POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Model Based Design of Automotive Embedded System

Supervisor
prof. Massimo Violante

Candidate
Pietro Scandale

Internship Tutor
Ideas & Motion s.r.l.
ing. Marco Novaro

October 2019

Summary

In the last years the number of Embedded Systems used in the Automotive Sector is increased drastically. Even if all car producers have worked on improvements in the area of mechanics, the main differentiation factor between brands is the electronics area. In fact, today's trend is to replace the traditional mechanical systems with modern embedded systems that allow to develop more advanced control strategies, providing added values for the customer and making vehicles smarter.

This leads software development to face challenges like shortened development times, high safety requirements and especially the growing complexity of the code because of the increasing number of functionalities. To master these challenges car producers and suppliers conduct a paradigm change in the software development from hand-coded to model-based development.

A model-based development process is specifically attractive in embedded domains like Automotive Software due to the fact that allows a platform-independent development reducing the reengineering process caused by fast changing hardware generation, allows to easily integrate new functions into previous versions of the software and accelerates the software development process.

One of the most used tool for Model Based Software Design is Simulink. It is a software integrated with Matlab and it is used principally for modeling and simulating of dynamic systems. By using Embedded Coder (that is an extension of Simulink and Matlab coder) it is possible to generate high quality C,C++,VHDL code preserving the same behavior as the model created in Simulink. This avoids the introduction of bugs due to human errors.

The aim of this Thesis is to introduce the reader to the Model Based Software Design focusing on the developing of Custom Simulink Library and to explain how to create a Simulink model and how to use Embedded Coder to generate C code, with the help of some examples.

The target board is the Aurix/Arduino-like board developed by Ideas & Motion S.r.l. It is equipped with an Aurix Tricore TC277 that with its

embedded safety and security features is the ideal platform for a wide range of automotive and industrial applications.

Acknowledgements

I would first like to thank my thesis advisor Massimo Violante for his support and for his valuable advices.

I would also like to thank Marco Novaro for giving me the possibility to enter in the Ideas & Motion family and Andrea Pastore for his support during the Thesis work.

Finally, I must express my very profound gratitude to my parents and to my friends for providing me unfailing support and continuous encouragement during my years of study and also during the researching and writing process of this thesis. This accomplishment would not have been possible without them. Thank you.

Contents

List of Figures

Chapter 1

Model Based Software Design

In the last 20 years the value chain in the car industry has changed drastically. Even if all car producers are still working on improvements in the area of mechanics, quality requirements and in the logistic area, the main differentiation factor between brands turned out to be the electronics area. Whereas areas such as power train and body had small product development cost increases over the years, the costs for the development of electronic systems has been tripled, [1]. In fact, today's trend is to replace the traditional mechanical systems with modern embedded systems that enables the deployment of more advanced control strategies, providing added values for the customer and making vehicles smarter.

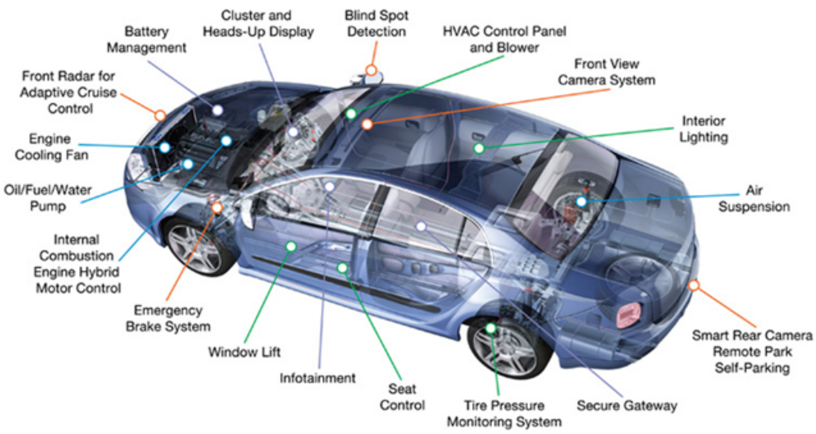


Figure 1.1. *Embedded Systems in a vehicle*

This leads software development to face challenges like shortened development times, high safety requirements and especially the growing complexity of the code because of the increasing number of functionalities. To master these challenges car producers and suppliers conduct a paradigm change in the software development from hand-coded to model-based development.

A model-based development process is specifically attractive in embedded domains like Automotive Software due to the fact that the development in these domains is driven by two strong forces: on the one side the evolutionary development of automotive systems, dealing with the iterated integration of new functions into a substantial amount of existing functionality from previous system versions; and on the other side platform-independent development, substantially reducing the amount of reengineering/ maintenance caused by fast changing hardware generations, [1]. As a result, a model-based approach is pursued to enable a shift of focus of the development process on the early phases, supporting a function based rather than a code-based engineering of automotive systems.

1.1 What is Model Based Software Design

Model-Based Design provides a mathematical and visual approach to develop complex control and signal processing systems. It centers on the use of system models throughout the development process for design, analysis, simulation, automatic code generation and verification, [2]. So with Model-Based Design the design phase is moved from the lab and field to the desktop.

Model Based Software Design (MBSD) is a software development process that aims to tackle increasing software development complexity by using abstraction and automation. Abstraction is achieved by employing suitable models of a software system while automation systematically transforms these models into executable source code, [3]. Engineers create a model to specify the behavior of an embedded system; the model, which consists of block diagrams, textual programs, and other graphical elements, is an executable specification that lets engineers run simulations to test ideas and verify designs throughout the development process, [2]. The benefits are the following:

- Improvement of the product quality: test activities during the design and development phase improve the quality of the product;
- Development of functions with high complexity: classical software development is difficult to use to design functions with high complexity.

Model-based development helps to develop high complex functions with viewer iterations and consequently less development effort;

- Better communication: the models provide great support in the communication with other colleagues because of the graphical design. It's also possible to involve people that are not familiar with software development thanks to the use of models. This helps to include extra know-how in the software development;
- Rapid Control Prototyping decreases development time by allowing corrections to be made early in the product process. So mistakes can be corrected and changes can be made while they are still inexpensive;
- Software bugs reduction: code can be automatically generated for embedded deployment, saving time and avoiding the introduction of manual error in the code.

1.2 Model Based Design tool: Simulink

One of the most used tool for MBSD is Simulink. It is a software for modeling, simulation and analysis of dynamic systems, developed by MathWorks company. It is integrated with MATLAB.

Instead of writing manually thousands of lines of code, Embedded Coder gives the possibility to automatically generate high quality C, C++,VHDL code, which has the same behavior as the model created in Simulink. It extends MATLAB Coder™ and Simulink Coder™ with advanced optimizations for precise control of the generated functions, files, and data.

1.2.1 S-function

S-functions (system-functions) are very useful in order to extend the capabilities of the Simulink environment. An S-function is a computer language description of a Simulink block written in MATLAB, C, C++, or Fortran, [4]. C, C++, and Fortran S-functions are compiled as MEX files using the mex utility. S-functions define how a Simulink block works and for this reason they are principally used to create custom Simulink blocks that can be used many times in a model. S-Function Block Parameters window allows to specify values to pass to the corresponding S-function; so it is necessary to read the S-function's documentation to understand which paramaters the block requires.

The S-Function Builder generates the following source files in the current folder:

- *sfun.c*, contains the C source code representation of the standard portions of the generated S-function, [4]. "sfun" is the name of the S-function specified in the S-function name field;

- *sfun.tlc*, permits the generated S-function to run in Simulink Rapid Accelerator mode and allows for inlining the S-function during code generation, [4].

Matlab Legacy Code Tool provides to transforms existing functions into C MEX S-functions that can be included in Simulink models. When Embedded Coder is invoked for code generation, an appropriate call to the created function is inserted into the generated code.

1.2.2 Code Generation process

When generating code from the Simulink model, Real-Time Workshop (or Simulink coder) is invoked: its task is the generation of the *model.rtw* file. This file contains informations about the model that is then used to generate code: is a database whose content provide a description of the individual blocks within the Simulink model, [5]. The code is generated through calls to a utility called Target Language Compiler. It works like a text processor: after reading the model.rtw file, it generates code into the desired language (*e.g.* C) based on target files (.tlc), which specify particular code for each block, and model-wide files, which specify the overall code style.

After the creation of the model, it is possible to call the Code Generation by simply clicking the Build Model button in the Simulink Model window; this action automatically calls Real-Time Workshop and TLC. Figure 1.2 shows how the Target Language Compiler works with its target files and Real-Time Workshop output in order to generate code.

All the generated files are placed in the build directory and include:

- The body for the generated C source code (*model.c*) that contains three main functions: *model_step*, *model_initialize*, *model_terminate*;

- The header file *model.h* that declares model data structures and a public interface to the model entry points and data structures, [6]. It is included by subsystem .c files in the model. The generated code can be included in another existing file by simply include *model.h*;

- The header file *model_types.h* that provides forward declarations for the real-time model data structure and the parameters data structure. These may be needed by function declarations of reusable functions, [6] . *model_types.h* is included by all the generated header files in the model;
- The header file *model_private.h* that defines parameters and private data structures of the generated code.

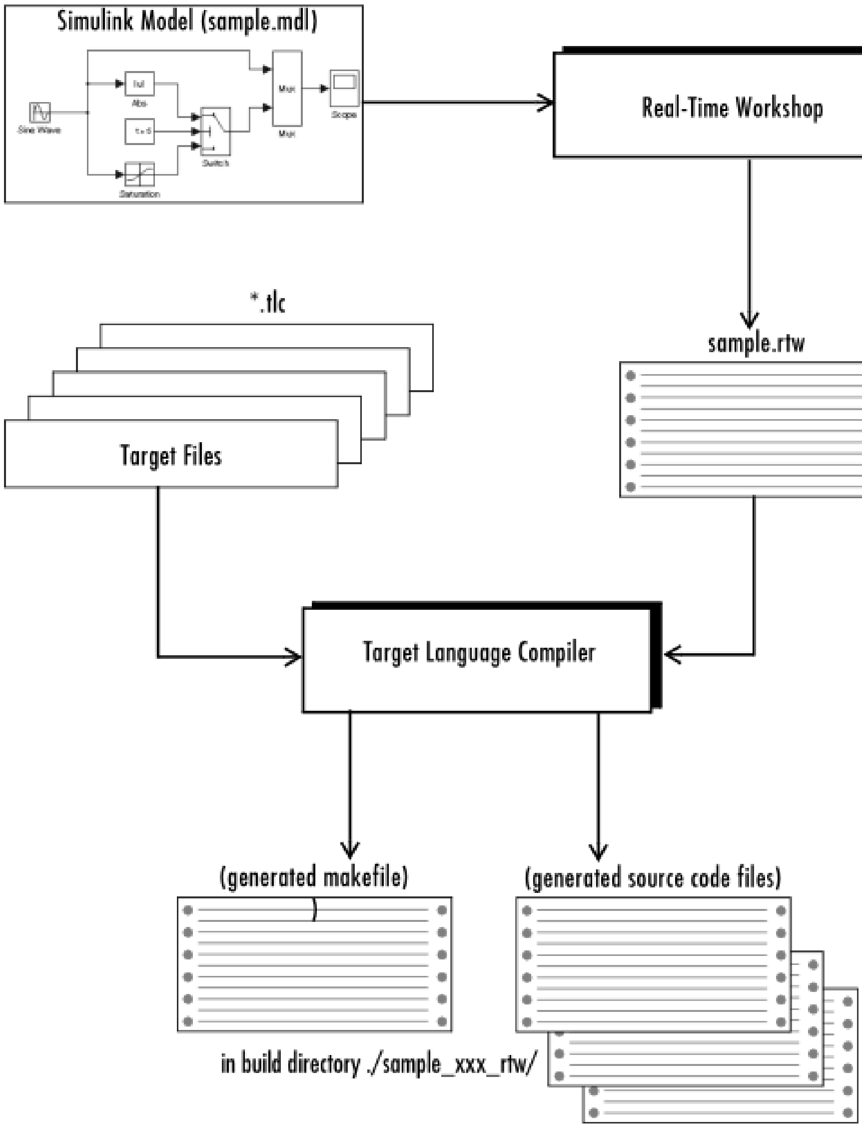


Figure 1.2. Target Language Compiler Process, [5]

Chapter 2

Aurix/Arduino-like

The Aurix/Arduino-like board (figure 2.1) was designed and developed by Ideas & Motion S.r.l. principally for the HYPER_SDF project.

Among the different applications currently under development in the automotive, the "automated vehicle" is the one which has constantly gained importance. In particular the Advanced Driver Assistance System (ADAS) is projected to be the most relevant growing market segment over the next years. The ultimate goal of the automated vehicle is to drastically improve the road safety through a precise real-time description of the scenario surrounding the vehicle.

Sensor data fusion between front and rear smart sensors is key for the development and implementation of complex algorithms supporting the autonomous driving. The HYPER_SDF project introduces an open powerful automotive development platform based on the proper combination of two diverse high-performance multi-core processors providing outstanding processing capabilities while featuring a state-of-the-art safety architecture. It was decided to use two processor beacause no processors with high performance that ensure ASIL D were avaliable. It was decided to use the Aurix Tricore because it guarantees high safety requirements and i.MX8 QM evaluation board for the high performance. The latter is used to execute the operations which require considerable processing power (*i.e.* sensor fusion etc.) while the first one has to monitor and validate all the fusion processor validation, using them to drive the vehicle.

The design of Aurix/Arduino-like was also driven by the aim to develop an easily usable board (*e.g.* to be used in the University) and eventually expandable: for this reason it was decided to follow the Arduino model, beacuse in this way all the existing modules are compatible and interfaceable with the board.

