                    %assign  reader_task = FEVAL("get_output_task_from_RT",model_name,bname_rt)
                                    %%GET THE TASK PERIODS
                                    %assign task_period_writer  = CAST("Number",FEVAL("get_task_period",model_name,writer_task))
                                    %assign task_period_reader = CAST("Number",FEVAL("get_task_period",model_name,reader_task))

                                    %if task_period_writer > task_period_reader
                                        %%CASE: SLOWER WRITER, FASTER READER, WRITER POINT OF VIEW

                                        /* Critical section: Get the temporary priority, update the variable */
                                        %<FcnMdlName()>_B.%<RT_output_var> = %<FcnMdlName()>_B.%<RT_input_var>;

                                        /* Independently, Post on the semaphore to the faster dependent Task */
                                        PostSem(&%<writer_task>_%<reader_task>_sem);

                                        /* Release the resource, so come back to the initial priority */
                                        ReleaseResource(%<writer_task>_%<reader_task>_res);

                                    %else
                                        %%CASE: FASTER WRITER, SLOWER READER, WRITER POINT OF VIEW

                                        %assign tasks_time_ratio = task_period_reader / task_period_writer

                                        if(%<writer_task>_%<reader_task>_RT_count == 0) {

                                            %<writer_task>_%<reader_task>_RT_count = %<CAST("Number",tasks_time_ratio - 1)>;

                                            /* Critical section: update the shared variable value and Post on the Semaphore */
                                            %<FcnMdlName()>_B.%<RT_output_var> = %<FcnMdlName()>_B.%<RT_input_var>;
                                            PostSem(&%<writer_task>_%<reader_task>_sem);
                                        }
                                        else {
                                            %<writer_task>_%<reader_task>_RT_count--;
                                        }
                                    %endif
                                %endif
                            %endif
                        %endforeach
                                }
                            }
                        };
                    %endif
                %endforeach
                %break
            %endif
    %endforeach
```

Figure 5.14: *TLC Task file - Shared Variables Management, Writer point of view.*

What is obtained is the exact behavior that Simulink provides to the Model-Based Designer when using Rate Transition blocks.

What is exploited of the standard ERT code generation is the data structure containing the declaration of the variables from the Task to the Rate Transition and from the Rate Transition to the other Task. They are declared in a structure of data signals in the *model.h* file.

- ***mbdtargettemplate_erika_rtos.tlc***: Writes all the additional model files starting from the buffers filled by the other TLC files, i.e. the OS configuration file *conf.oil* and the task file *task.c*. This file will take care of indent the new additional files.

```
%% ============================================
%%
%%   Real-Time Workshop Embedded Coder template


%selectfile NULL_FILE

%%BACKWARD COMPATIBILITY CHECKS
%<LibSetCodeTemplateComplianceLevel(2)>

%if CodeFormat == "Embedded-C"
    %%CREATE TASK.C FILE USING THE TEMPLATE
    %<SLibRAppIDTask("task")>

    %%CREATE CONF.OIL FILE USING THE TEMPLATE
    %<SLibRAppIDOil("conf.oil")>
%endif