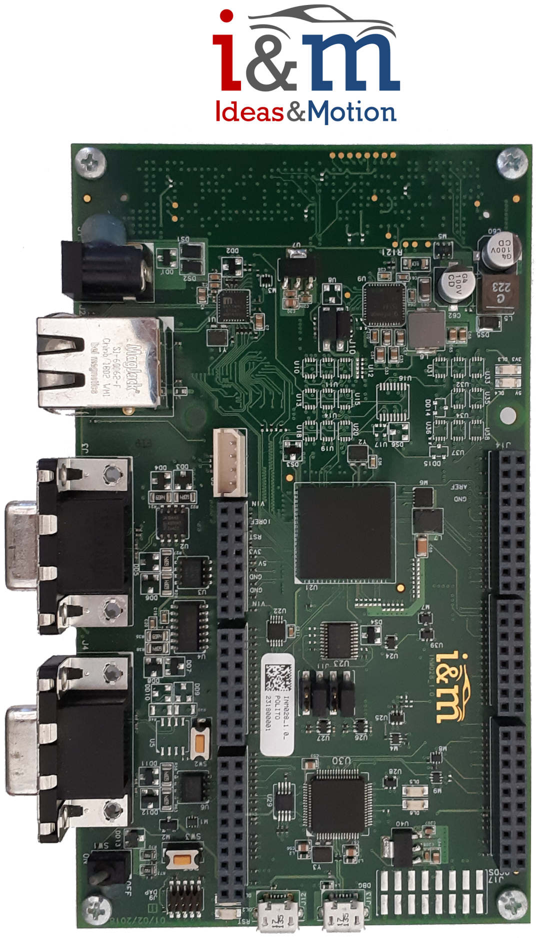


Figure 2.1. *Aurix/Arduino-like board and Ideas & Motion S.r.l logo*

2.1 Aurix™ Infineon TC277

AURIX™ is Infineon's brand new family of microcontrollers. It's based on an innovative multicore architecture and has been designed to meet the highest safety standards, while simultaneously increasing performance significantly. It is equipped with a triple TriCore with 200 MHz, 4MB of Flash memory and a Powerful Generic Timer Module (GTM). The TC27xT series aim for a reduced complexity, best-in-class power consumption and significant cost savings.

2.2 Peripherals

2.2.1 ADC

The TC277 provides a series of analog input channels connected to a cluster of Analog/Digital Converters using the Successive Approximation Register (SAR) principle to convert analog input values (voltages) to discrete digital values. The TC277 is based on individual SAR converters with dedicated Sample&Hold units, [8]. Each converter of the ADC cluster can operate independent of the others, controlled by a dedicated set of registers and triggered by a dedicated group request source. The results of each channel can be stored in a dedicated channel-specific result register or in a group-specific result register. A background request source can access all analog input channels that are not assigned to any group request source. These conversions are executed with low priority. The background request source can, therefore, be regarded as an additional background converter.

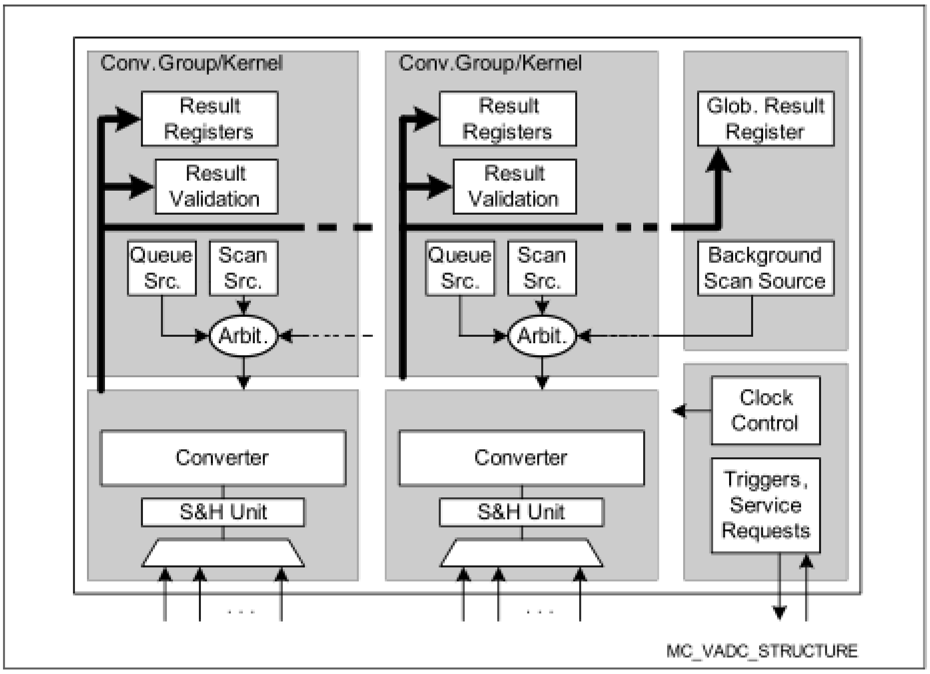


Figure 2.2. *ADC structure overview*, [8]

2.2.2 CCU6

The CCU6 is a high-resolution 16-bit capture and compare unit with application specific modes, mainly for AC drive control. It is made up of a Timer T12 Block with three capture/compare channels and a Timer T13 Block with one compare channel. The T12 channels can independently generate PWM signals or accept capture triggers, or they can jointly generate control signal patterns to drive AC-motors or inverters, [8]. The timer T12 block is the main unit to generate the 3-phase PWM signals. A 16-bit counter is connected to 3 channel registers via comparators, that generate a signal when the counter contents match one of the channel register contents. The T12 block also offers options for individual compare and capture functions, as well as dead-time control and hysteresis-like compare mode.

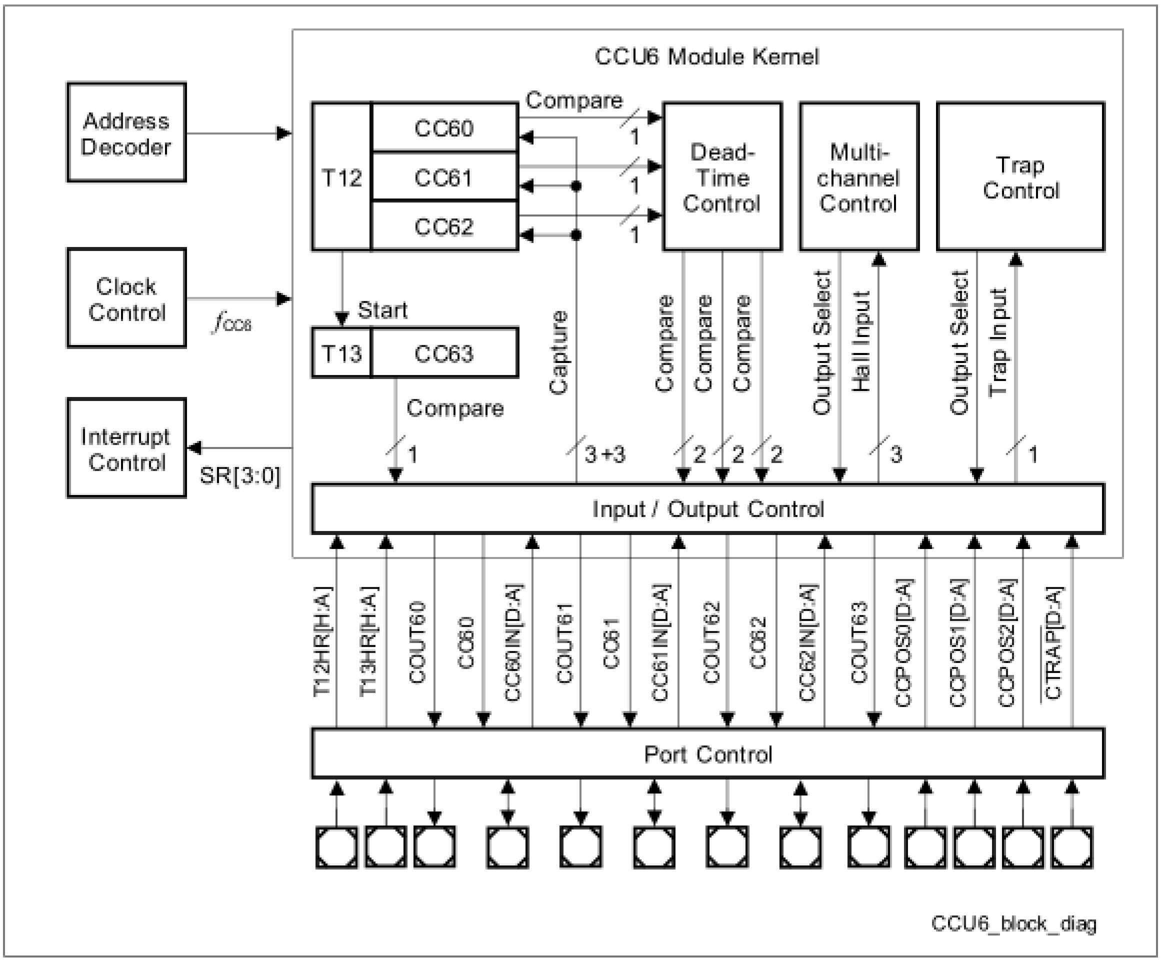


Figure 2.3. *CCU6 block diagram*, [8]

Timer T12 has been configured in order to receive its input clock (f T12) from the module clock f CC6 (100 MHz) via a programmable prescaler and an optional 1/256 divider. The bit fields T12CLK and T12PRE are used to control these options. T12 can count up or down, depending on the selected operation mode. CDIR is a direction flag that indicates the current counting direction. T12 counter register is connected to a Period Register T12PR via a comparator: this register determines the maximum count value for T12. It's possible to select among two operations mode: Edge-Aligned and Center-Aligned mode according to the value of the CTM flag.

In Edge-Aligned Mode (CTM = 0), timer T12 is always counting upwards (CDIR = 0). When the value given by the period register (period-match T12_PM) is reached, the value of T12 is cleared with the next counting step (saw tooth shape).

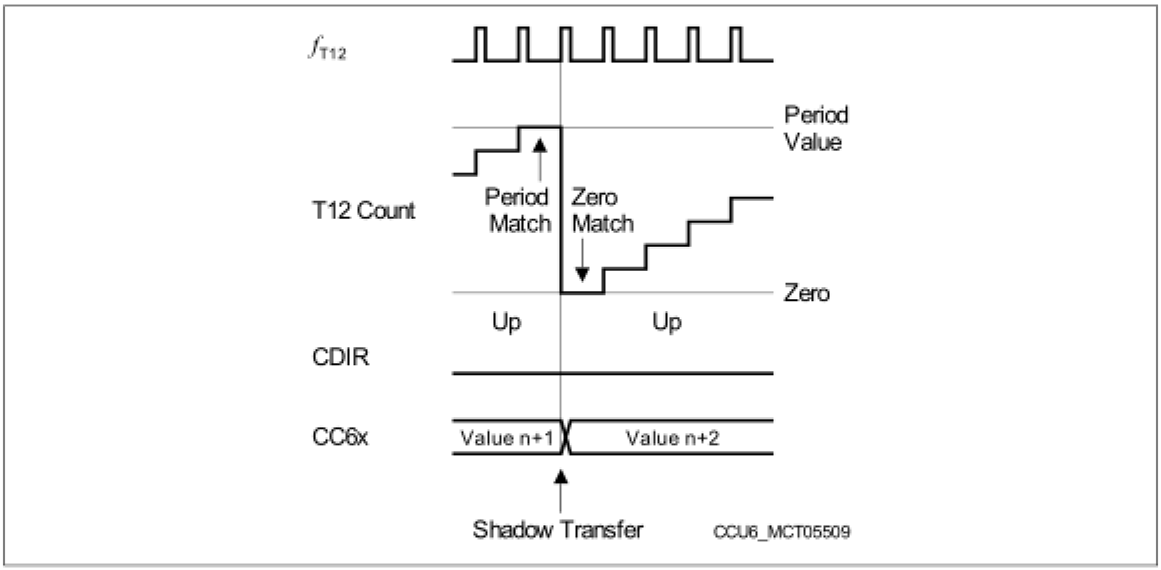


Figure 2.4. T12 Operation in Edge-Aligned Mode, [8]

In Center-Aligned Mode (CTM = 1), timer T12 is counting upwards or downwards (triangular shape). When reaching the value given by the period register (period-match T12_PM) while counting upwards (CDIR = 0), the counting direction control bit CDIR is changed to downwards (CDIR = 1) with the next counting step. When reaching the value 0001H (one-match T12_OM) while counting downwards, the counting direction control bit CDIR is changed to upwards with the next counting step (figure 2.5).

This operating mode is prefered in motor control applications because the current sampling is synchronized with the PWM period. So CCU6 has been configured to work in this way: in correspondance of the Period Match, the current sampling task is executed while in correspondance of the One Match, the computed duty cycles for the following period are updated.

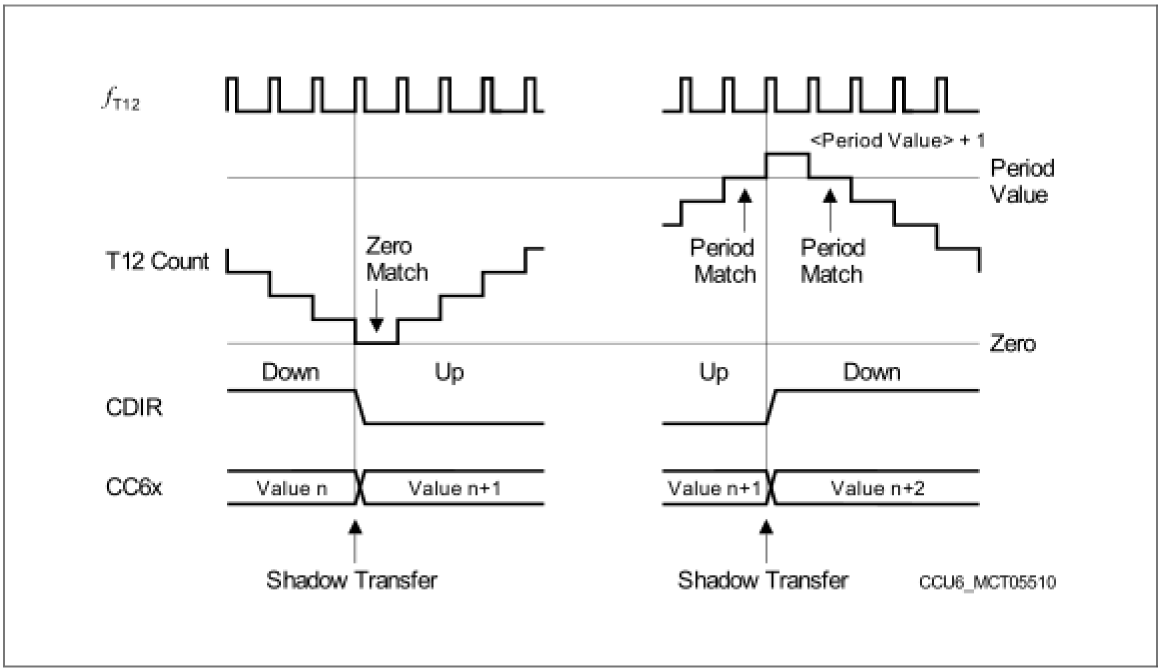


Figure 2.5. *T12 Operation in Center-Aligned Mode*, [8]

The Period Register receives a new period value from its Shadow Period Register: is controlled via the 'T12 Shadow Transfer' control signal, T12_ST. Providing a shadow register for the period value as well as for other values related to the generation of the PWM signal allows a concurrent update by software for all relevant parameters. It's possible to enable the Shadow transfer by setting the bit STE12 only in correspondence of the T12_PM or T12_OM, otherwise the upload has no effect

There are three individual capture/compare channels associated with Timer T12; they have been configured to work in Compare Mode: the three individual compare channels CC60 CC61, and CC62 can generate a three-phase PWM pattern. Each compare channel has its own equal comparator connected to the T12 counter register. A match signal is generated when the content of the counter matches the contents of the associated compare register (CC60R, CC61R, CC62R). Foreach compare register is associated a shadow register CC6xSR, that is preloaded by software and transferred into the compare register when signal T12 shadow transfer, T12_ST, is set.

The shadow registers are fundamental because they facilitate a concurrent update by software for all relevant parameters of a three-phase PWM; not only for the compare value but also for the other values related to the generation of the PWM signal facilitates.

The generation of (complementary) signals for the high-side and the low-side switches of one power inverter phase is based on the same compare channel, [8]. For example, if the high-side switch should be active while the T12 counter value is above the compare value (State Bit = 1), then the low-side switch should be active while the counter value is below the compare value (State Bit = 0). In most cases, the switching behavior of the connected power switches is not symmetrical due to switch-on and switch-off times. A problem arises if the time for switch-on is smaller than the time for switch-off of the power device: a short-circuit can occur in the inverter bridge leg, which damage the complete system. It's possible to solve this problem by HW, by using the programmable Dead-Time Generation Block of the CCU6 unit: it inserts a programmable time that delays the passive to active edge of the switching signals.

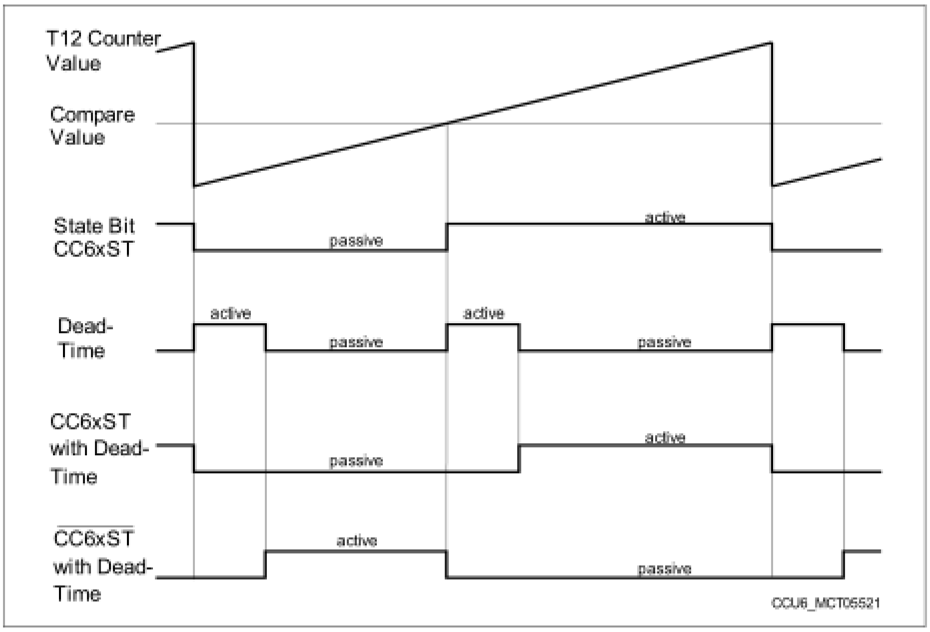


Figure 2.6. *Dead-Tme Generation Waveforms*, [8]

2.2.3 CAN

Controller Area Network, better known as CAN-bus, it is a robust vehicle bus standard designed to allow microcontrollers and devices to communicate each other withouth a host computer. It is a message-based protocol and has been designed to work without problems even in environments that are strongly disturbed by the presence of electromagnetic waves. Although initially it was applied only in the automotive sector, it is currently used in many embedded industrial applications, where it is required a high level of noise immunity.

A CAN bus consists of two or more nodes. The bus logic corresponds to a "wired-AND" mechanism. Recessive bits (equivalent to the logic 1 level) are overwritten by dominant bits (logic 0 level). As long as no bus node is sending a dominant bit, the bus is in the recessive state. In this state, a dominant bit from any bus node generates a dominant bus state. The maximum CAN bus speed is, by definition, 1 Mbit/s. This speed limits the CAN bus to a length of up to 40 m. For bus lengths longer than 40 m, the bus speed must be reduced, [8].

The binary data of a CAN frame is coded in NRZ code (Non-Return-to-Zero). To ensure re-synchronization of all bus nodes, bit stuffing is used. This means that during the transmission of a message, a maximum of five consecutive bits can have the same polarity. Whenever five consecutive bits of the same polarity have been transmitted, the transmitter will insert one additional bit (stuff bit) of the opposite polarity into the bit stream before transmitting further bits. The receiver also checks the number of bits with the same polarity and removes the stuff bits from the bit stream (= destuffing). In CAN FD format frames, the CAN bit stuffing method is changed for the CRC Sequence. The stuff bits will be inserted at fixed positions

In the CAN protocol, address information is defined in the identifier field of a message. The identifier indicates the contents of the message and its priority. The lower the binary value of the identifier, the higher is the priority of the message. For bus arbitration, CSMA/CD with NDA (Carrier Sense Multiple Access/Collision Detection with Non-Destructive Arbitration) is used.

Standard message identifier has a length of 11 bits. CAN specification 2.0B extends the message identifier lengths to 29 bits, i.e. the extended identifier. Four different data frame formats are supported which differ in the length of the Arbitration Field and Control Field:

- Classical CAN Base format: 11-bit long identifier, constant bit rate
- Classical CAN Extended format: 29-bit long identifier, constant bit rate
- CAN FD Base format: 11-bit long identifier, dual bit rate
- CAN FD Extended format: 29-bit long identifier, dual bit rate

In addition for Classical CAN remote frames exist, for 11-bit and 29bit identifiers.

There are three types of CAN frames:

- Data Frames
- Remote Frames
- Error Frames

A Data Frame contains a Data Field of 0 to 8 bytes in length. A Remote Frame contains no Data Field and is typically generated as a request for data (e.g. from a sensor). Data and Remote Frames can use an 11-bit “Standard” identifier or a 29-bit “Extended” identifier. An Error Frame can be generated by any node that detects a CAN bus error.

2.2.4 GPIO Ports

The TC27x has digital General Purpose Input/Output (GPIO) port lines which are connected to the on-chip peripheral units, [8]. Each port line has a number of control and data bits, enabling very flexible usage of the line. Each port pin can be configured for input or output operation. In input mode, the output driver is switched off (high-impedance). The actual voltage level present at the port pin is translated into a logical 0 or 1 via a Schmitt-Trigger device and can be read via the read-only register Pn_IN. Input signals are connected directly to the various inputs of the peripheral units (AltDataIn).

The level of the pin can be read by software via Pn_IN or a peripheral can use the pin level as an input. In output mode, the output driver is activated and drives the value supplied through the multiplexer to the port pin. Switching between input and output mode is accomplished through the Pn_IOCR register, which enables or disables the output driver. If a peripheral unit uses a GPIO port line as a bi-directional I/O line, register Pn_IOCR has to be written for input or output selection. The Pn_IOCR register further controls the driver type of the output driver, and determines whether an internal weak pull-up, pull- down, or without input pull device is alternatively connected to the pin when used as an input. This offers additional advantages in an application.

The output multiplexer in front of the output driver selects the signal source for the GPIO line when used as output. If the pin is used as general-purpose output, the multiplexer is switched by software (Pn_IOCR register) to the Output Data Register Pn_OUT. Software can set or clear the bit in Pn_OUT through separate Pn_OMSR or Pn_OMCR registers. The set or clear operations for the bits in Pn_OUT can also be done for up to four bits per register in Pn_OMSRx and Pn_OMCRx (x=0,4,8,12). Alternatively, the set, clear or toggle function can be achieved through Pn_OMR, where adjacent pins within the same port can be set, cleared or toggled within one write operation. The manipulation of the control bits in these registers can directly influence the state of the port pin. If the on-chip peripheral units use the pin for output signals, the alternate output lines ALT1 to ALT7 can be switched via the multiplexer to the output driver. The data written into the output register Pn_OUT by software can be used as input data to an on-chip peripheral.

When selected as general-purpose output line, the logic state of each port pin can be changed individually by programming the pin-related bits in the Output Modification Set Register Pn_OMSR, Output Modification Set Register x Pn_OMSRx (x=0,4,8,12), Output Modification Clear Register Pn_OMCR, Output Modification Clear Register x Pn_OMCRx (x=0,4,8,12) or Output Modification Register, OMR. The bits in Pn_OMSR/Pn_OMSRx and Pn_OMCR/Pn_OMCRx make it possible to set and clear the bits in the Pn_OUT register. While the bits in Pn_OMR allows the bits in Pn_OUT to be set, cleared, toggled or remain unchanged.

2.3 Real Time Operating System

In Real-Time System the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced.

It can be decomposed into a set of subsystems, *i.e.* the controlled object and the real-time computer system. A real-time computer system must react to stimuli from the controlled object within time intervals dictated by its environment. The instant at which a result is produced is called a deadline. If the result has utility even after the deadline has passed, the deadline is classified as soft; if missing its deadline makes the result useless, but missing does not cause serious damage, the deadline is classified as firm while if a catastrophe could happen if a deadline is missed, the deadline is hard, [9]. Commands and Control systems, Air traffic control systems are examples for hard real-time systems. On-line transaction systems, airline reservation systems are soft real-time systems.

An operating system is a system software that manages the hardware and software resources of the machine, providing basic services to the application software.

Most operating systems allow multiple programs to execute at the same time. This is called multi-tasking. In reality there is no parallel management of processes but they are executed in sequence: the times are so short that the user seems that the programs go simultaneously. A part of the operating system called scheduler is responsible for deciding which program to run and when, and provides the illusion of simultaneous execution by rapidly switching between each program.

The type of an operating system is defined by how the scheduler decides which program to run when. The scheduler in a Real Time Operating System (RTOS) is designed to provide a predictable (normally described as deterministic) execution pattern. This is particularly of interest to embedded systems as embedded systems often have real time requirements. A real time requirements is one that specifies that the embedded system must respond to a certain event within a strictly defined time. A guarantee to meet real time requirements can only be made if the behaviour of the operating system's scheduler can be predicted (and is therefore deterministic).

Traditional real time schedulers achieve determinism by allowing the user to assign a priority to each thread of execution. The scheduler then uses the priority to know which thread of execution to run next. A thread of execution is called task.

2.3.1 OSEK/VDX Standards

OSEK (in english: "Open Systems and their Interfaces for the Electronics in Motor Vehicles") 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 was designed to provide a standard software architecture for the various electronic control units (ECUs) in a car.

OSEK was founded in 1993 by a German automotive company consortium (BMW, Robert Bosch GmbH, DaimlerChrysler, Opel, Siemens, and Volkswagen Group) and the University of Karlsruhe. In 1994, the French cars manufacturers Renault and PSA Peugeot Citroën, which had a similar project called VDX (Vehicle Distributed eXecutive), joined the consortium. Therefore, the official name is OSEK/VDX.

The OSEK operating system serves as a basis for application programs which are independent of each other, and provides their environment on a processor, [11]. The OSEK operating system enables a controlled real-time execution of several processes which appear to run in parallel. The OSEK operating system provides a defined set of interfaces for the user. These interfaces are used by entities which are competing for the CPU. There are two types of entities:

- Interrupt service routines managed by the operating system
- Tasks (basic tasks and extended tasks)

The hardware resources of a control unit can be managed by operating system services. These operating system services are called by a unique interface, either by the application program or internally within the operating system. OSEK defines three processing levels:

- Interrupt level
- Logical level for scheduler
- Task level

Within the task level tasks are scheduled (non, full or mixed preemptive scheduling) according to their user assigned priority. The run time context is occupied at the beginning of execution time and is released again once the task is finished. The following priority rules have been established:

- Interrupts have precedence over tasks
- The interrupt processing level consists of one or more interrupt priority levels
- Interrupt service routines have a statically assigned interrupt priority level
- Assignment of interrupt service routines to interrupt priority levels is dependent on implementation and hardware architecture
- For task priorities and resource ceiling-priorities bigger numbers refer to higher priorities.
- The task's priority is statically assigned by the user

The main purpose of the OSEK operating system (OS) specification is to achieve portability between application software from different electronic control units (ECU). Because the specification ends with defining an API on C-language level together with the declaration of the relevant datatypes, applications still are not portable between OS-implementations of different vendors. A new language is so defined to achieve portability. The OSEK implementation language (OIL) specifies means to declare and define all relevant OS-objects, [12]. Currently it is intended to specify all OS-objects for an application in a centralized OIL-file. OIL-files have to be parsed to collect the specified informations and translated into C data structures and probably some code. This task will be typically handled by a system generation which will be delivered by the operating system vendor.