%%INDENT NON STANDARD FILES
%if GenerateSampleERTMain
  %<SLibIndentFile("task.c","")>
  %%<SLibIndentFile("conf.oil","")>
%endif
```

Figure 5.15: *TLC Target Template file.*

In summary, the generated code lets the tasks wait for the periodic event generated by the alarm once per task period and call the step functions generated by Embedded Coder, and in case of depending tasks, the mechanism to update the data in a secure manner is also involved.

In order to allow Simulink to find the System Target File and by consequence to allow the user to choose it from the list of the possible System Target Files, all of them were placed in the same directory in the following MATLAB path: '*$matlab_root/$matlab_version/rtw/c/erika_rtos/*'.

The custom Erika Enterprise System Target File is now able to be chosen in the Embedded Coder configuration options before staring the code generation process:

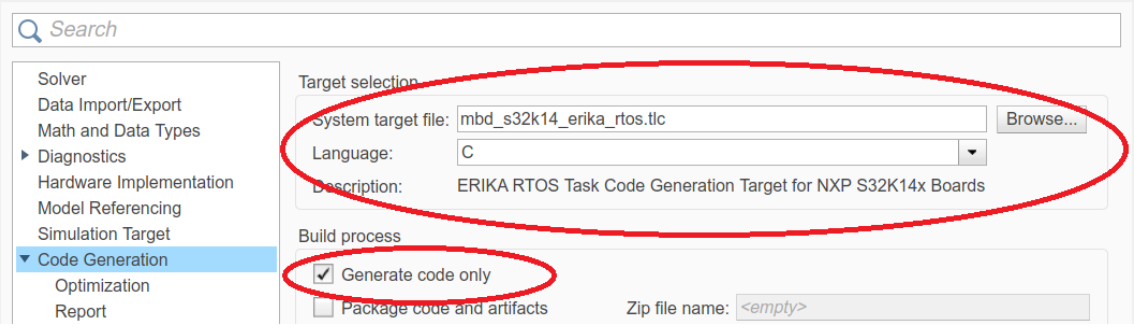


Figure 5.16: *Erika Enterprise RTOS Task TLC file selection.*

Due to the purpose of simple code generation, no makefile has to be generated; so, only the *Generate code only* checkbox has to be set.

Chapter 6

Code Generation Tests

The following model examples aims to let the Model-Based Designer understand how to use the Simulink Library dealing with some common modelling situations. All of them were tested in order to verify whether the code behavior well met the model specifications running the specified Task-level application on the target Board.

6.1 A first example: single Task

As example of code generation, let's consider the following simple specifications:

A task has to use one of the target board ADCs to convert the value of the board potentiometer each 500 ms: whenever the converted value is greater than the imposed threshold of 2000, the embedded blue LED is turned on, otherwise is turned off. Let's proceed step by step:

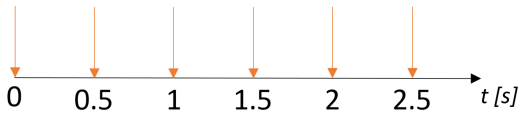


Figure 6.1: *Task Activation Alarm Timing Diagram.*

1. Create a new Simulink Model; here, create an empty Subsystem and call it "Task_trial":

Figure 6.2: *Empty Task Subsystem*

2. Modify the Subsystem parameters in order to let it be treated as atomic unit, with a sample time of 500 ms. To do so, right click on the subsystem and select ***Block Parameters (Subsystem)***:

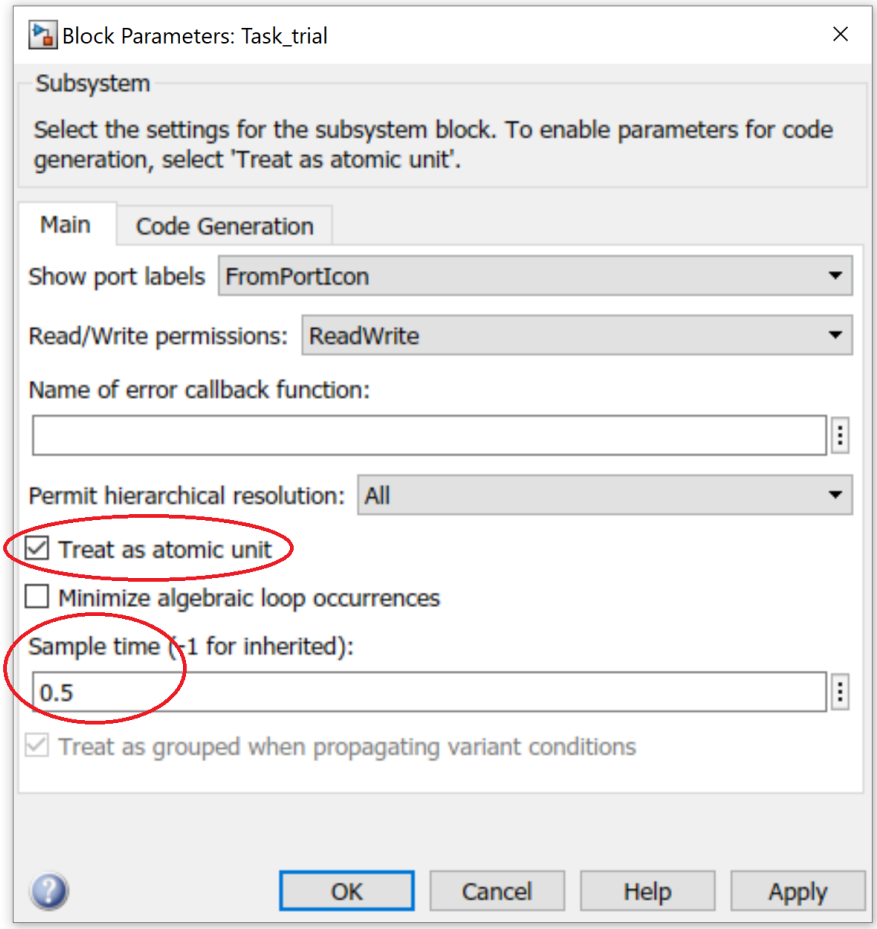


Figure 6.3: *Subsystem Main Parameters*

For code generation reasons,set the function packaging to *Nonreusable function* and Function name option to *Use Subsystem name* on the Code Generation tab:

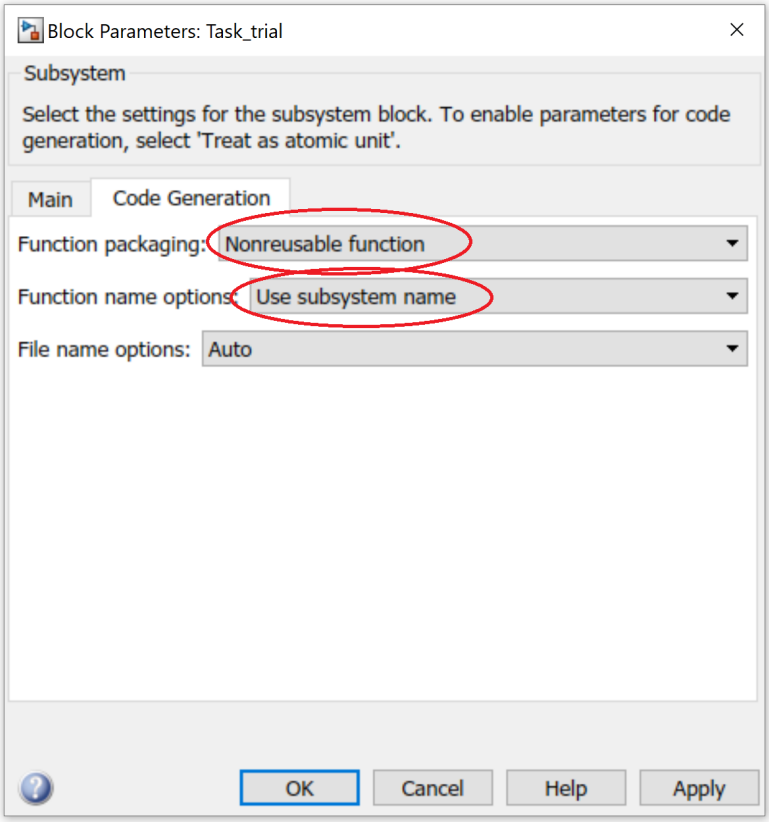


Figure 6.4: *Subsystem Code Generation Parameters*

3. After this first configuration phase, it is time to fill the Task Subsystem with the Erika Enterprise Simulink Library Blocks. Instantiate an initialize Simulink block, and delete everything into it except for the event listener:

Figure 6.5: *Initialize Block*

4. Inside the Initialize block, add the adc_setup block configured for setting the ADC0 by using its block mask:

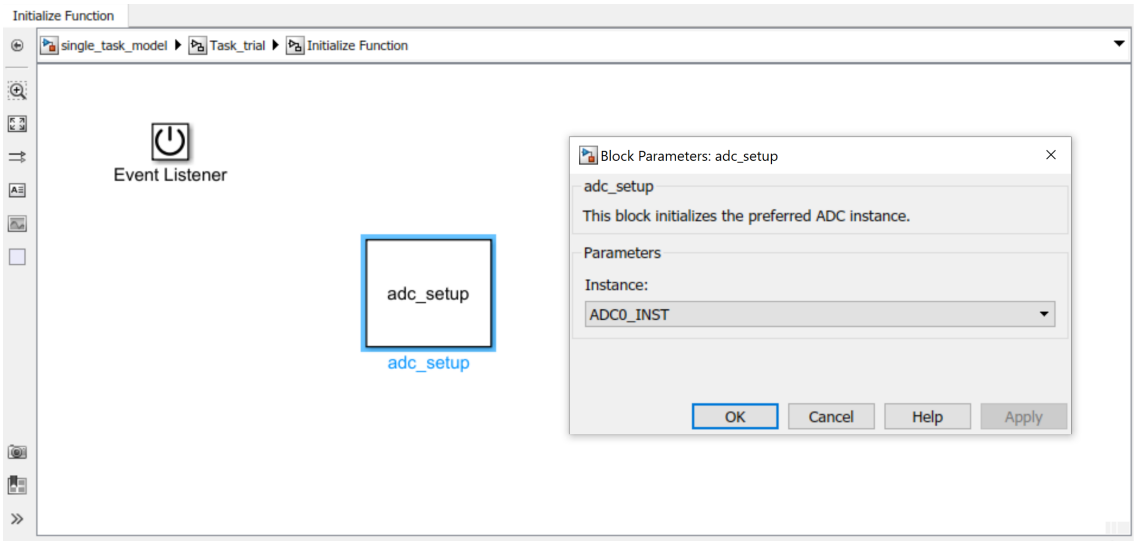


Figure 6.6: *adc_setup Mask configuration*

5. Inside the Initialize block, add the GPIO_port_config_output block configured for setting the blue LED as GPIO output by using its block mask:

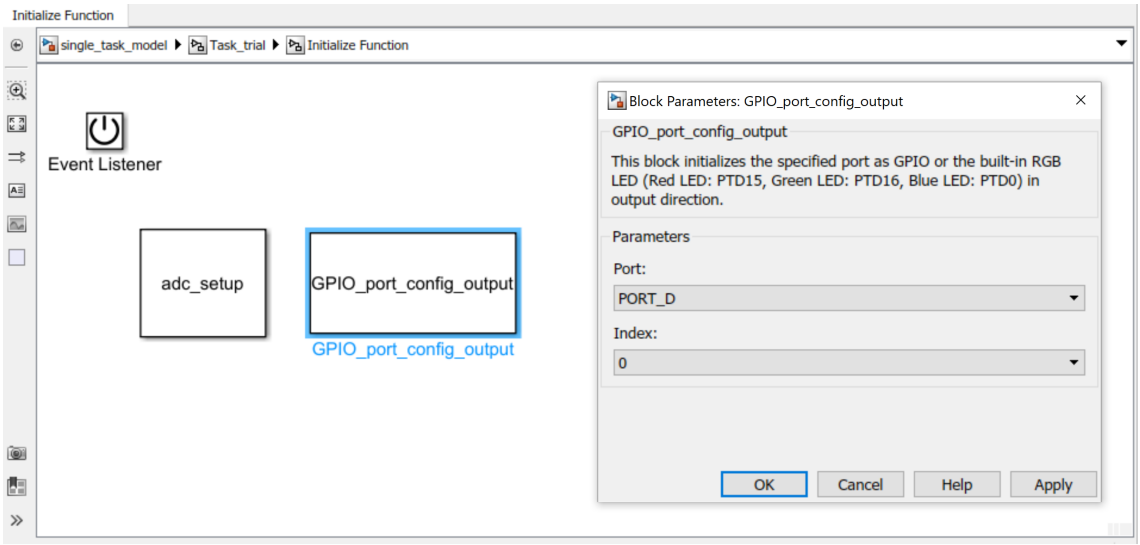


Figure 6.7: *GPIO_port_config_output Mask configuration*

6. Inside the Task Subsystem block, instantiate the blocks *adc_get_value* and *GPIO_write* from the implemented Erika Enterprise library, connect them with some other standard Simulink blocks (constants, switch) and configure them as follows:

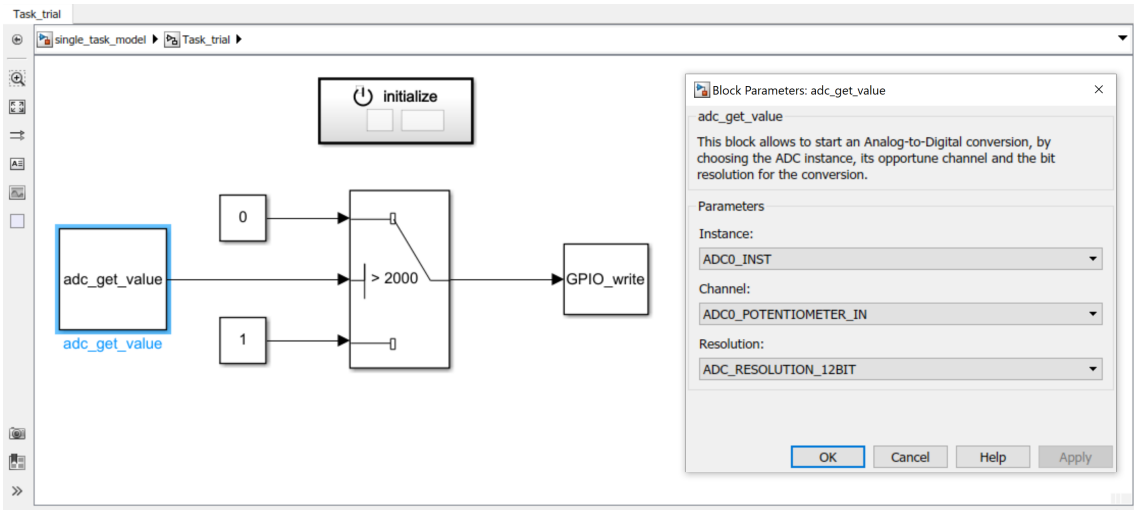


Figure 6.8: *adc_get_value block configuration*

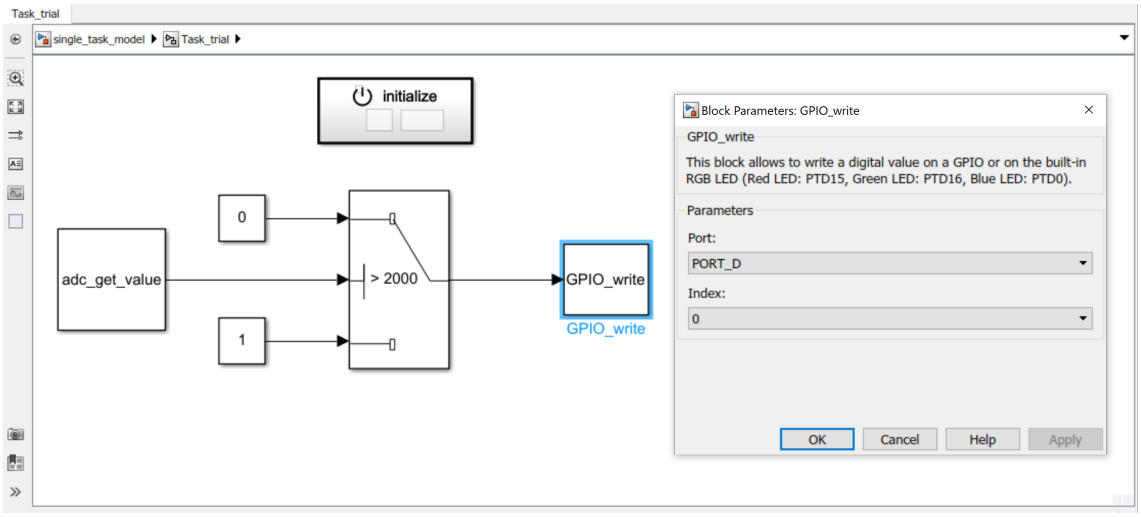


Figure 6.9: *GPIO_write block configuration*

7. Configure the Solver in discrete and Fixed-step type in the *Solver* tab of the Coder Configuration Parameters;

8. Set the System Target File as *mbd_s32k14_erika_rtos.tlc*, select *Generate code only* and deselect *Generate Makefile* in the *Code Generation* tab of the Coder Configuration Parameters;

9. Generate the code by clicking on the Embedded Coder button;

If everything is opportunely set and modeled, the code generation succeed. Let's analyze the generated code:

- Starting from the **conf.oil** file, the C file containing all of the model generated functions is included to the needed external files:

```
65  APPDATA myApp {
66    APP_SRC = "hal.c";
67    APP_SRC = "code.c";
68    APP_SRC = "task.c";
69    APP_SRC = "single_task_model.c";
70  };
```

Figure 6.10: *conf.oil file external files includes*

It also includes the Task configurations and the related Event and Alarm to execute it periodically with the specified sample time:

```
85  TASK Task_trial {
86    PRIORITY = 1;
87    ACTIVATION = 1;
88    SCHEDULE = FULL;
89    AUTOSTART = TRUE;
90    STACK = PRIVATE {
91      SIZE = 512;
92    };
93
94    EVENT = ScheduleEvent_Task_trial;
95  };
96
97  EVENT ScheduleEvent_Task_trial {
98    MASK = AUTO;
99  };
100
101 ALARM Alarm_Task_trial {
102   COUNTER = SystemTimer;
103   ACTION = SETEVENT {
104     TASK = Task_trial;
105     EVENT = ScheduleEvent_Task_trial;
106   };
107
108   AUTOSTART = TRUE {
109     ALARMTIME = 250;
110     CYCLETIME = 500;
111   };
112 };
```

Figure 6.11: *conf.oil: task, alarm and event definition.*

- The **task.c** will contain the following task implementation:

```
20 /* ERIKA Enterprise. */
21 #include "ee.h"
22
23 /* Autogenerated Model Header files */
24 #include "single_task_model.h"
25 #include "single_task_model_private.h"
26
27 /* Task_trial execution counter */
28 uint16_t volatile Task_trial_count;
29
30 /* Task_trial body: here the step function is called */
31 TASK(Task_trial)
32 {
33   EventMaskType mask;
34   while (OSEE_TRUE) {
35     WaitEvent(ScheduleEvent_Task_trial);
36     GetEvent(Task_trial, &mask);
37     if (mask & ScheduleEvent_Task_trial) {
38       ClearEvent(ScheduleEvent_Task_trial);
39
40       /* call MBSD auto-generated code step function */
41       single_task_model_Task_trial();
42     }
43
44     ++Task_trial_count;
45   }
46
47   /* Terminate TASK Task_trial */
48   TerminateTask();
49 }
50
51 ;
```

Figure 6.12: *task.c - Task definition.*

As can be read, it will wait for the opportune event, triggered from the RTOS each 500 ms as specified in the OIL file, and then the opportune step function will be called. After this, it will wait again for the next event, and so on and so forth.

- About the **single_task_model.c** file, the function that will be called are *single_task_mod_Task_trial_Init()*, corresponding to the initialize function, abd *single_task_model_Task_trial()*, corresponding to the step function.

```
28 void single_task_mod_Task_trial_Init(void)
29 {
30   /* SystemInitialize for Atomic SubSystem: '<S1>/Initialize Function' */
31
32   /* S-Function (GPIO_port_config_output): '<S2>/GPIO_port_config_output' */
33   GPIO_port_config_output(((uint8_T)4U), ((uint8_T)1U));
34
35   /* S-Function (adc_setup): '<S2>/adc_setup' */
36   adc_setup(1U);
37
38   /* End of SystemInitialize for SubSystem: '<S1>/Initialize Function' */
39 }
```

Figure 6.13: *Model Init function.*

```
41 /* Output and update for atomic system: '<Root>/Task_trial' */
42 void single_task_model_Task_trial(void)
43 {
44   uint8_T rtb_Switch;
45
46   /* Switch: '<S1>/Switch' incorporates:
47    *  S-Function (adc_get_value): '<S1>/adc_get_value'
48    */
49   rtb_Switch = (uint8_T)(((uint16_T)adc_get_value(1U, ((uint8_T)17U), ((uint8_T)
50     2U))) <= 2000);
51
52   /* S-Function (GPIO_write): '<S1>/GPIO_write' */
53   GPIO_write(4U, ((uint8_T)1U), rtb_Switch);
54 }
```

Figure 6.14: *Model Step function.*

6.2 Depending Tasks with different Rates

As exention of the first example, it will be taken in consideration a model that describes a situation in which there is a data dependency between two tasks with different rates. Let's have a look to the following specifications:

A task has to use one of the target board ADCs to convert the value of the board potentiometer each 500 ms: whenever the converted value is greater than the imposed threshold of 2000, the embedded red LED is turned on, otherwise is turned off. Additionally, a second Task will update the blue LED status accordingly to value of the red LED each 2 s.

So, for each second task call, the first one will be called four times.

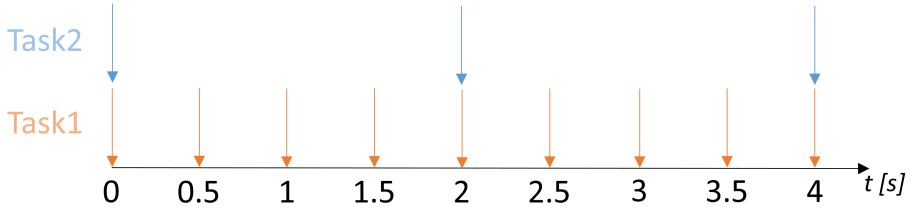


Figure 6.15: *Tasks Activation Alarm Timing Diagram.*

To ensure a deterministic data transfer, a Rate Transition block between the two tasks is needed. The model appears as follow:

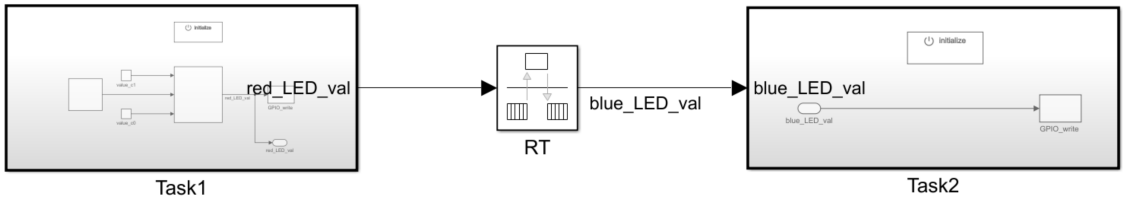


Figure 6.16: *Different Rates Tasks Model.*

With the custom STF, the Rate Transition block presence will generate the emulation of the ERT scheduler function generated by Embedded Coder that counts, for each period corresponding to the minimum between the two task periods, how much times the slower task step function is called; whenever the counter reaches a value equal to the division between the higher task period and the lower one, the data is updated and the semaphore will signal the slower task, so it will be executed and the counter restarts.

Considering that the Task periods are known from the beginning, static analysis like this can be done;

In this case, the faster task is the writer one: so, it will take care of counting the periods and update the data.

```
37 TASK(Task1)
38 {
39   EventMaskType mask;
40   while (OSEE_TRUE) {
41     WaitEvent(ScheduleEvent_Task1);
42     GetEvent(Task1, &mask);
43     if (mask & ScheduleEvent_Task1) {
44       ClearEvent(ScheduleEvent_Task1);
45 
46       /* call MBSD auto-generated code step function */
47       RT_model_Task1();
48       if (Task1_RT_count == 0) {
49         Task1_RT_count = 3;
50 
51         /* Critical section: update the shared variable value and Post on the Semaphore */
52         RT_model_B.blue_LED_val = RT_model_B.red_LED_val;
53         PostSem(&Task1_Task2_sem);
54       } else
55         Task1_RT_count--;
56     }
57 
58     ++Task1_count;
59   }
60 
61   /* Terminate TASK Task1 */
62   TerminateTask();
63 }
64 ;
```

Figure 6.17: *Writer Task (faster).*

As mentioned on the chapter regarding the System Target File, in case of dependencies between tasks with different rates the TLC files will generate a semaphore declaration inside the OIL configuration file; in order to handle its reference, it was decided to assign its name depending on the involved task names.