2.3.2 Erika Enterprise RTOS

ERIKA Enterprise is an innovative RTOS developed for microcontrollers. Its kernel is a complete OSEK/VDX environment, which can be used to implement multithreading applications. The Erika Enterprise API provides support for thread activation, mutual exclusion, alarms, events and counting semaphores. The ERIKA Enterprise kernel implements innovative algorithms such as Fixed Priority with supremacy threshold, Stack Resource Policy (SRP), and Earliest Deadline First (EDF), which can be used to program tasks with real-time requirements. 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 mul- ticore platforms while guaranteeing predictable real-time performance and retaining the programming model of conventional single processor architectures, [10]. The advanced features provided by Erika Enterprise are:

- Support for four conformance classes to match different application requirements;
- Support for preemptive and non-preemptive multitasking;
- Support for fixed priority scheduling;
- Support for stack sharing techniques, and one-shot task model to reduce the overall stack usage;
- Support for shared resources;
- Support for periodic activations using Alarms;
- Support for centralized Error Handling;
- Support for hook functions before and after each context switch;

Erika Enterprise is supported by RT-Druid, a tool suite based on Eclipse Framework for the automatic configuration and deployment of embedded applications which enables to easily exploit multi-processor architectures and achieve the desired performance without modifying the application source code.

It is an OIL language compiler, which is able to generate the ERIKA Enterprise configuration from an OIL specification. It generates all the files needed, such as the makefiles and the ERIKA Enterprise internal data structure initializations.

Chapter 3

Getting Started

The best way to load the executable and debug the Aurix/Arduino-like board is with the use of TRACE32: it allows to test embedded hardware and software by using the on-chip debug interface (the most common is JTAG). TRACE32 tools connect to this one to control the core, so the access to the data being processed by the core is guaranteed. It is possible to have start, stop, step control; to read and write memory and registers; to set breakpoints; to track values of variables and so on.

Since is not so easy to obtain this instrument (for cost reason), another solution has been founded in order to avoid to use it. This has been possible thanks to the presence of the FTDI (FT2232HL) module on the target boad. This module contains a small EEPROM configuration memory and convert the USB in JTAG + serial.

In this chapter will be explained how to set the environment before interfacing with the Aurix/Arduino-like board. In particular the first part is dedicated to the Software resources (section 3.1) with a short description about each needed software and some hints on where to download and how to install them correctly. The second part is dedicated to explain how to program the FTDI module (section 3.1.4), import the project in Eclipse (section 3.2.2), compile ERIKA RTOS (section 3.2.3), build the project (section 3.2.4) and in the end how to program and debug the board (section 3.2.5).

3.1 Software resources

The following software are needed:

- Cygwin package
- Ninja Genie
- FT_Prog
- Java jre1.8.0_171 version
- HighTec Free TriCore™ Entry Tool Chain

3.1.1 Cygwin

Description

Cygwin is a large collection of GNU and Open Source tools which provide functionality similar to a Linux distribution on Windows.

How to get it

Installing and Updating Cygwin Packages from `https://cygwin.com/install.html`

Installation hints

The cygwin installation folder must be `C:\cygwin` (or `C:\cygwin64` depending on the Operating System configuration). The following packages shall be download during the installation:

- Category: Devel
  - binutils: GNU assembler, linker and similar utilities
  - gcc-core: GNU Compiler Collection (C, OpenMP)
  - make: The GNU version of the "make" utility
- Category: Basic
  - sed: the GNU sed stream editor

The path `C:\cygwin\bin` (or `C:\cygwin64\bin`) must be added to the *Environment variables* of Windows to the *Path* (inside *System variables*) and then moved up to the second position.

3.1.2 Ninja Genie

Description

Ninja is a small build system with a focus on speed. It differs from other build systems in two major respects: it is designed to have its input files generated by a higher-level build system and to run builds as fast as possible.

How to get it

The two executables `ninja.exe` and `genie.exe` can be found in the `Getting_Started\Ninja_Genie` folder and must **NOT** be installed.

Installation hints

The `Ninja_Genie` folder must be copied in `C:` and the path `C:\Ninja_Genie` must be added to the *Environment variables* of Windows to the *Path* (inside *System variables*) and then moved up to the first position.

3.1.3 Java Runtime Environment

Description

Java Runtime Environment must be present for using RT-Druid in order to write and compile application based on ERIKA Enterprise.

How to get it

The executable `jdk-8u171-windows-x64.exe` can be found in the `Getting_Started` folder.

Installation hints

Double click on the executable and simply follow the instructions.

WARNING: Only with this version of Java everything works so it's important to install it.

3.1.4 FT_Prog

Description

FT_Prog is a free EEPROM programming utility for use with FTDI devices. It is used for modifying EEPROM contents that store the FTDI device descriptors to customize designs, [13].

How to get it

FT_Prog is available as a free download from https://www.ftdichip.com/Support/Utilities.htm#FT_PROG

Installation hints

Download and simply follow the instructions.

3.1.5 HighTec Free TriCore™ Entry Tool Chain

Description

The tool chain consists of a compiler based on the proven high performance GNU compiler for TriCore™ from HighTec and the Universal Debug Engine limited to level1 functionality. The HighTec Free TriCore™ Entry Tool Chain provides all required features to develop and test software for TriCore™ and AURIX™.

How to get it

HighTec Free TriCore™ Entry Tool Chain is available as a free download from https://free-entry-toolchain.hightec-rt.com/ and follow the instructions in order to correctly generate the License File and obtain the program.

Installation hints

Read the manual https://free-entry-toolchain.hightec-rt.com/getting_started.pdf?d=20180608 and follow the instructions of the *Installing the Free TriCore Entry Tool Chain* and *First Starting of Eclipse* chapter.

3.2 Configuration and Build process

First of all make sure that the board is connected to the PC in the right way (figure 3.1).

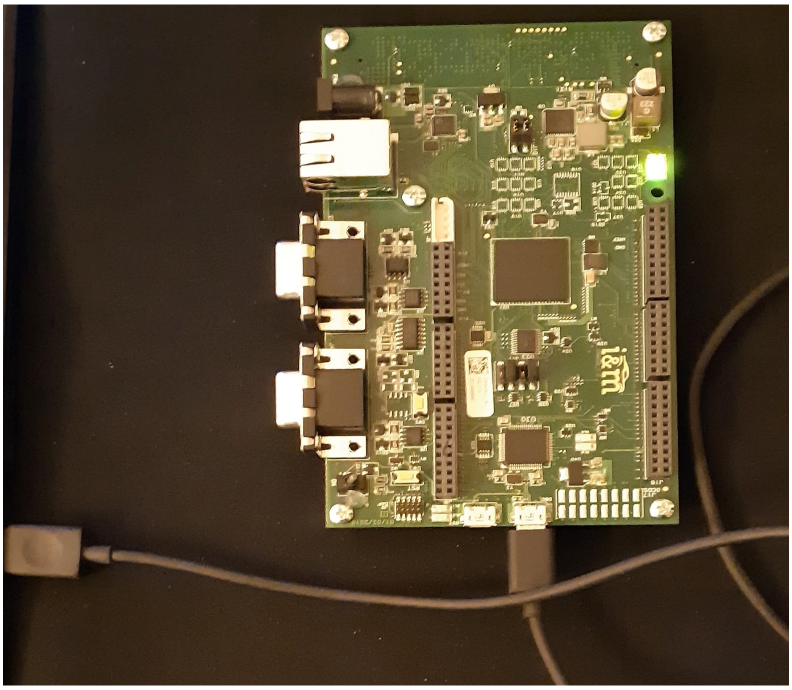


Figure 3.1. *Aurix/Arduino-like board right connection*

3.2.1 FTDI programming

Hereby is listed a step-by-step guide in order to correctly program the FTDI:

1. Launch FT_Prog.
2. Scan for Device: click on the *Scan and Parse* button on the toolbar to scan the USB bus for available connected FTDI devices (figure 3.2).

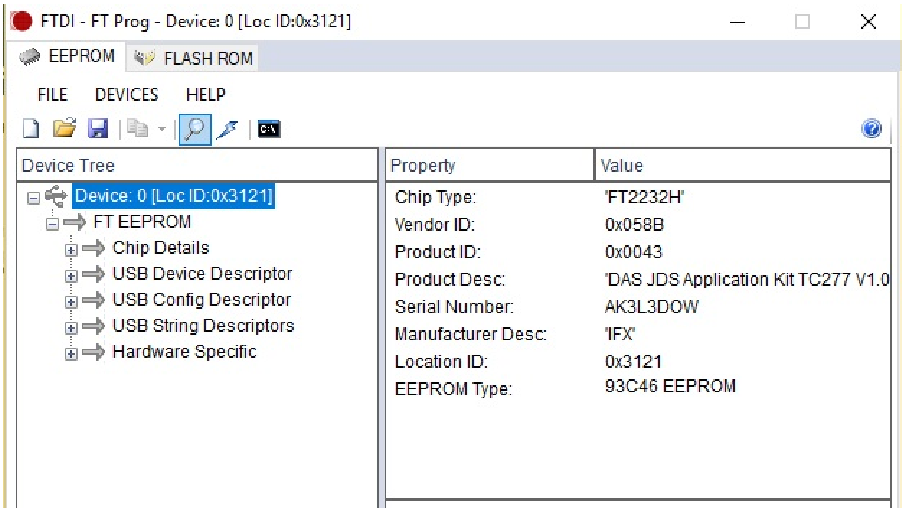


Figure 3.2. *FT_Prog: Scan and Parse*

3. Use an Existing EEPROM Template: right-click the required device within the *Device Tree*; select *Apply Template* from the menu and then select *From File* to apply these one to the target device. Find and select the *TC277_ IFX_ EVB. XML* file located in the *Getting_Started* folder.

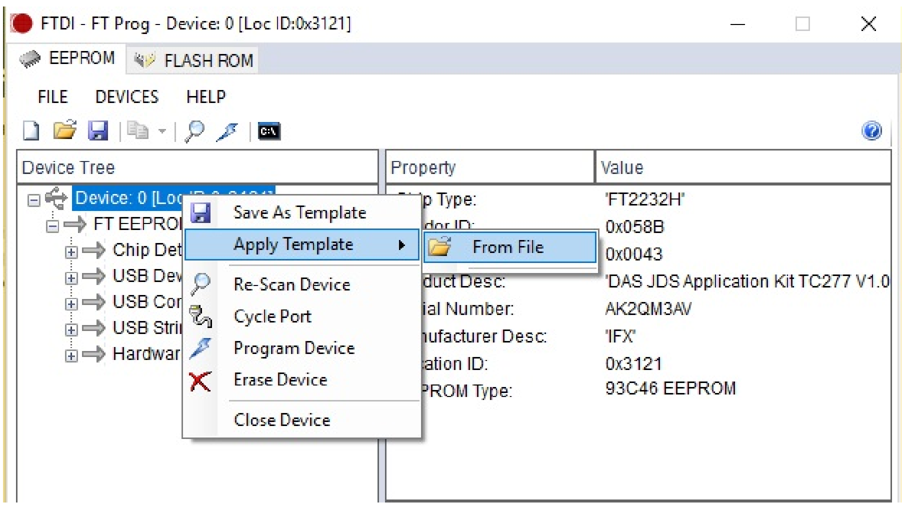


Figure 3.3. *FT_Prog: Apply Template*

4. Program Device: right-click the required device within the *Device Tree* and then select *Program Devices* (figure 3.4).

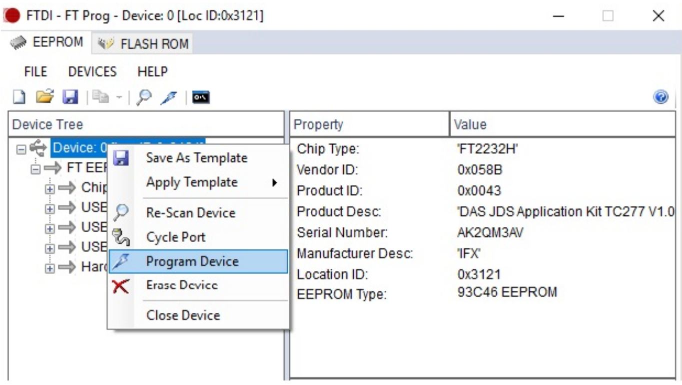


Figure 3.4. *FT_Prog: Program Device*

3.2.2 Import Existing Code

Some steps has to be followed in order to correctly import existing code in Eclipse:

1. Launch Eclipse for TriCore.
2. From *File* menu select *New* → *Project...* and then *New Project* wizard appears.
3. Select *C/C++* → *Makefile Project with Existing Code* and then click *Next*.

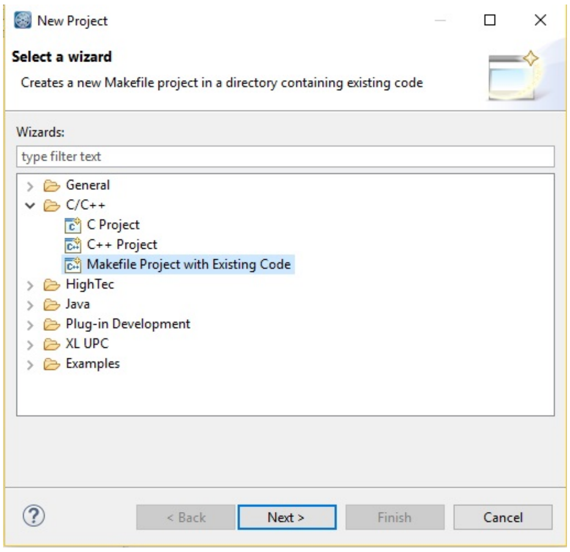


Figure 3.5. *Eclipse: Select a wizard*

4. The next wizard page allows to choose a Project Name (*e.g. test*) and the location of the existing project (*e.g.* ***C:\Users\Pietro\Desktop\Getting_Started\TricoreBswl***). Select *Cygwin GCC* in the *Toolchain for Indexer Settings*.