```
48     USEEXTENSIONAPI = TRUE {
49       /* Semaphore for Rate Transition between Tasks Task1 and Task2 */
50       SEMAPHORE = DEFAULT { NAME = "Task1_Task2_sem";
51         COUNT= 0;
52       };
53     };
```

Figure 6.18: *OIL Semaphore specification.*

```
68  TASK(Task2)
69  {
70    EventMaskType mask;
71    while (OSEE_TRUE) {
72      WaitEvent(ScheduleEvent_Task2);
73      GetEvent(Task2, &mask);
74      if (mask & ScheduleEvent_Task2) {
75        ClearEvent(ScheduleEvent_Task2);
76
77        /* Critical section: wait for any updates on data */
78        WaitSem(&Task1_Task2_sem);
79
80        /* call MBSD auto-generated code step function */
81        RT_model_Task2();
82        ++Task2_count;
83      }
84    }
85
86    /* Terminate TASK Task2 */
87    TerminateTask();
88  }
89  ;
```

Figure 6.19: *Reader Task (slower).*

As can be seen from the Tasks definitions and step functions, they use a struct containing the input and output variables to exchange the data: this is what we additionally exploit about the ERT code generation. The struct and the instantiation of the struct are contained in the model header file.

```
40 /* Block signals (default storage) */
41 typedef struct {
42   uint8_T blue_LED_val;                /* '<Root>/RT' */
43   uint8_T red_LED_val;                 /* '<S1>/Switch' */
44 } B_RT_model_T;
45

62 /* Block signals (default storage) */
63 extern B_RT_model_T RT_model_B;
64
```

Figure 6.20: *Data Communication Struct.*

```
62 /* Output and update for atomic system: '<Root>/Task1' */
63 void RT_model_Task1(void)
64 {
65   /* Switch: '<S1>/Switch' incorporates:
66    *  S-Function (adc_get_value): '<S1>/adc_get_value'
67    */
68   RT_model_B.red_LED_val = (uint8_T)(((uint16_T)adc_get_value(1U, ((uint8_T)17U),
69     ((uint8_T)2U))) <= 2000);
70
71   /* S-Function (GPIO_write): '<S1>/GPIO_write ' */
72   GPIO_write(4U, ((uint8_T)16U), RT_model_B.red_LED_val);
73 }
```

Figure 6.21: *Task1 Step Function.*

```
86 /* Output and update for atomic system: '<Root>/Task2' */
87 void RT_model_Task2(void)
88 {
89   /* S-Function (GPIO_write): '<S2>/GPIO_write ' */
90   GPIO_write(4U, ((uint8_T)1U), RT_model_B.blue_LED_val);
91 }
```

Figure 6.22: *Task2 Step Function.*

6.3 Independent Tasks with different Rates

In this case, the system specifications require the presence of three different tasks, each of them with a different rate:

A task has to read each 500 ms the value of the target board button1; when it is pushed, the green LED is turn on. Another task has to use one of the target board ADCs to convert the value of the board potentiometer each 1.5 s: whenever the converted value is greater than the imposed threshold of 2000, the embedded blue LED is turned on, otherwise is turned off. A third will read each 1 s the value of the board button0 to control the red LED.

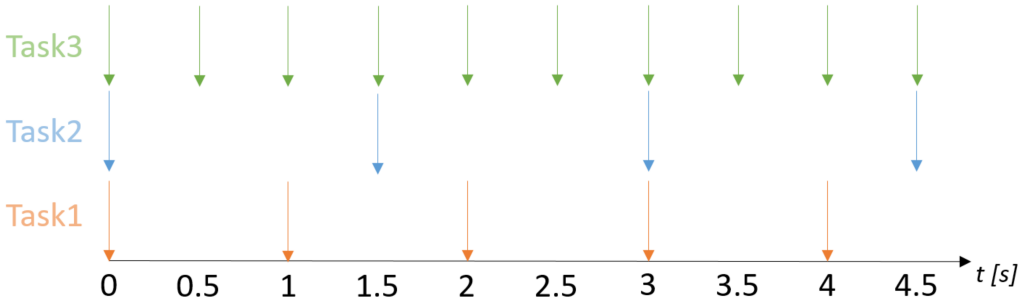


Figure 6.23: *Tasks Activation Alarm Timing Diagram.*

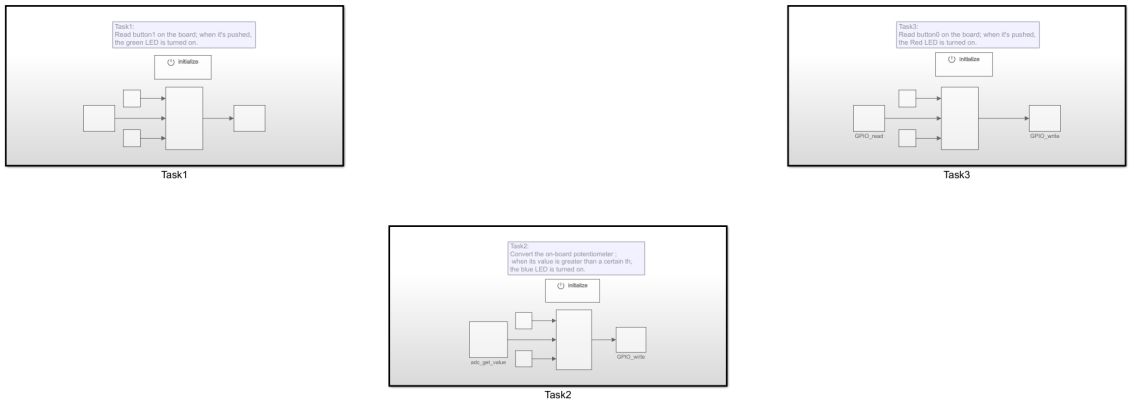


Figure 6.24: *Multi Task Subsystems.*

As can be read in the specifications, the three tasks have no dependencies, so the tasks events will be periodically generated at each individual task period and the OS scheduler will take care about managing the preemption in case of different priority. Here are presented the tasks models. The first and the third task will have the same model, but with different set mask parameters. The generated OIL file will contain the definition of three different tasks, three events and alarms with the opportune period, depending on the relating task.

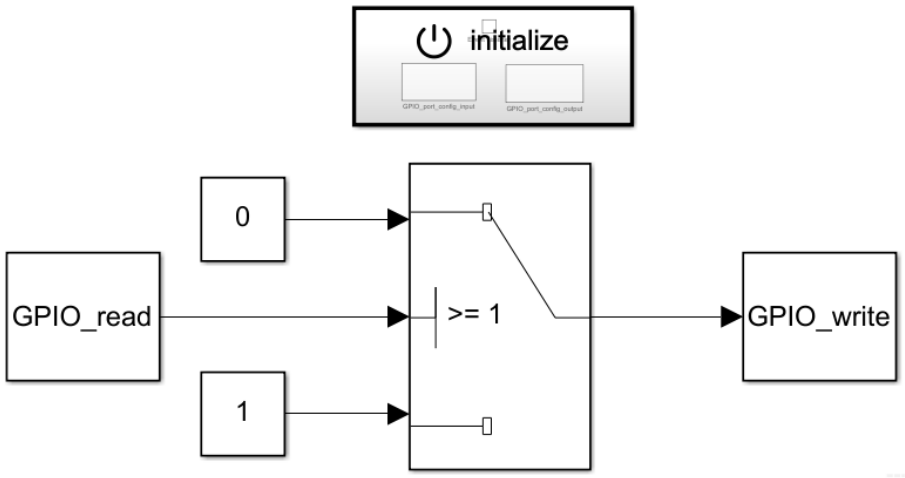


Figure 6.25: *Task 1 and 3 (Buttons read) model.*

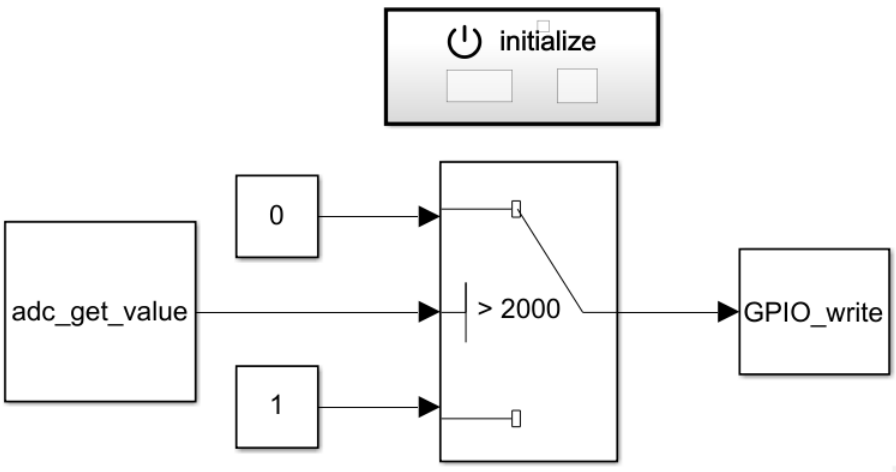


Figure 6.26: *Task 2 (Potentiometer read) model.*

6.4 Single Task using a Stateflow diagram

Sometimes, Model Based Designers may need to add some stateflow control on their models. Let's take as example the following specifications:

A task has to use one of the target board ADCs to convert the value of the board potentiometer each 10 ms. Then, an analysis of the signal has to be done: if the corresponding digital value is less than 1500, just the red LED turns on; if it is between 1500 and 3000, just the yellow LED turns on; if it is greather than 3000, just the blue LED turns on.

By using the stateflow diagram, the specifications are satisfied in a straightforward way. the diagram will receive as input the converted ADC value and will drive with 3 different signals the GPIO LED status, depending on the value.

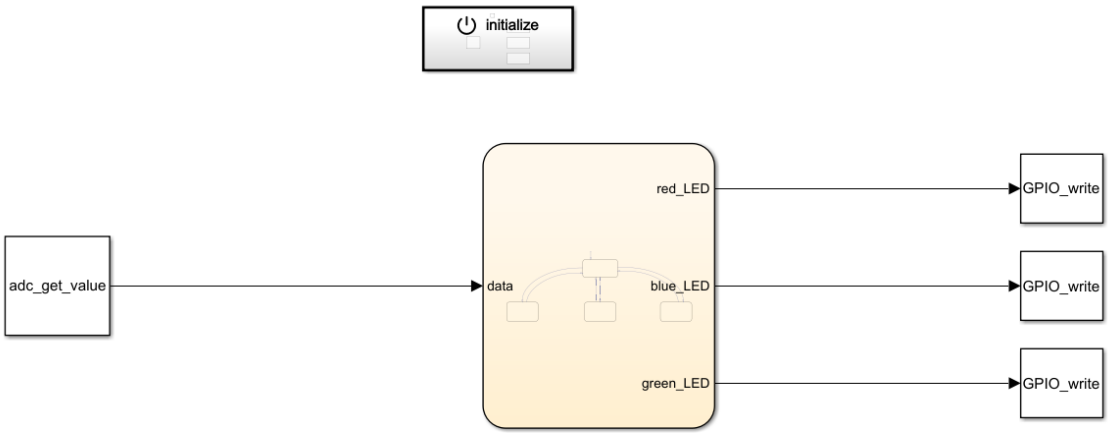


Figure 6.27: *Analog analyzer Task model.*

Each state has to clear the undesiderable LEDs (remember that in the target board writing 0 on the GPIO means to turn on the corresponding LED).

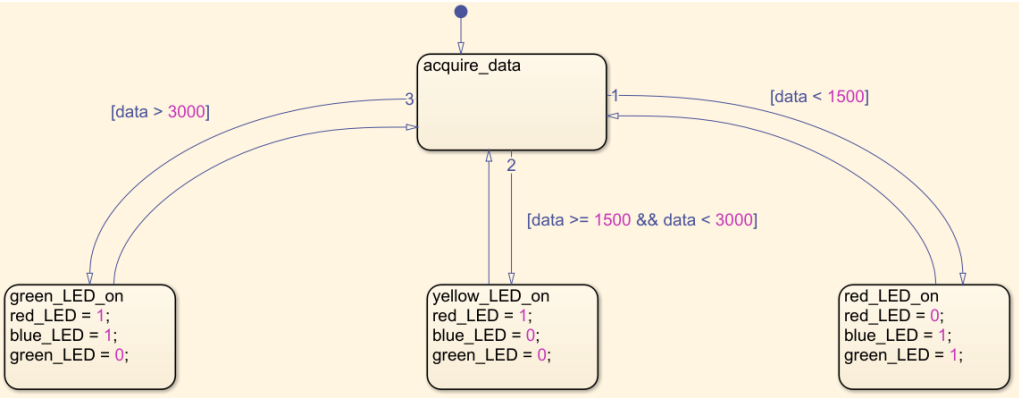


Figure 6.28: *Stateflow Diagram.*

By analyzing the generated code, it is clear that the diagram presence generates the possible states that can be reached in the model C file:

```
23 /* Named constants for Chart: '<S1>/Chart' */
24 #define stateflow_mo_IN_NO_ACTIVE_CHILD ((uint8_T)0U)
25 #define stateflow_mode_IN_yellow_LED_on ((uint8_T)4U)
26 #define stateflow_model_IN_acquire_data ((uint8_T)1U)
27 #define stateflow_model_IN_green_LED_on ((uint8_T)2U)
28 #define stateflow_model_IN_red_LED_on  ((uint8_T)3U)
```

Figure 6.29: *Diagram State definitions.*

Moreover, the model Header file contains the custom structure that the code will use to store the LEDs values and the structure needed to store the actual state:

```
41 /* Block signals (default storage) */
42 typedef struct {
43   uint8_T red_LED;                     /* '<S1>/Chart' */
44   uint8_T blue_LED;                    /* '<S1>/Chart' */
45   uint8_T green_LED;                   /* '<S1>/Chart' */
46 } B_stateflow_model_T;
47
48 /* Block states (default storage) for system '<Root>' */
49 typedef struct {
50   uint8_T is_active_c3_stateflow_model;/* '<S1>/Chart' */
51   uint8_T is_c3_stateflow_model;       /* '<S1>/Chart' */
52 } DW_stateflow_model_T;
```

Figure 6.30: *Diagram Struct definitions.*

Finally, the step function that will be called by the Task, is generated as follows. As can be seen, the ADC value is read, then this value is used to set the opportune state and LEDs values; after this, the three GPIO values are updated with the new values.

```
64  /* Output and update for atomic system: '<Root>/analog_analyzer' */
65  void stateflow_model_analog_analyzer(void)
66  {
67    /* local block i/o variables */
68    uint16_T rtb_adc_get_value;
69
70    /* S-Function (adc_get_value): '<S1>/adc_get_value' */
71    rtb_adc_get_value = adc_get_value(1U, ((uint8_T)1U), ((uint8_T)1U));
72
73    /* Chart: '<S1>/Chart' */
74    if (stateflow_model_DW.is_active_c3_stateflow_model == 0U) {
75      stateflow_model_DW.is_active_c3_stateflow_model = 1U;
76      stateflow_model_DW.is_c3_stateflow_model = stateflow_model_IN_acquire_data;
77    } else {
78      switch (stateflow_model_DW.is_c3_stateflow_model) {
79       case stateflow_model_IN_acquire_data:
80        if (rtb_adc_get_value < 1500) {
81          stateflow_model_DW.is_c3_stateflow_model = stateflow_model_IN_red_LED_on;
82          stateflow_model_B.red_LED = 0U;
83          stateflow_model_B.blue_LED = 1U;
84          stateflow_model_B.green_LED = 1U;
85        } else if ((rtb_adc_get_value >= 1500) && (rtb_adc_get_value < 3000)) {
86          stateflow_model_DW.is_c3_stateflow_model =
87            stateflow_mode_IN_yellow_LED_on;
88          stateflow_model_B.red_LED = 1U;
89          stateflow_model_B.blue_LED = 0U;
90          stateflow_model_B.green_LED = 0U;
91        } else {
92          if (rtb_adc_get_value > 3000) {
93            stateflow_model_DW.is_c3_stateflow_model =
94              stateflow_model_IN_green_LED_on;
95            stateflow_model_B.red_LED = 1U;
96            stateflow_model_B.blue_LED = 1U;
97            stateflow_model_B.green_LED = 0U;
98          }
99        }
100       break;
101
```