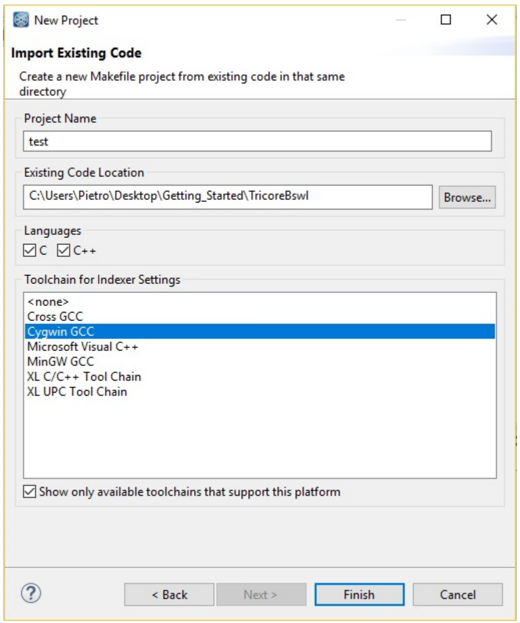


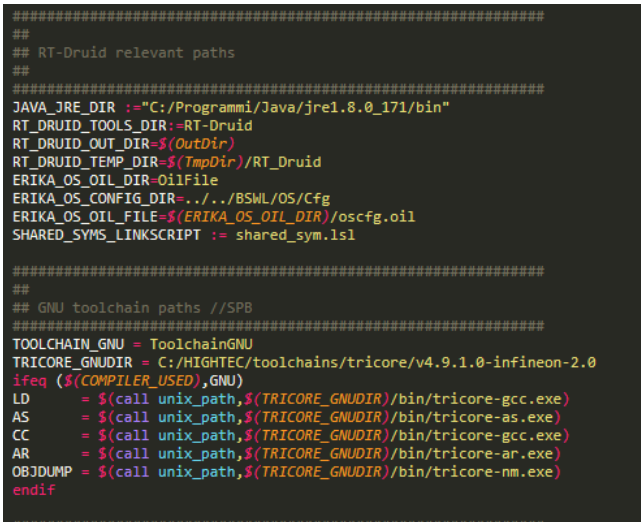
Figure 3.6. *Eclipse: Project Configuration*

5. Click *Finish* to close the wizard and to import the project.

3.2.3 Build ERIKA RTOS

`TricoreBswl` project contains *pathcfg.mk* makefile used to set all the configurations paths (figure 3.7). Only the values saved in the following two variables must be modified:

- **JAVA_JRE_DIR** contains the path of Java Runtime Environment binaries;
- **TRICORE_GNUDIR** contains the file system location of HIGHTECH TriCore compiler.



```
####################################################################
##
## RT-Druid relevant paths
##
####################################################################
JAVA_JRE_DIR :="C:/Programmi/Java/jre1.8.0_171/bin"
RT_DRUID_TOOLS_DIR:=RT-Druid
RT_DRUID_OUT_DIR=$(OutDir)
RT_DRUID_TEMP_DIR=$(TmpDir)/RT_Druid
ERIKA_OS_OIL_DIR=OilFile
ERIKA_OS_CONFIG_DIR=../../BSWL/OS/Cfg
ERIKA_OS_OIL_FILE=$(ERIKA_OS_OIL_DIR)/oscfg.oil
SHARED_SYMS_LINKSCRIPT := shared_sym.lsl

####################################################################
##
## GNU toolchain paths //SPB
####################################################################
TOOLCHAIN_GNU = ToolchainGNU
TRICORE_GNUDIR = C:/HIGHTEC/toolchains/tricore/v4.9.1.0-infineon-2.0
ifeq ($(COMPILER_USED),GNU)
LD      = $(call unix_path,$(TRICORE_GNUDIR)/bin/tricore-gcc.exe)
AS      = $(call unix_path,$(TRICORE_GNUDIR)/bin/tricore-as.exe)
CC      = $(call unix_path,$(TRICORE_GNUDIR)/bin/tricore-gcc.exe)
AR      = $(call unix_path,$(TRICORE_GNUDIR)/bin/tricore-ar.exe)
OBJDUMP = $(call unix_path,$(TRICORE_GNUDIR)/bin/tricore-nm.exe)
endif
```

Figure 3.7. *pathcfg.mk makefile*

The configuration of ERIKA Enterprise system is defined inside *oscfg.oil* file. The following command must be run on Cygwin command line each time *oscfg.oil* is modified. So first of all open Cygwin and move inside the `TricoreBswl` project where a *make.bat* file is present. Run the following commands in the same order:

1. *./make.bat oscfg* to generate ERIKA RTOS configuration file;
2. *./make.bat os* to build ERIKA RTOS.

   An help command can be also invoked to have the list of all accepted commands (*./make.bat help*).

In case of no errors, the outputs files are located in `TricoreBswl\PrjOutput\ErikaOs_out` folder.

3.2.4 Build the Project

Before building the project, the builder settings must be changed:

1. Right click on the top level directory of the project and then click on *Properties*.
2. The *Properties* dialog appears. Click on *C/C++ Build* and remove the tick from *Use default build command*. Now write on *Build command* the following instruction: `C:\cygwin\bin\mintty.exe -e C:\Users\`

`Pietro\Desktop\Getting_started\TricoreBswl\make.bat` > *file.txt*
Where the first path is the location of *mintty.exe* in your PC, *-e* in order to treat remaining arguments as the command to execute, the second path is the location of the *make.bat* file while *file.txt* contains the output message of the building process. Then click on *Apply* → *Ok*.

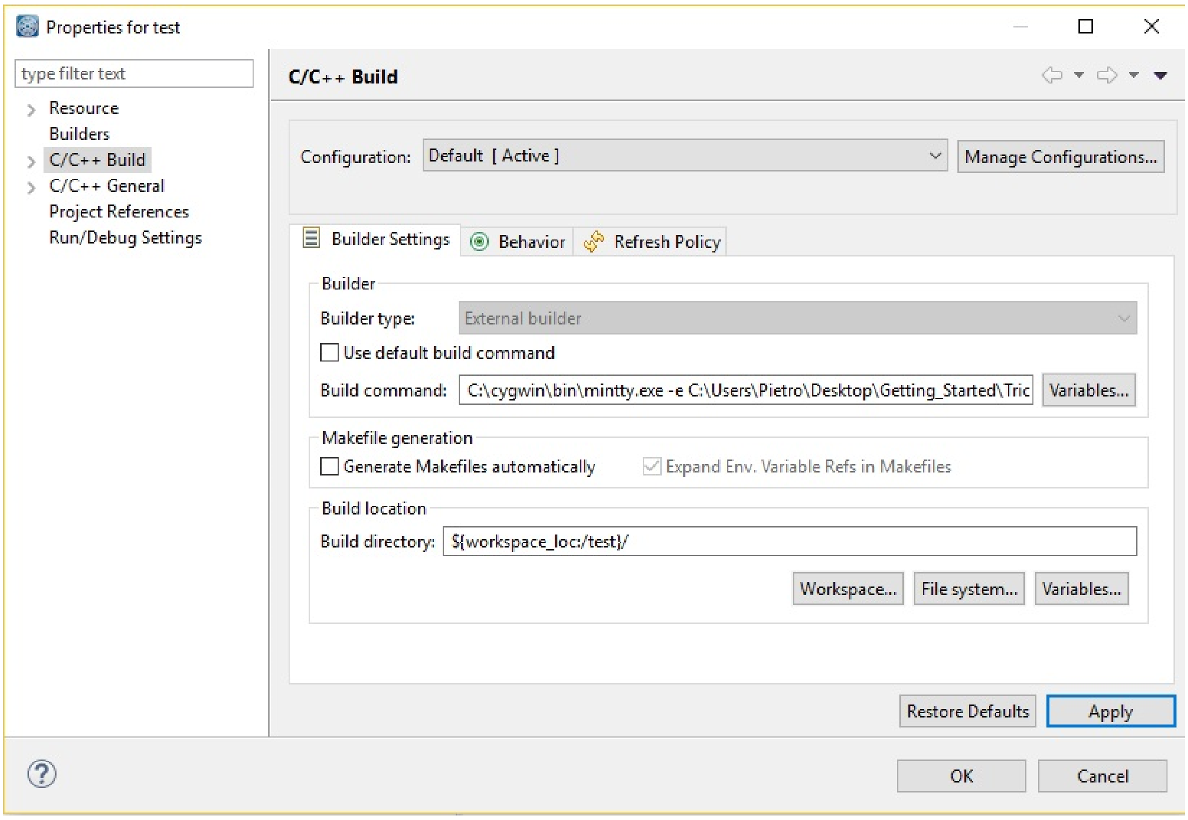


Figure 3.8. *Eclipse: Builder Configuration*

3. Right click on the top level directory of the project → *Build Project*. If all is done correctly the building starts and a message like below appears.

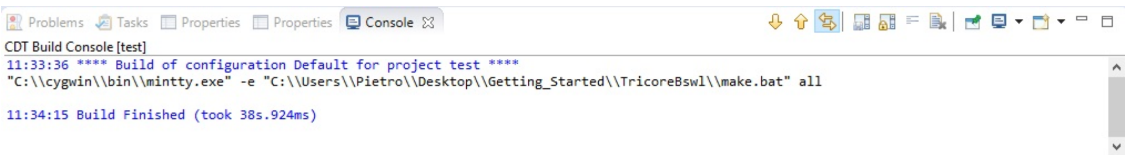


Figure 3.9. *Eclipse: Console Message*

4. Open *file.txt* and read the building report in order to understand if the compilation successful or if there are some errors or warnings.

In case of no errors, the outputs files are located in `TricoreBswl\PrjOutput\bswl_out` folder.

3.2.5 Debug

In order to start a debug session:

1. Right click on the top level directory of the project → *Debus As* → *Debug configuration...*.
2. The *Debug Configurations* dialog appears. Right click on *Universal Debug Engine as debug type* → *New* to create a new debug launch configuration for Universal Debug Engine.

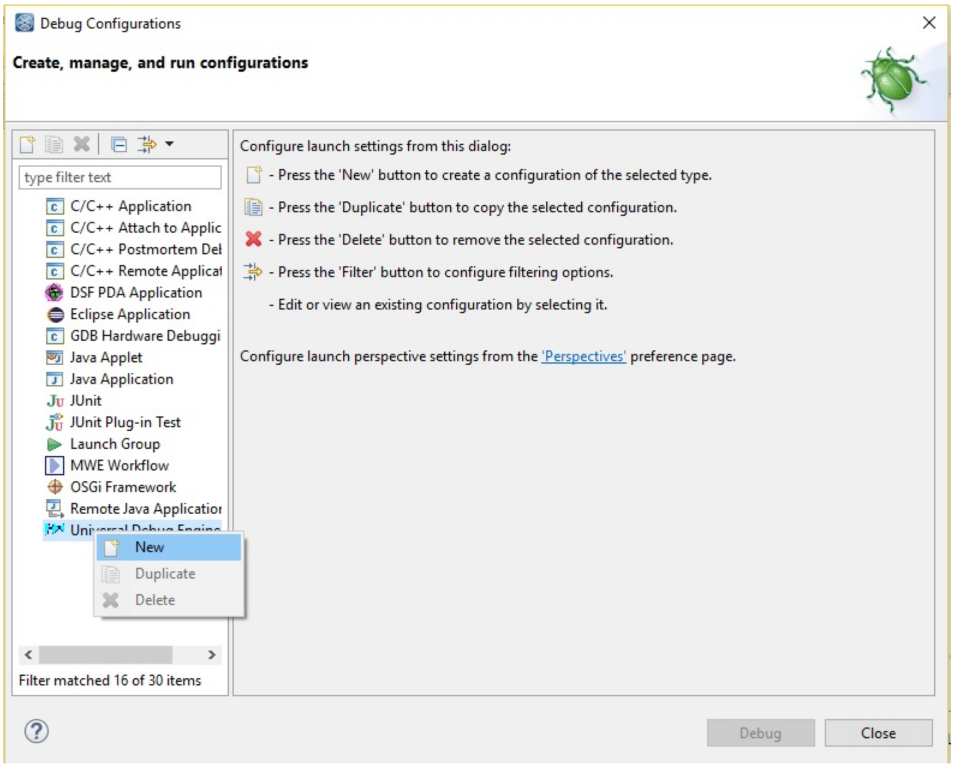


Figure 3.10. *Eclipse: Debug Configuration*

3. A new debug configuration *test Default* is created. Fill all inputs field with the appropriate values. In *C/C++ Applications* click on *Browse...*

and select the *.elf* file generated after the building (*Tricore_Bswl_cpu0.elf* is located in `Getting_Started\TricoreBswl\PrjOutput\bswl_out`). In *Project* write the name of the current project.

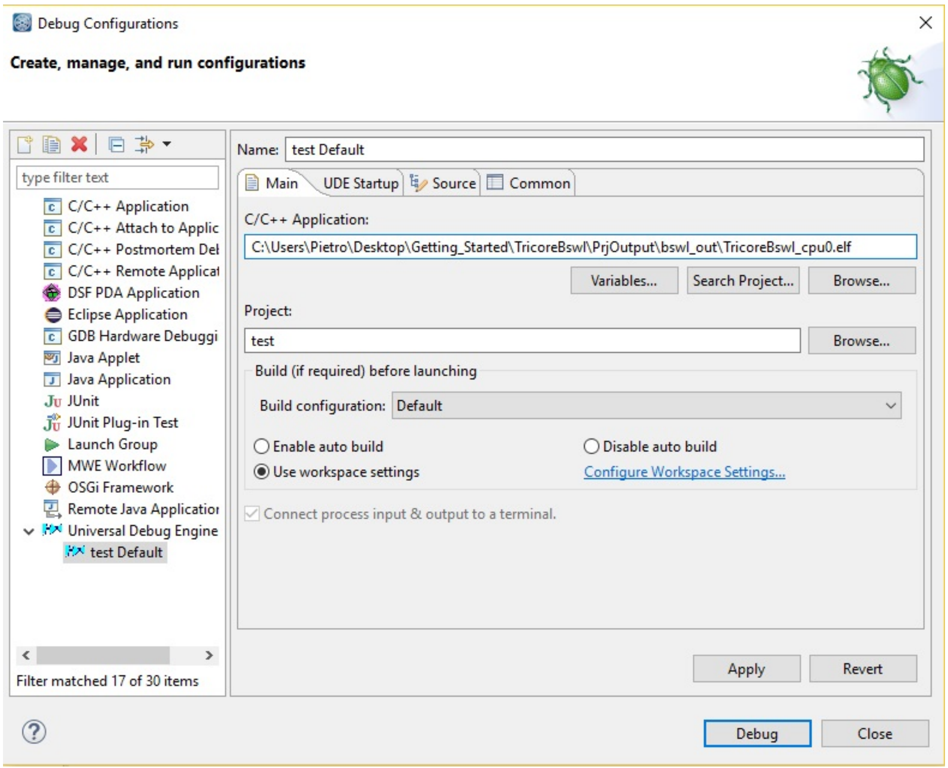


Figure 3.11. *Eclipse: Universal Debug Engine Main Congifuration*

4. Click on *Ude Startup*. The field *Select UDE Workspace File* is usually filled automatically otherwise find manually the right file in the workspace. In *Select UDE Target Configuration File* click on *Browse configuration* and select the *AppKit_TC277C_singlecore.cfg* file located in the *Getting_Started* folder.

   Push *Debug* to start UDE perspective

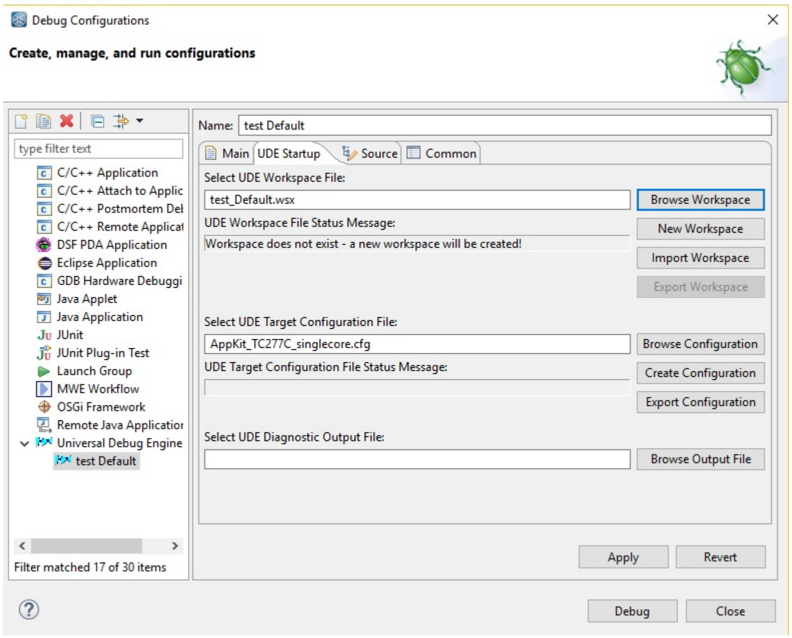


Figure 3.12. *Eclipse: Universal Debug Engine Main Congifuration*

5. The *UDE Memory Programming Tool* will appear after launching the UDE perspective.

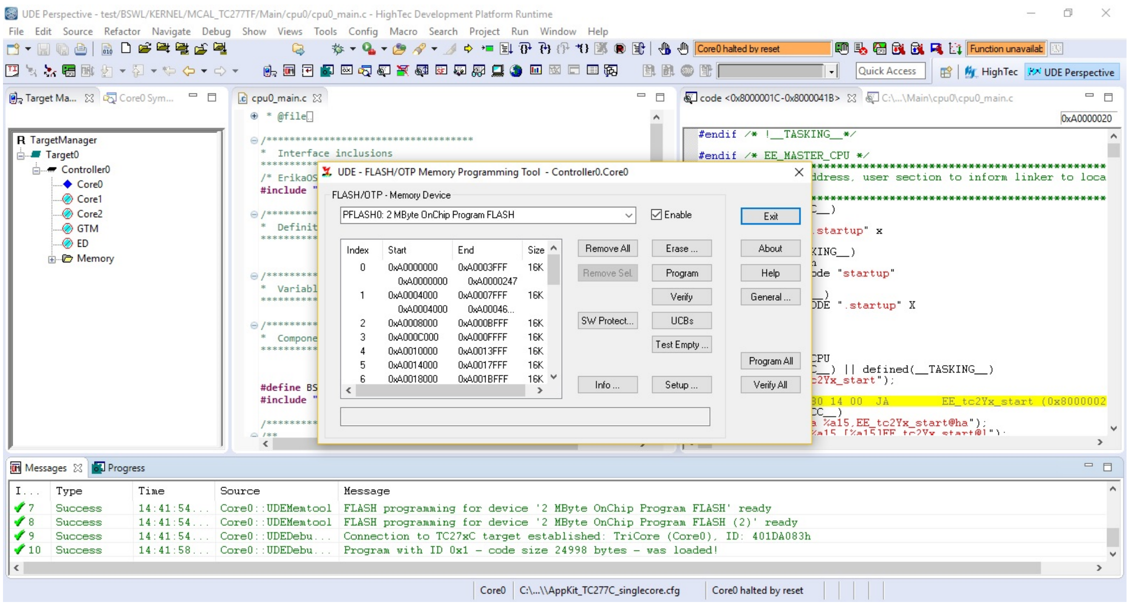


Figure 3.13. *Eclipse: Universal Debug Engine Memory Programming Tool*

6. Start flashing with the *Program* button. A progress dialog appears. After successful programming close both dialogs.

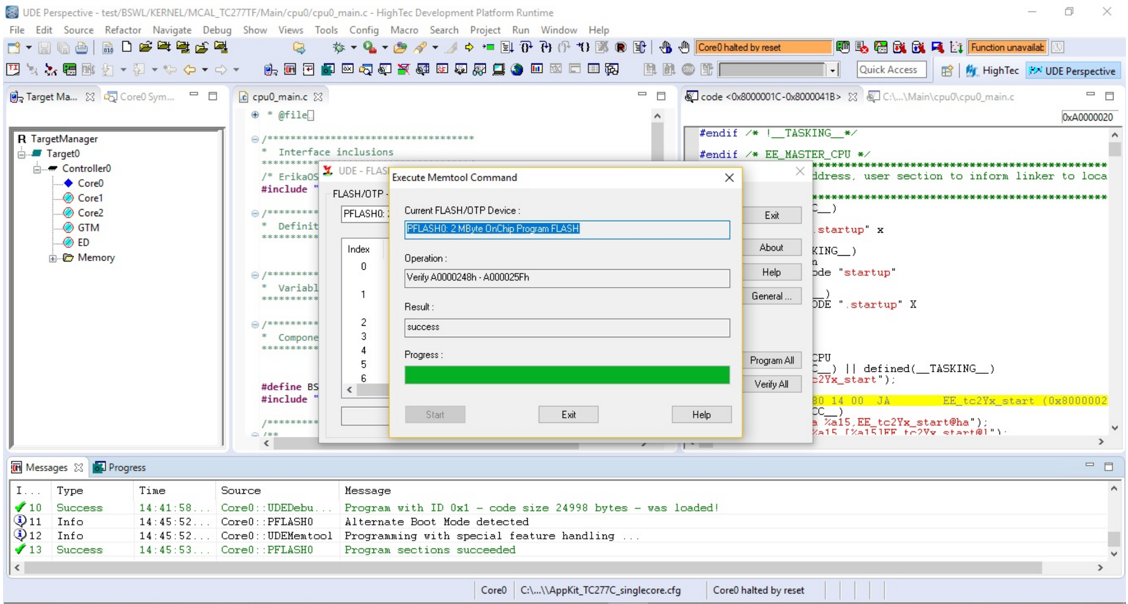


Figure 3.14. *Eclipse: Programming Success*

7. From the *Debug* menu, select *Step over subroutine*. At this moment your application is executing but stopped on the function *main()*. This means the C startup code has been executed completely. The Editor view shows the C source files of your application and a yellow arrow shows the line where the execution has stopped. To run your application, select *Start Program Execution* from the Debug menu and to restart your application, select *Restart Program Execution*, [14].

Chapter 4

Custom Simulink Library

A Simulink library is a collection of blocks that can be used to create instances of those blocks in a Simulink model. It's possible to create instances of blocks from existing Simulink libraries, or to create custom libraries in order to group and maintain instances of the own blocks in models. The installed libraries can be accessed from the Simulink Library Browser and it's not possible to modify them. Instead, if customized blocks want to be created, the user can create custom library with custom blocks and add it to the Library Browser.

In this chapter will be explained how to create a new Simulink Library (section 4.2) and fill it with custom block (section 4.3), with the help of some example. A script is also provided (section 4.4) to deploy the Library in order to find it in the Library Browser. The last paragraph (section 4.5) contains a description of all the basic blocks of the *Aurix/Arduino-like BSP* library.

4.1 Software resources

The following tools are needed:

- Matlab
- Simulink

4.2 Simulink Library Creation

1. Start Simulink.

2. *C*lick on *New* pane → *Blank Library*.
   Now Simulink displays a new window, labeled as *Library: untitled*.

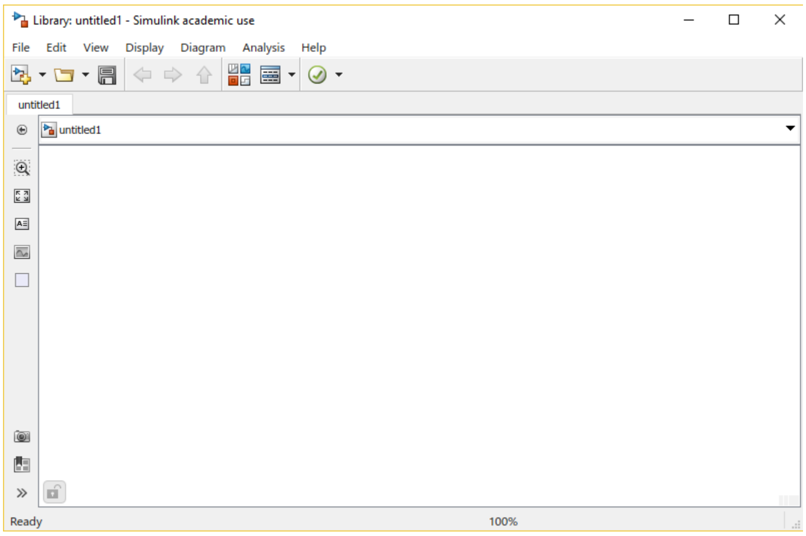


Figure 4.1. Blank Library

A blank library is now available and so it's possible to drag basic block.

The new library can appear in the Simulink Library Browser only if the model property *EnableLBRepository* is on when the library is saved. Run the following command in the MATLAB command prompt:

set_param(gcs,'EnableLBRepository','on');

Now the Library can be saved and named (*e.g. Tricore_tc277c.slx*).

4.3 Simulink Block Generation

Matlab Legacy Code Tool is used in order to create Simulink blocks: the tool transforms existing C (or C++) functions into C MEX S-functions that can be included in Simulink models.

The following diagram illustrates a general procedure for using the Legacy Code Tool:

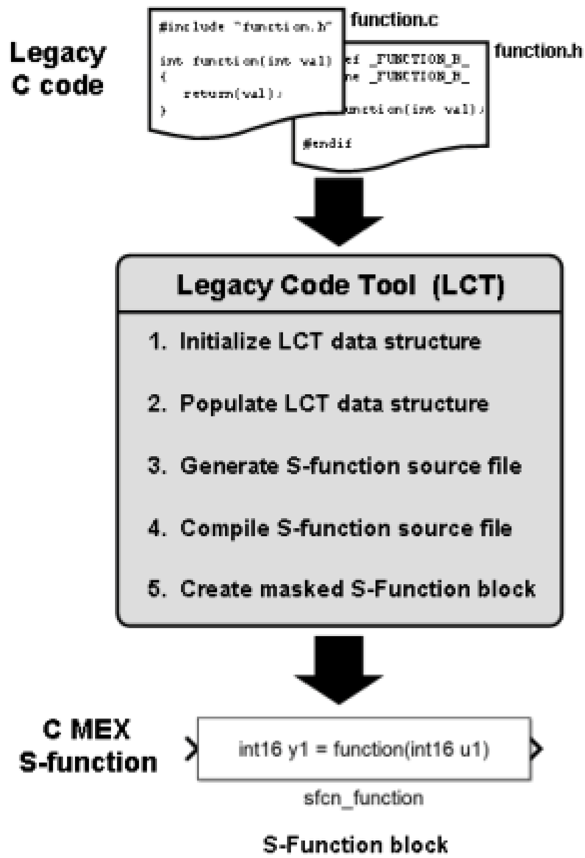


Figure 4.2. *Diagram showing the correct use of Legacy Code Tool, [4]*

An example is provided in order to better explain how to integrate an existing C function into a Simulink model using Legacy Code Tool. It's possible to create a MATLAB script containing the following commands in the same order or to run the commands one by one on the MATLAB command line:

1. ***def=legacy_code('initialize');*** initialize a Matlab struct *def* with fields that represent Legacy Code Tool properties;

```
def =

  struct with fields:

                    SFunctionName: ''
    InitializeConditionsFcnSpec: ''
                    OutputFcnSpec: ''
                     StartFcnSpec: ''
                 TerminateFcnSpec: ''
                      HeaderFiles: {}
                      SourceFiles: {}
                     HostLibFiles: {}
                   TargetLibFiles: {}
                         IncPaths: {}
                         SrcPaths: {}
                         LibPaths: {}
                       SampleTime: 'inherited'
                          Options: [1×1 struct]
```

2. ***def.HeaderFiles = {'header_file.h'};*** the header file that contains the function declaration;

3. ***def.SFunctionName = 'function_name';*** specifies a name for the S-function;

4. ***def.OutputFcnSpec = 'return-spec = function-name(argument-spec)';*** defines the function that the S-function calls at each time step, where

    - *return-spec* defines the data type and variable name for the return value of the existing C function. If the function does not return a value, the return specification can be omitted or defined as as void. Otherwise the data type (i.e. uint8, uint16 ect) must be followed by a token of the form y1, y2, ..., yn, where n is the total number of output arguments, [4];
    - *function-name* is the function name and must be the same of the existing C function name;
    - *argument-spec* defines one or more data type (i.e. uint8, uint16 etc) and token pairs that represent the input, output, parameter, and work vector arguments of the existing C function. The function input and output arguments map to block input and output ports and parameters map to workspace parameters, [4]. Token can have the following forms:
        - Input — u1, u2, ..., un, where n is the total number of input arguments;

- Output — y1, y2, ..., yn, where n is the total number of output arguments;
- Parameter — p1, p2, ..., pn, where n is the total number of parameter arguments;
- Work vectors (persistent memory) — work1, work2, ..., workn, where n is the total number of work vector arguments.