```
102       case stateflow_model_IN_green_LED_on:
103        stateflow_model_DW.is_c3_stateflow_model = stateflow_model_IN_acquire_data;
104        break;
105
106       case stateflow_model_IN_red_LED_on:
107        stateflow_model_DW.is_c3_stateflow_model = stateflow_model_IN_acquire_data;
108        break;
109
110       default:
111        stateflow_model_DW.is_c3_stateflow_model = stateflow_model_IN_acquire_data;
112        break;
113      }
114    }
115
116    /* End of Chart: '<S1>/Chart' */
117
118    /* S-Function (GPIO_write): '<S1>/GPIO_write' */
119    GPIO_write(1U, ((uint8_T)16U), stateflow_model_B.red_LED);
120
121    /* S-Function (GPIO_write): '<S1>/GPIO_write1' */
122    GPIO_write(1U, ((uint8_T)17U), stateflow_model_B.green_LED);
123
124    /* S-Function (GPIO_write): '<S1>/GPIO_write2' */
125    GPIO_write(1U, ((uint8_T)1U), stateflow_model_B.blue_LED);
126  }
```

Figure 6.31: *Model Step Function.*

6.5 Multiple dependent and independent Tasks, Different Rates

A generic case that can be described by a model implies the presence of independent and dependent tasks, each of them with its own rate, maybe different to the others. Let's have a look to the following specifications:

A Task has to read the value of the on-board button0 each each 500 ms; another Task will use the Button0 value to turn off the green LED, each 1500 ms. In the meanwhile, a Task has to read the value of the on-boad Button1 each 1000 ms, and another one will use the Button1 value to turn off the red LED each 250 ms. In parallel, a Task has to take care of toggling the blue LED each 250 ms.

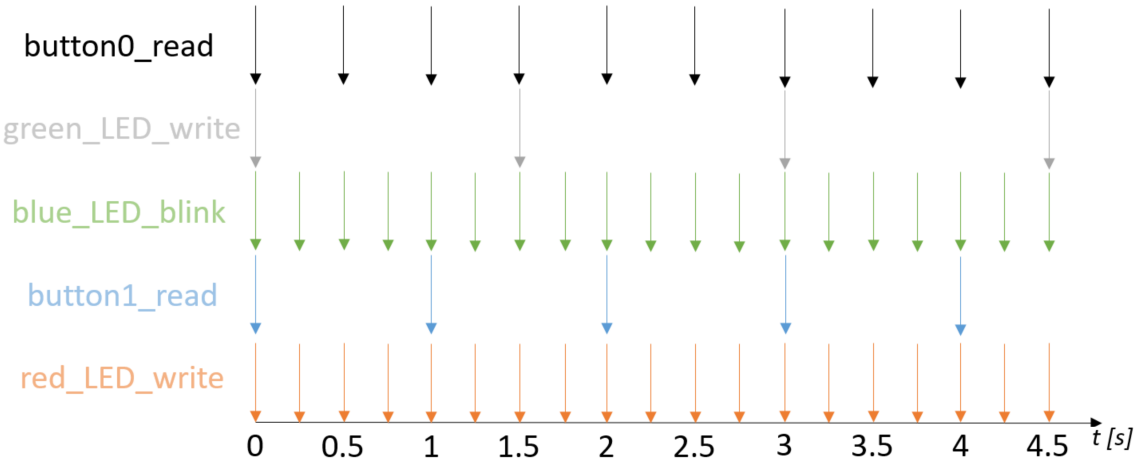


Figure 6.32: *Task Activation Alarm Timing Diagram* .

By imposing a Rate Monotonic Scheluing behavior, each Task will have its own priority level, pedending on its cycle period. This will not affect the data determinism, neither Priority Inversion Solution, due to the fact that the OS object *Resource* is used.

The situation described above can be moleded as two pairs of dependent tasks that use a Rate Transition to transfer data to the opportune one, and a single independent task, that toggles the Blue LED without caring about the other ones. A pair of the dependent ones describes the situation in which the writer task is slower than the reader, the other one describes the opposite situation, i.e. the writer task is faster than the reader. So, the following Simulink model describes the specifications:

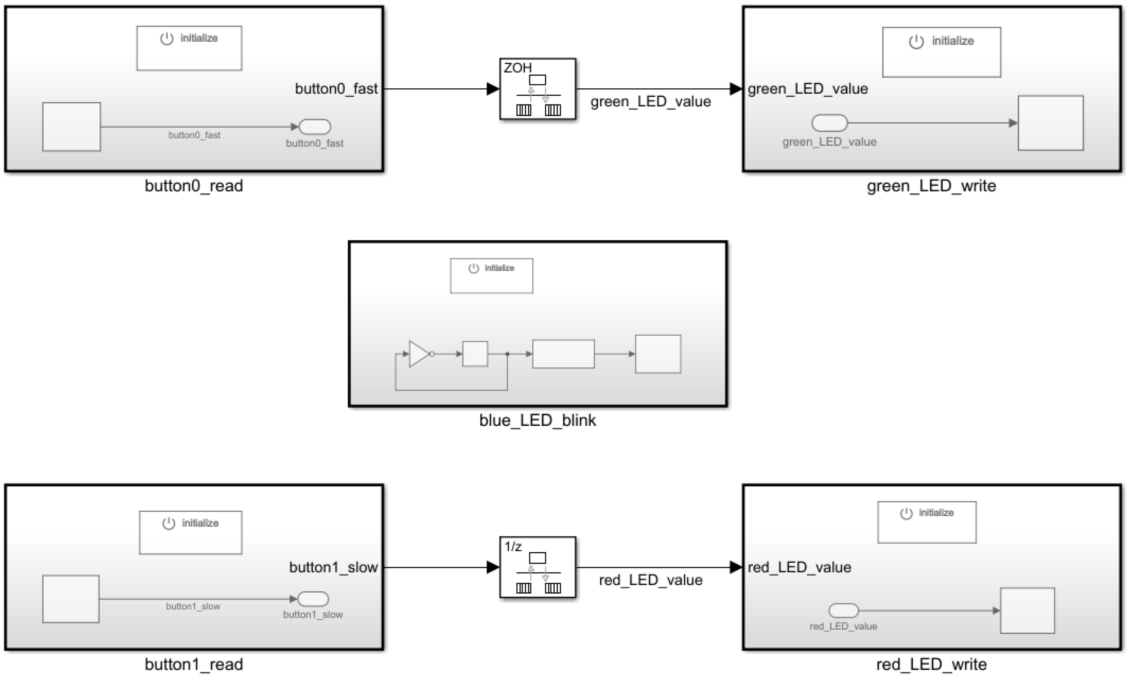


Figure 6.33: *Generic Model.*

The independent task model contains a memory block, that will be responsible to mantain the previous LED value in order to toggle it; this is translated as two variable declarations during code generation:

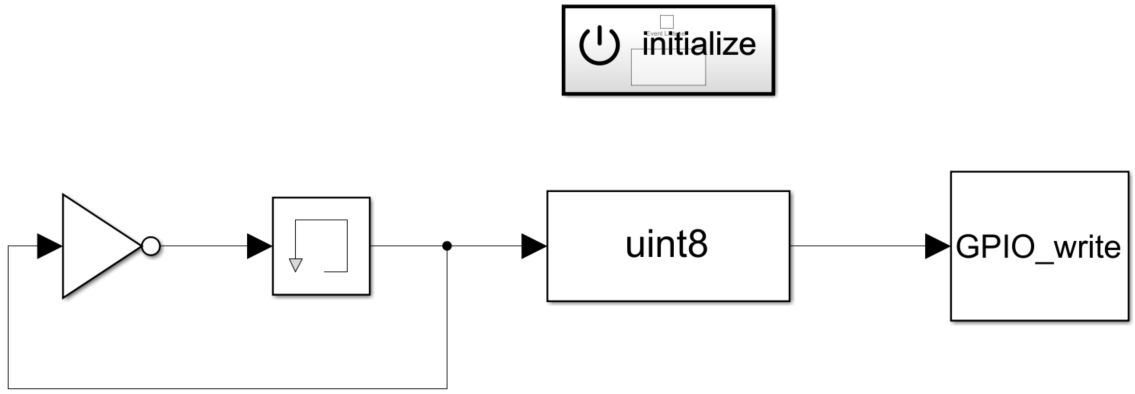


Figure 6.34: *blue_LED_blink model.*

```
73  void Generic_5_Tasks_blue_LED_blink(void)
74  {
75    uint8_T rtb_DataTypeConversion;
76
77    /* DataTypeConversion: '<S1>/Data Type Conversion' incorporates:
78     *  Memory: '<S1>/Memory'
79     */
80    rtb_DataTypeConversion = Generic_5_Tasks_DW.Memory_PreviousInput;
81
82    /* S-Function (GPIO_write): '<S1>/GPIO_write' */
83    GPIO_write(4U, ((uint8_T)1U), rtb_DataTypeConversion);
84
85    /* Update for Memory: '<S1>/Memory' incorporates:
86     *  Logic: '<S1>/NOT'
87     */
88    Generic_5_Tasks_DW.Memory_PreviousInput =
89      !Generic_5_Tasks_DW.Memory_PreviousInput;
90  }
```

Figure 6.35: *blue_LED_blink Step Function.*

After setting all of the Task periods and the configuration for code generation, what is generated is the already analyzed OIL configuration Tasks, Semaphores, Alarms and Events definitions by using the opportune names and periods; task.c will contain the definition of the independent task, as seen on the other examples, the definition of the two dependent tasks with the faster writer, as handled in the second example (*Depending Tasks with different Rates*) and introduces the case with a slower writer and faster reader:

```
101 TASK(button1_read)
102 {
103   EventMaskType mask;
104
105   /* Task Body */
106   while (OSEE_TRUE) {
107     WaitEvent(ScheduleEvent_button1_read);
108     GetEvent(button1_read, &mask);
109     if (mask & ScheduleEvent_button1_read) {
110       ClearEvent(ScheduleEvent_button1_read);
111
112       /* Rise the priority to the maximum priority between tasks sharing this resource */
113       GetResource(button1_read_red_LED_write_res);
114
115       /* call MBSD auto-generated code step function */
116       Generic_5_Tasks_button1_read();
117
118       /* Increment execution Counter */
119       button1_read_count++;
120
121       /* Critical section: Get the temporary priority, update the variable */
122       Generic_5_Tasks_B.red_LED_value = Generic_5_Tasks_B.button1_slow;
123
124       /* Independently, Post on the semaphore to the faster dependent Task */
125       PostSem(&button1_read_red_LED_write_sem);
126
127       /* Release the resource, so come back to the initial priority */
128       ReleaseResource(button1_read_red_LED_write_res);
129     }
130   }
131 } ;
```

Figure 6.36: *button1_read Task (slower writer).*

As can be seen, the writer task will not wait any events to update the task; if the two tasks arrive together, the reader will have to wait for the second one in any cases, so an implicit prioritization comes (i.e. the writer task is executed firstly, by updating the shared data and let the reader task continue, as the Rate Transition behavior imposes). During the writer Task Execution, as said in part 5.2,

its priority is rised to the highest priority among the Tasks sharing the produced data, by locking the shared Resource. After updating the data, the Resource is released and the reader will be executed without waiting the writer until a new write comes, and so on.