An example is the following:
def.OutputFcnSpec = 'uint8 y1 Dio_READ_Channel(uint32 p1), that returns as output one *uint8* value and receives as input one *uint32* parameter.

5. ***legacy_code('sfcn_cmex_generate', def);*** generate an S-function source file from the existing C function.

6. ***legacy_code('sfcn_tlc_generate', def)*** generates the TLC file, needed to recognize the blocks of the created S-function from Embedded Coder during the Code Generation Process.

7. ***legacy_code('compile', def);*** compile and link the S-function source file into a dynamically loadable executable for Simulink.

8. ***legacy_code('slblock_generate', def);*** generate masked S-function blocks that call the S-functions. The software places the blocks in a new model. From there you can copy them to an existing model.

The following script is an example used in order to generate the *CCU6_PWM_Setup* block (figure 4.3):

```
def.HeaderFiles = {'aswl_if.h', 'CCU6_if.h'};
def.SFunctionName = 'CCU6_PWM_Setup';
def.OutputFcnSpec = 'void CCU6_PWM_Setup(uint8 p1, double p2, double p3)';
legacy_code('sfcn_cmex_generate', def);
legacy_code('sfcn_tlc_generate', def);
legacy_code('compile', def);
legacy_code('slblock_generate', def);
```

Look at the *def.HeaderFiles*: in the creation of the basic blocks for *Aurix/Arduino-like* board, it was necessary to add also the header file *aswl_if.h* containing the basic type declaration in order to avoid compilation error and so "manual" intervention after the code generation.

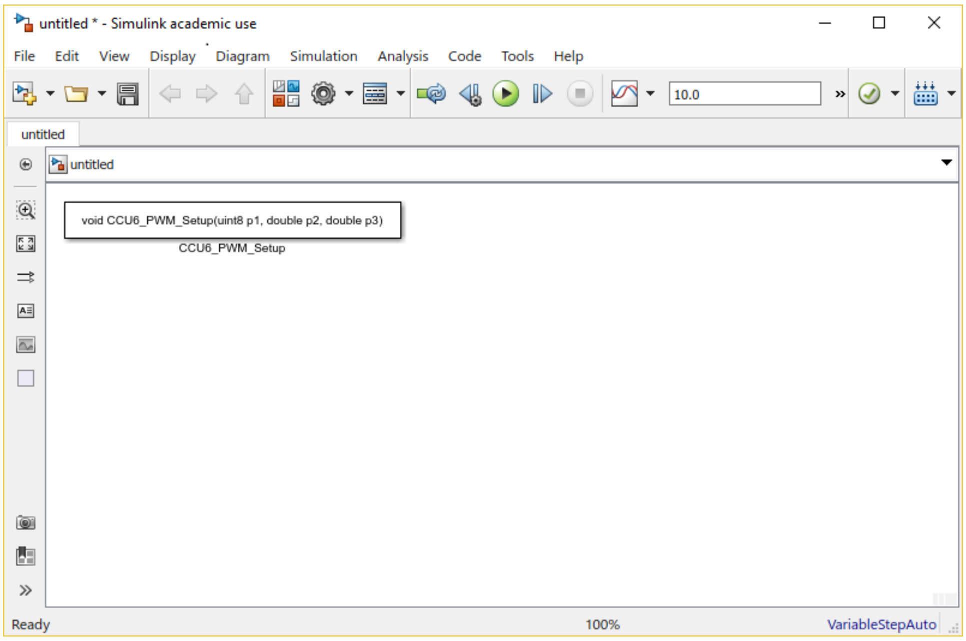


Figure 4.3. *CCU6_PWM_Setup block*

Now it's possible to create/edit a block mask. A mask is a custom user interface for a block that hides the block's contents, making it appear to the user as a block with its own icon and parameter dialog box.

The following example is provided in order to better explain how to design a mask:

1. Right click on the created block and go to *Mask-Edit Mask...*

   The Mask Editor appears: it's a dialog box that helps to create and customizes the block mask. It contains a set of dialog box, [4]:

   - *Icon & Ports* pane helps you to create a block icon that contains descriptive text, state equations, image, and graphics;
   - *Parameters & Dialog* pane enables you to design mask dialog boxes using the dialog controls in the Parameters, Display, and Action palettes;
   - *Initialization* pane allows you to add MATLAB commands that initialize the masked block;
   - *Documentation* pane enables you to define or modify the type, description, and help text for a masked block;

2. Click on *Parameters & dialog* tab to design mask dialog boxes

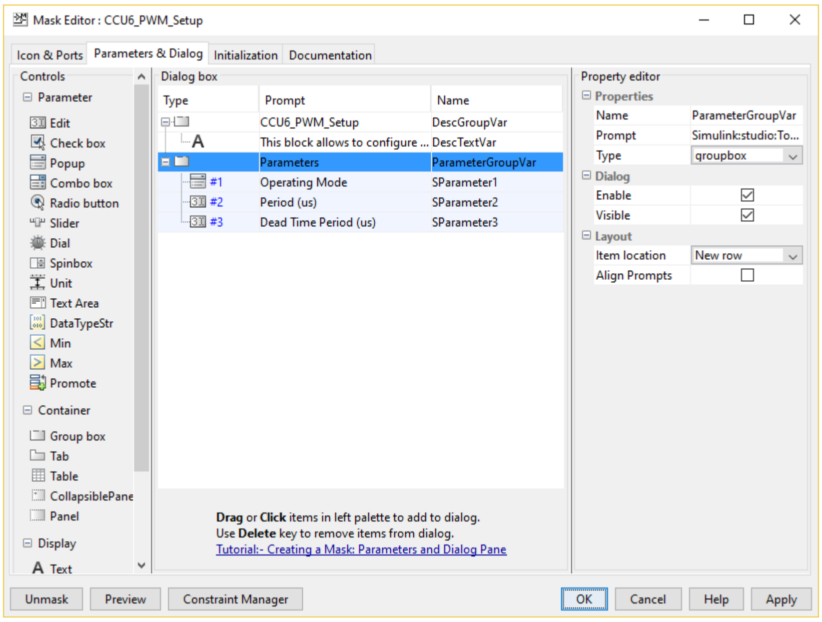


Figure 4.4. *PARAMETERS & DIALOG pane window*

It's possible to add a mask name (*e.g. CCU6_PWM_Setup*), a mask description (*e.g. "This block allows to configure..."*), and set the parameters (*i.e.* user inputs that take part in simulation) configuration. In the example the *SParameter1* (i.e. *Operating Mode*) is defined as *popup* type that allows to select a parameter value from a list of possible values. *SParameter1* and *SParameter2* (*i.e. Period* and *Dead Time Period*) are defined as *edit* type because have no fixed assumable values. It's also possible to add constraints in order to avoid the insertion of unreasonable value for the specified parameters (go on *Constraint* and select *Add New Constraint*). At the end click on *Apply* and then *Ok*. The mask is ready to be used.

3. Double click on the block and the new mask appear (figure 4.5).

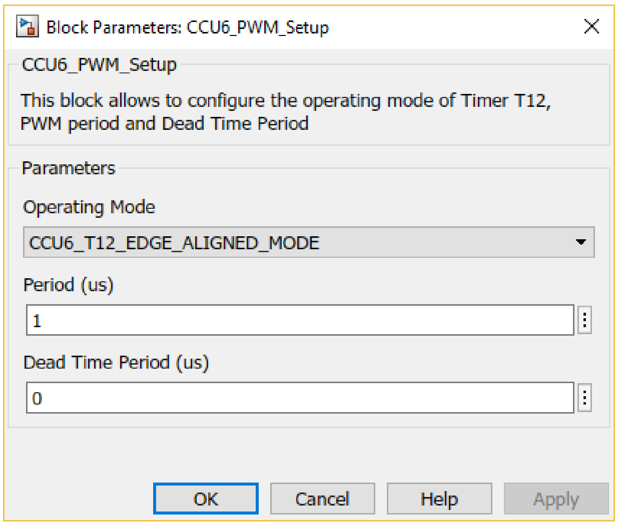


Figure 4.5. *Block Mask*

4.4 Add Libraries to the Library Browser

When the Custom Library is filled with all the needed Simulink blocks, it's time to add the new Library to the Library Browser. This process is useful in order to find easily the new blocks. In order to do so, the following script named as *slblocks.m*, must be run:

```
function blkStruct = slblocks
        % This function specifies that the library should appear
        % in the Library Browser
        % and be cached in the browser repository

        Browser.Library = 'Tricore_tc277c';
        % 'Tricore_tc277c' is the name of the library

        Browser.Name = 'Aurix Arduino-like BSP';
        % 'Aurix Arduino-like BSP' is the library name that appears
             % in the Library Browser

        blkStruct.Browser = Browser;
```

Open the Library Browser and refresh to see the new library. Right-click the library list and select *Refresh Library Browser*.

Custom library can be used in an identical way of any other Simulink library: blocks are dragged from a library and placed into a model in the usual way.

4.5 Aurix/Arduino-like Simulink Library Description

The Simulink Library for *Aurix/Arduino-like* board contains the basic blocks for the configuration and use of GPIO Ports, ADC, CAN and PWM. The following subsections are introduced by a brief description of the modules and followed by the description of all the basic blocks.

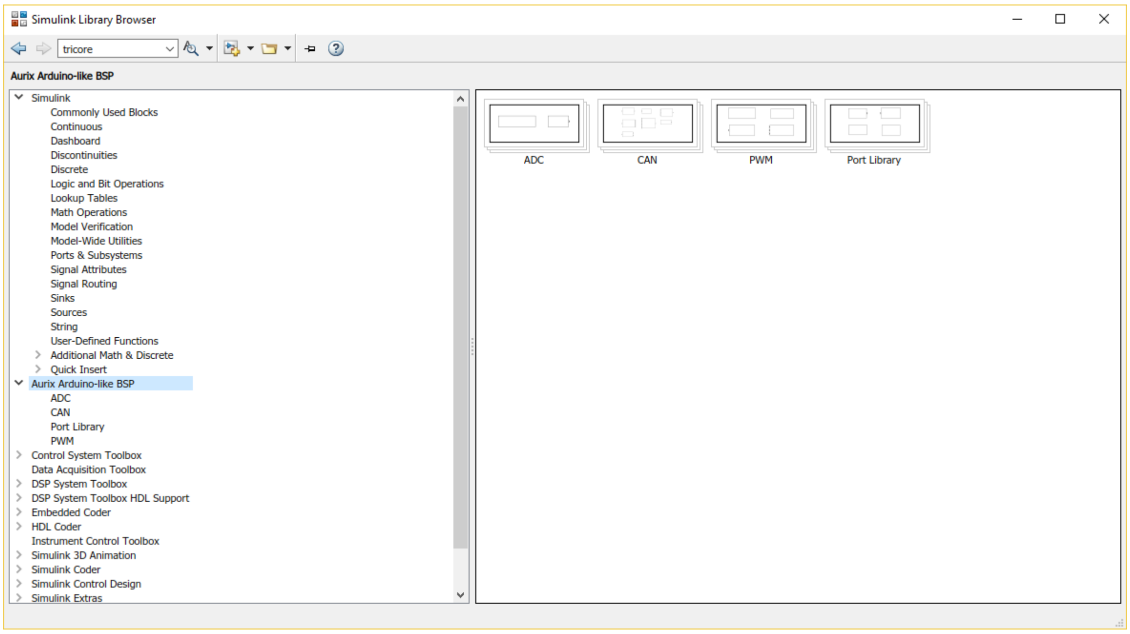


Figure 4.6. *Aurix/Arduino-like Simulink library*

4.5.1 GPIO Ports

Aurix TriCore TC277 has digital General Purpose Input/Output (GPIO) port lines which are connected to the on-chip peripheral units.

- ***PORT_00_34_Conf***: This block allows to set the right pin configuration of PORT 00 - 34 (figure 4.7).

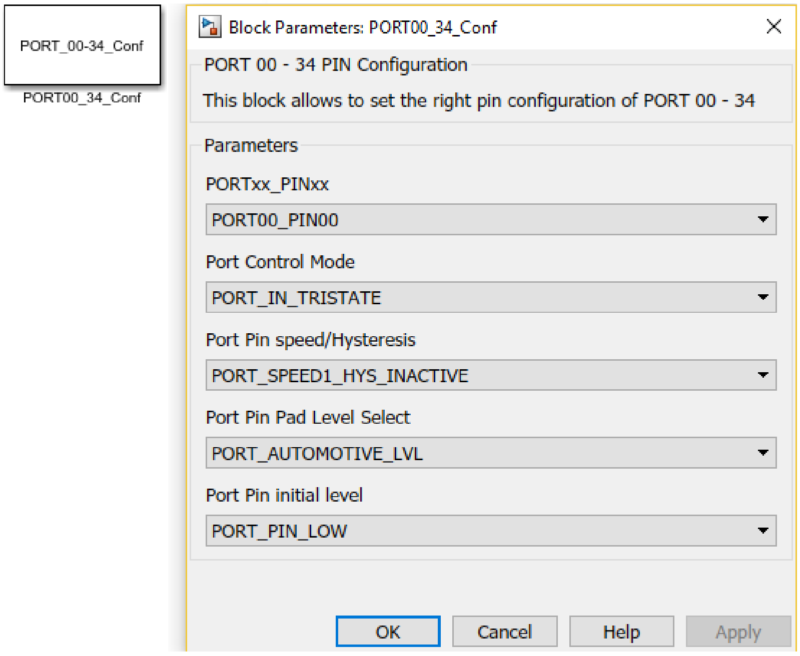


Figure 4.7. *PORT_00_34_CONF block mask*

Parameters:

- *PORTxx_PINxx* allows to select the desired port pin;
- *Port Control Mode* determines the port line functionality. When a port line is configured as input, its hysteresis function can be activated/inactivated. When a port line is configured as output, its speed grade can be configured;
- *Port Pin speed/Hysteresis* allows to choose the speed grade when port lines are configured as output, and determines if hysteresis is active or inactive when port lines are configured as input;
- *Port Pin Pad Level Select* allows to select the pad level;
- *Port Pin initial level* determines the port pin initial level.