```
164 TASK(red_LED_write)
165 {
166   EventMaskType mask;
167
168   /* Task Body */
169   while (OSEE_TRUE) {
170     WaitEvent(ScheduleEvent_red_LED_write);
171     GetEvent(red_LED_write, &mask);
172     if (mask & ScheduleEvent_red_LED_write) {
173       ClearEvent(ScheduleEvent_red_LED_write);
174
175       /* Data dependency between button1_read and red_LED_write; Rate Transition: Rate Transition2 */
176       if (button1_read_red_LED_write_RT_count == 0) {
177         button1_read_red_LED_write_RT_count = 3;
178
179         /* Critical section: wait for any updates on data */
180         WaitSem(&button1_read_red_LED_write_sem);
181       } else {
182         button1_read_red_LED_write_RT_count--;
183       }
184
185       /* call MBSD auto-generated code step function */
186       Generic_5_Tasks_red_LED_write();
187
188       /* Increment execution Counter */
189       red_LED_write_count++;
190     }
191   }
192 };
```

Figure 6.37: *red_LED_write Task(faster reader).*

The faster reader will wait each time it expects a data update event, as a Rate Transition presence would have imposed. In this way, the data determinism is fully respected. The follwing timing diagram gives a better idea about the synchronization between tasks:

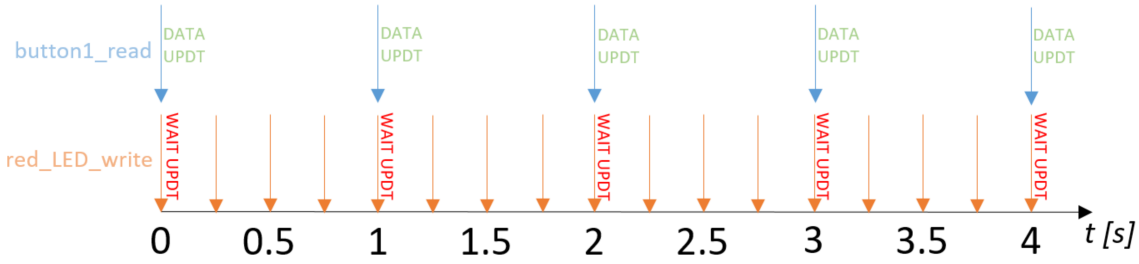


Figure 6.38: *red_LED_write Task Synchronization.*

6.6 Multi-data dependencies, Dependency chains and independent Tasks

The most generic case is presented in the following. The proposed specifications will imply indpendent Tasks, dependent Tasks with writer with higher and lower priorities with respect to their reader Tasks, and data dependency chains:

A Task has to blink the on-board blue LED each 250 ms; in the meanwhile, a Task has to read the value of the on-board button0 and another one has to read the value of the on-board potentiometer, respectively each 200 and 500 ms: these two produced data will be read by a Task each 1000 ms, that will take care of updating the red LED status with the button value if the potentiometer converted value is greater than 2000, otherwise the digital 1 will be written on the LED. Each 100 ms, a Task has to update the value of the green LED by copying the value of the red LED. In parallel, a Task will generate a PWM signal, by increasing the Duty Cycle of 25% each 1000 ms, restarting to the starting value of 25% when reaching the 100% Duty Cycle.

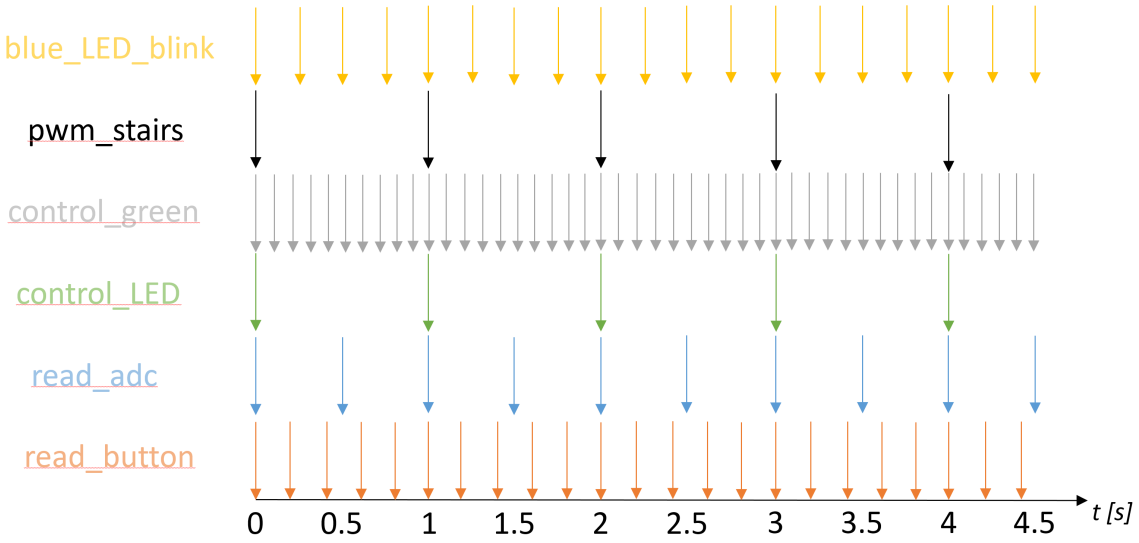


Figure 6.39: *Tasks Activation Alarm Timing Diagram.*

The corresponding model appears as follows:

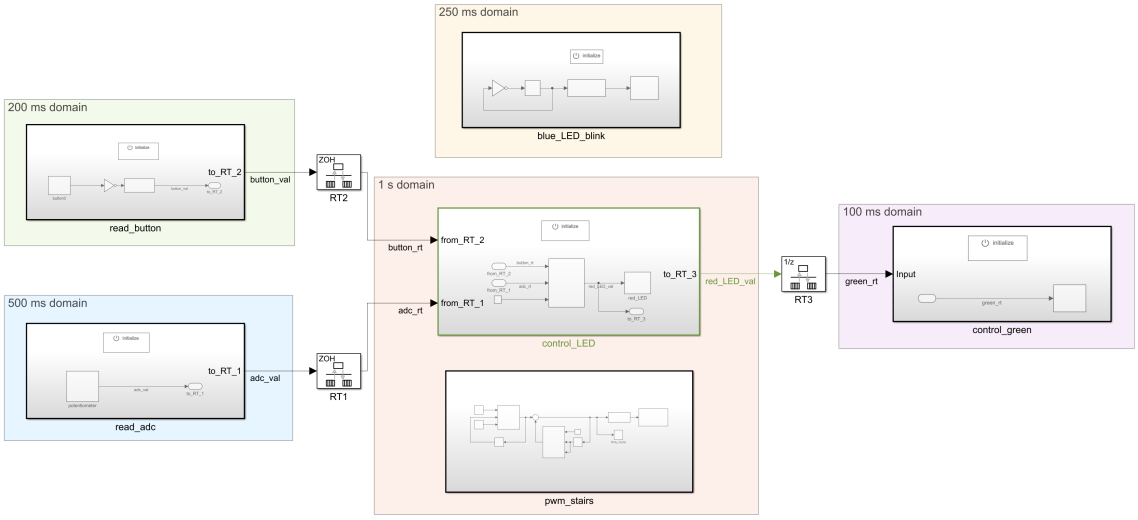


Figure 6.40: *Task-level Simulink Model.*

The variable Duty Cycle is generated by the Task *pwm_stairs*, by the following model:

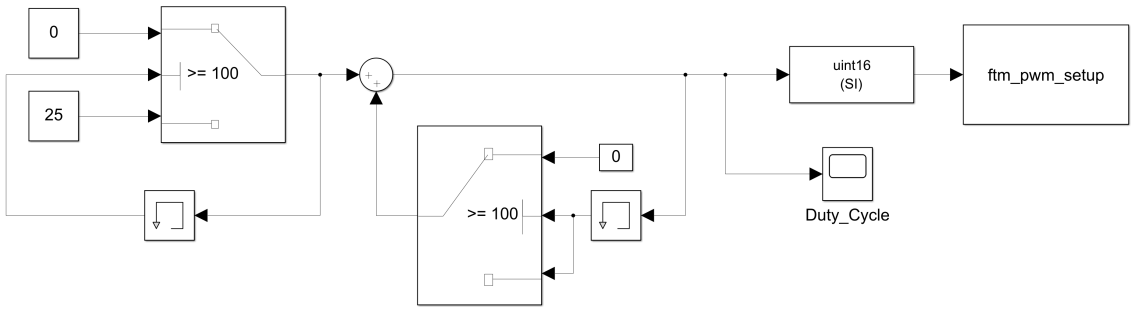


Figure 6.41: *Duty Cycle stairs generator model (pwm_stairs Task).*

In order to refer to the previous Duty Cycle value, a memory block is used. Moreover, the BSP function *ftm_pwm_setup* expects a 16-bit parameter for the Duty Cycle, so a *data conversion type* block is needed. The model was simulated, and what is obtained is the expected waveform representing the Duty Cycle changing, each step time corresponding to the Task cycle period of 1 second:

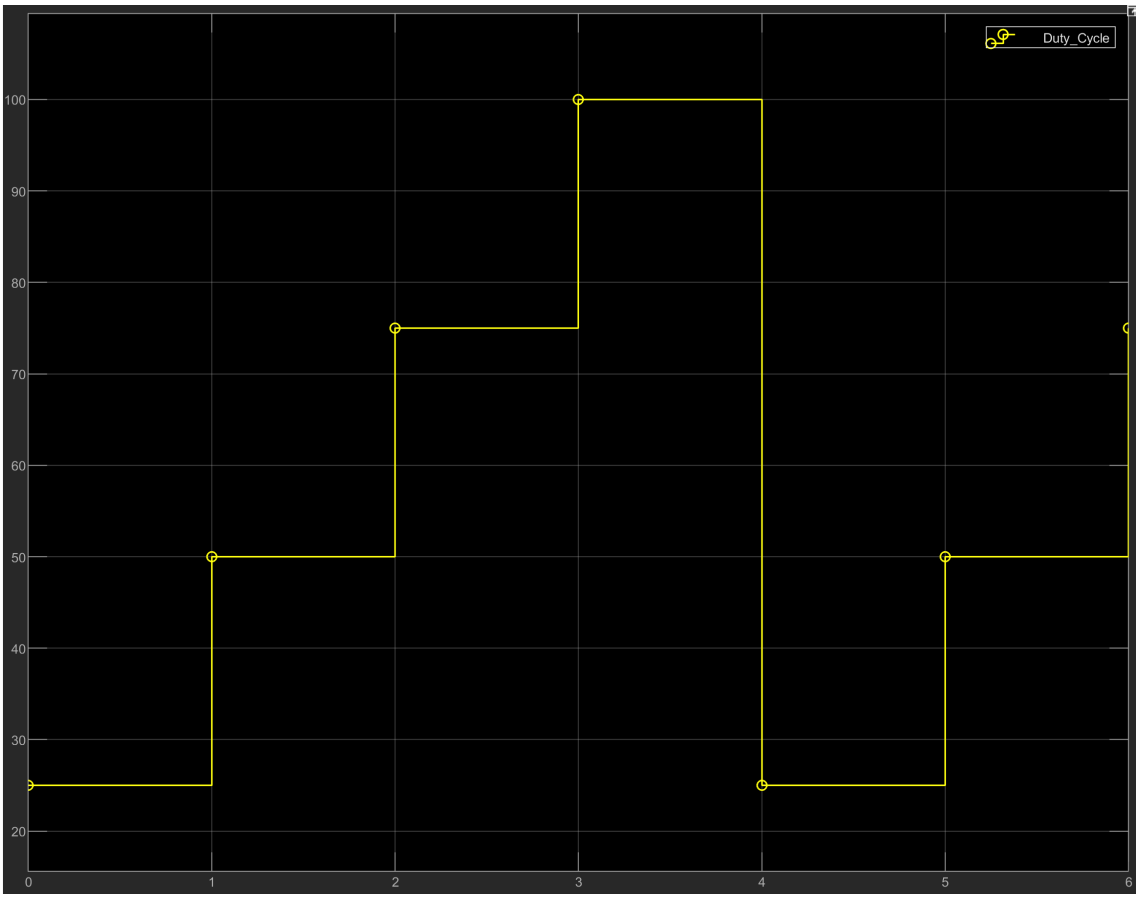


Figure 6.42: *Duty Cycle Scope.*

The obtained code is similar to the previous analyzed cases, except for the *control_LED* Task, that shares data with three different Tasks, two as inputs and one as output: the specified Task cycle periods impose priorities such that the higher priority reader task *control_green*, at the end of the chain, will transfer its priority to the writer Task *control_LED*, at the middle of the chain, in case the shared data has to be updated (each 10 executions of the reader Task), due to the **Immediate Priority Ceiling**: the *blue_LED_blinking* Task will not be temporary able to preempt it because of the priority transfer, so there will not be any Priority Inversion Problems. At the end of its Task body, the Resource is released and the priority comes back to its normal value.

```
69  TASK(control_LED)
70  {
71    EventMaskType mask;
72
73    /* Task Body */
74    while (OSEE_TRUE) {
75      WaitEvent(ScheduleEvent_control_LED);
76      GetEvent(control_LED, &mask);
77      if (mask & ScheduleEvent_control_LED) {
78        ClearEvent(ScheduleEvent_control_LED);
79
80        /* Data dependency between read_adc and control_LED; Rate Transition: RT1 */
81        WaitSem(&read_adc_control_LED_sem);
82
83        /* Data dependency between read_button and control_LED; Rate Transition: RT2 */
84        WaitSem(&read_button_control_LED_sem);
85
86        /* Rise the priority to the maximum priority between tasks sharing this resource */
87        GetResource(control_LED_control_green_res);
88
89        /* call MBSD auto-generated code step function */
90        fully_generic_control_LED();
91
92        /* Increment execution Counter */
93        control_LED_count++;
94
95        /* Critical section: Get the temporary priority, update the variable */
96        fully_generic_B.green_rt = fully_generic_B.red_LED_val;
97
98        /* Independently, Post on the semaphore to the faster dependent Task */
99        PostSem(&control_LED_control_green_sem);
100
101       /* Release the resource, so come back to the initial priority */
102       ReleaseResource(control_LED_control_green_res);
103     }
104   }
105 }
```

Figure 6.43: *control_LED Task body.*

Chapter 7

Conclusions

The presented Model-Based methodology allows a faster development of the whole software stack: starting from the Simulink model, the Task application layer is generated; the Operating System is generated too, containing the necessary resources for the upper layer, because of the generation of the Oil configuration file dependent on the Model itself.

Thanks to the OSEK OS code generation and to Simulink Embedded coder, the process results faster and easier for the developer, taking into account that it only has to create a Simulink Library starting from the Board Support Package. This leads the devlopers to focus more on high quality Board Support Package development.

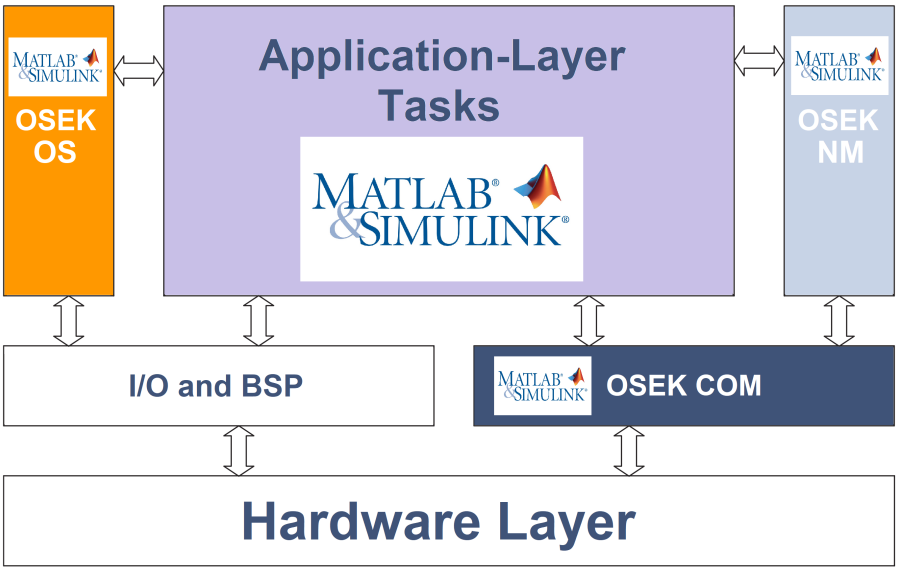


Figure 7.1: *System Code Generation Diagram.*

About integrating the generated code, due to the fact that what are generated are fully compilable syntax-error-free files, the integration of the generated code to

an existing project results quite easy.

- The OIL file, the task.c file, the model C file and all of the header files can just simply copied and pasted to the project working directory.
- After this, it is just sufficient to add inside the *hal.h* header file the definition of **MATLAB_GEN_CODE** in order to activate the MATLAB generated code interface previously introduced in part 4.1.
- At the end, a call to the model initialize function has to be added just before the *StartOS* function call into the main function, that will start the Operating System.

In order to run the generated code, what is needed is first of all to clean the Erika project and recompile the OS, due to the possible new OIL configuration file, and then rebuild the project. After this, the code is ready to run.

Possible improvements on what developed can be related to the autobuild process after code generation: as said, the Real-Time Workshop can continue to generate a device-specific executable and loading it directly to the target Hardware. Going to this direction, starting from the Task application level model, the development would be completely independent on Erika Enterprise RTOS even if all of its features would be exploited. Moreover, by loading automatically the executable on the target device, Simulink external mode can be used in order to allow the host (PC) to communicate with the model running on the device during runtime. These processes can improve the model debug features; with the actual used Eclipse environment, no Task schedulability analysis can be performed freely: this new possible features can lead the developer to become Eclipse-independent by moving the debug process into the MATLAB environment.

Bibliography

[1] Kai Qian, David Haring, and Li Cao. *Embedded Software Development with C*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[2] Wikipedia. Osek — wikipedia, l'enciclopedia libera, 2019. [Online; checked on 12th of May, 2019].

[3] Road vehicles – Open interface for embedded automotive applications – Part 3: OSEK/VDX Operating System (OS). Standard, International Organization for Standardization, Geneva, CH, March 2005.

[4] Road vehicles – Open interface for embedded automotive applications – Part 4: OSEK/VDX Communication (COM). Standard, International Organization for Standardization, Geneva, CH, March 2005.

[5] Road vehicles – Open interface for embedded automotive applications – Part 5: OSEK/VDX Network Management (NM). Standard, International Organization for Standardization, Geneva, CH, March 2006.

[6] Joseph Lemieux. *Programming in the OSEK/VDX Environment*. CMP Books, 1st edition, 2009.

[7] OSEK. *OSEK/VDX - Operating System*. OSEK.

[8] OSEK. *OSEK/VDX - System Generation. OIL: OSEK Implementation Language*. OSEK.

[9] Evidence S.r.l. *ERIKA Enterprise Manual, Real-Time made easy*. 1.4.5 edition, 2012.

[10] Inc. Frescale Semiconductor. *OpenSDA - User's Guide*. Freescale.

[11] Why Adopt Model-Based Design for Embedded Control Software Development? White paper, Mathworks, Natick, USA, 2014.

[12] The Mathworks Inc. *Writing S-functions*. The Mathworks.

[13] Evgeni Perelroyzen. *Digital integrated circuits : Design-for-Test Using Simulink and Stateflow*. CRC Press, 2007.

[14] The Mathworks Inc. *Real-Time Workshop For Use with SIMULINK*. The Mathworks.

[15] NXP Semiconductors. *S32K1xx Series Reference Manual*. NXP Semiconductors.