• ***PORT_40_Conf***: This block allows to set the right pin configuration of PORT40. PORT40 is an input port only (figure 4.8).

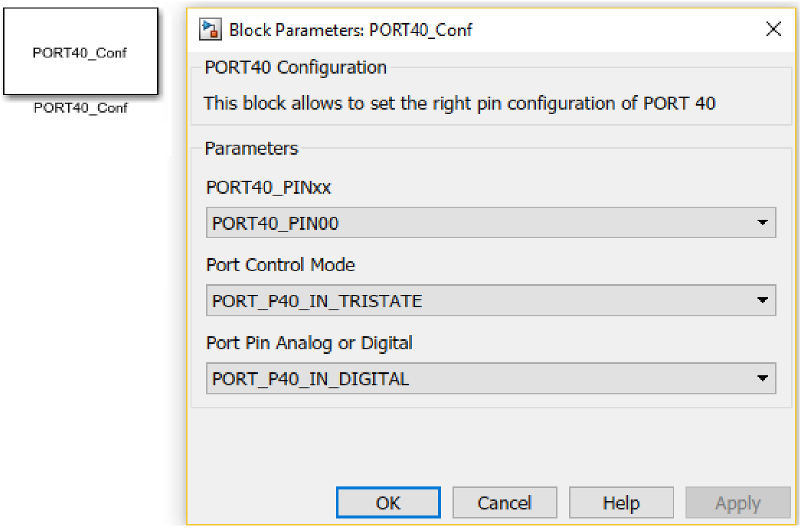


Figure 4.8. *PORT_40_CONF block mask*

Parameters:

- *PORT40_PINxx* allows to select the desired PORT40 pin;
- *Port Control Mode* determines the port line functionality;
- *Port Pin Analog or Digital* allows to choose Analog or Digital input.

- ***Dio_READ_Channel***: This block returns as output the digital value (*unsigned* 8 bits value) of the selected port pin.

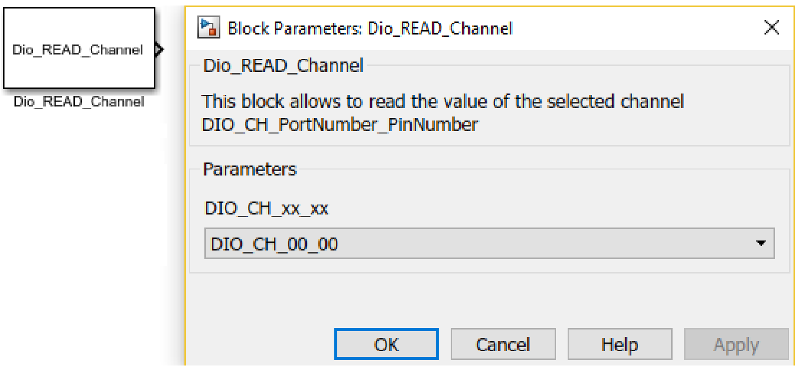


Figure 4.9. *Dio_READ_Channel block mask*

Parameters:

– *DIO_CH_xx_xx* allows to select the desired digital channel.

- ***Dio_WRITE_Channel***: This block receives as input the digital value (*unsigned* 8 bits value) to be written on the port pin.

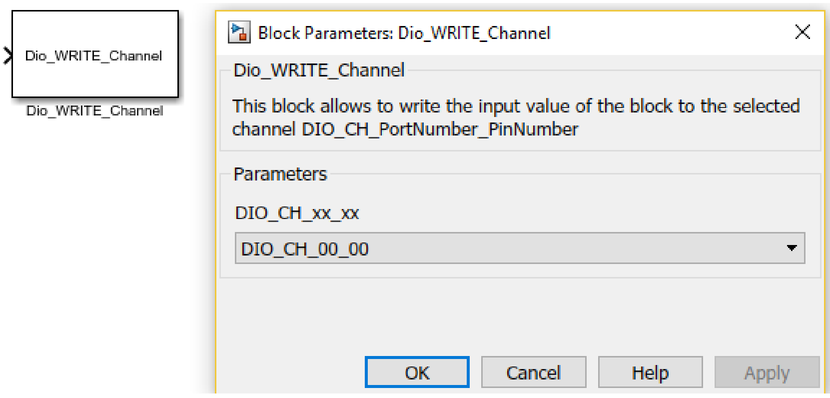


Figure 4.10. *Dio_WRITE_Channel block mask*

Parameter:

– *DIO_CH_xx_xx* allows to select the desired digital channel.

4.5.2 ADC

Aurix TriCore TC277 provides a series of analog input channels connected to a cluster of Analog/Digital Converters using the Successive Approximation Register (SAR) principle to convert analog input values (voltages) to discrete digital values.

Now only 5 ADCs channels have been configured: 2 channels of ADC group 2 and 3 channels of ADC group 7 are available. The Port Pins configured to work as ADCs inputs are: *AnalogInA4 (ADC2.0)*, *AnalogInA5 (ADC2.1)*, *PORT00_PIN04 (ADC7.2)*, *PORT00_PIN03 (ADC7.3)* and *PORT00_PIN02 (ADC7.4)*. The *Port Control Mode* of the last three Port Pins must be configured as *PORT_IN_TRISTATE*. ADCs resolution is 12 bit.

- ***Adc_StartBackgroundConvertion***: This block starts or stops the background conversions on all channels of all groups of the ADC module (figure 4.11).

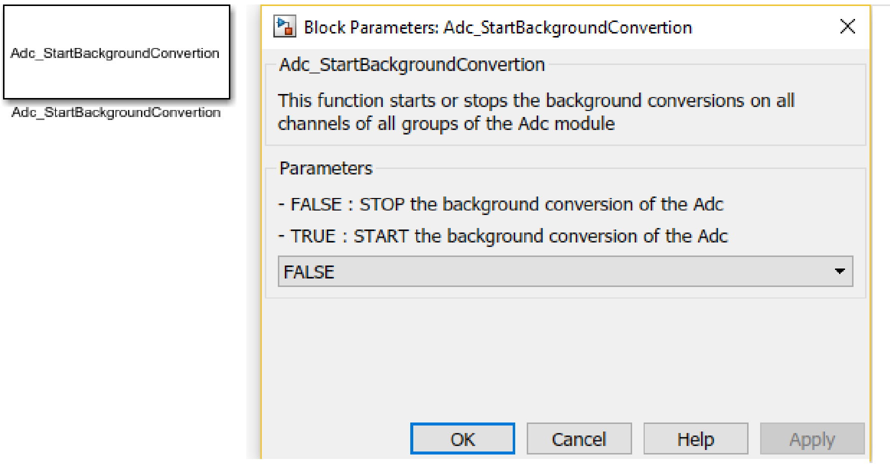


Figure 4.11. *Adc_StartBackgroundConvertion block mask*

Parameter:

– *FALSE* or *TRUE* in order to disable or enable the background conversion, respectively;

- ***Adc_Read***: This block returns as output the converted value in bit (*unsigned* 12 bits) of the selected ADC group channel.

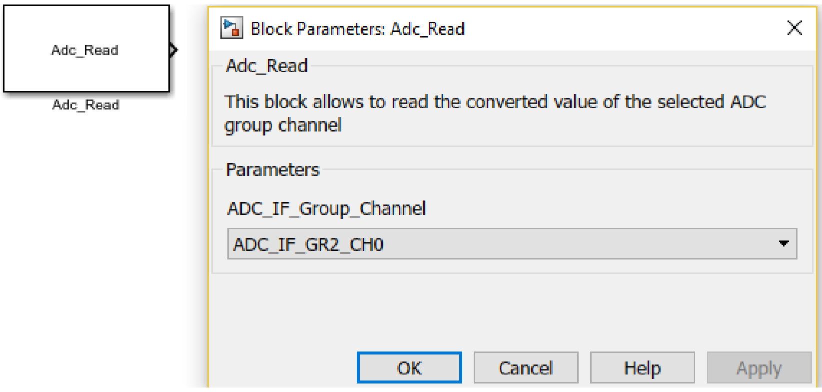


Figure 4.12. *Adc_Read block mask*

Parameter:

– *ADC_IF_Group_Channel* allows to select the desired ADC group channel.

4.5.3 CAN

CAN is an asynchronous serial bus system with one logical bus line. It has an open, linear bus structure with equal bus participants called nodes. A CAN bus consists of two or more nodes.

WARNING: Before using CAN blocks the following command must be written in the Matlab console in order to import *Can_tMsg* type:

Simulink.importExternalCTypes('aswl_if.h')

"aswl_if.h" file is located in the *Getting_Started* folder and it contains *Can_tMsg* type definition.

An example of CAN message structure is the following:

Can_tMsg CanMsg= {
.u8Node = 1,
.u8Length = 8,
.u8Data = {12,0,0,0,0,0,0,0,},
.u8Standard_Extended = 0,
.u32ID = 0x1800D0C0,
};

- ***Can_Msg_Static***: This block allows to create a Static *Can_tMsg*, *i.e.* a structure that cannot be modified runtime.

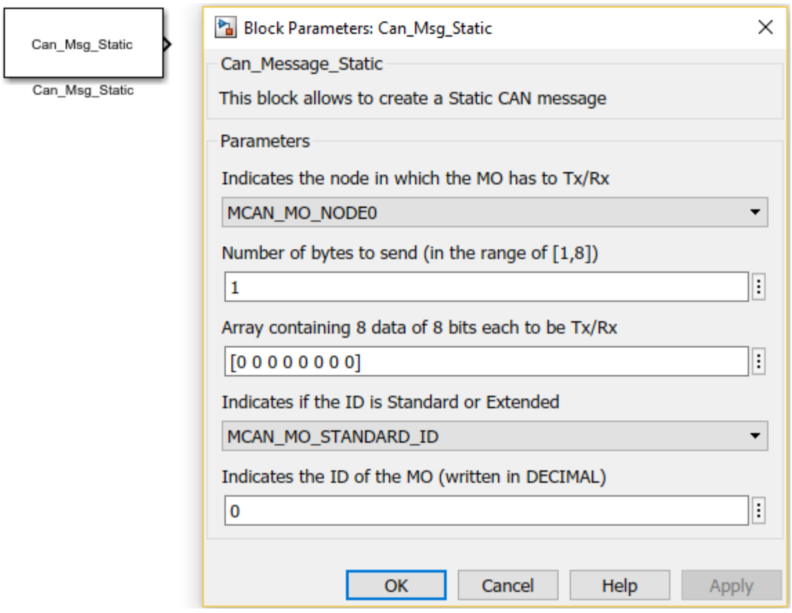


Figure 4.13. *Can_Msg_Static block mask*

Parameters:

- *1st param* indicates the node in which the MO has to Tx/Rx (**WARNING**: only MCAN_MO_NODE1 is configured);
- *2nd param* represents the number of bytes to send (in the range of [1,8]);
- *3rd param* is an array containing 8 data of 8 bits each to be Tx/Rx;
- *4th param* indicates if the ID is Standard or Extended;
- *5th param* indicates the ID of the MO (**WARNING**: the value must be written in DECIMAL);

- ***Can_Msg_Dynamic***: This block returns as output a *Can_tMsg* struct according to the input values, so it's possible to modify the message structure runtime. The order of the inputs values is the same used for *Can_Msg_Static*.

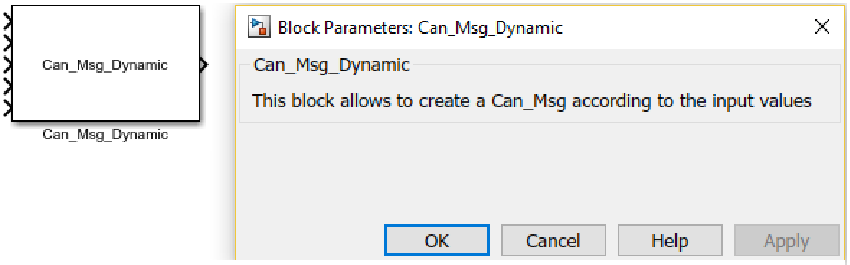


Figure 4.14. *Can_Msg_Dynamic block mask*

- ***Can_Msg_unpacked***: This block allows to unpack a *Can_tMsg*. The outputs are respectively the number of useful bytes and a pointer to the array of data.

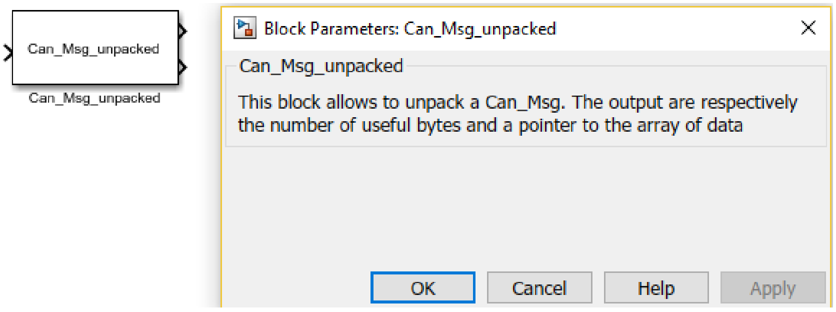


Figure 4.15. *Can_Msg_unpacked block mask*

- ***Packed_Can_8bytes_array***: This block receives as inputs 8 values of 8 bits each and returns as output a pointer to an array containing those values.

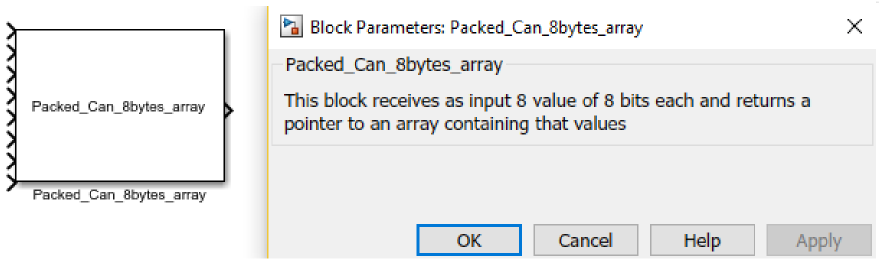


Figure 4.16. *Packed_Can_8bytes_array block mask*

- ***UnPacked_Can_8bytes_array***: This block receives as input a pointer to an 8 byte array and returns as outputs the values of each single byte.

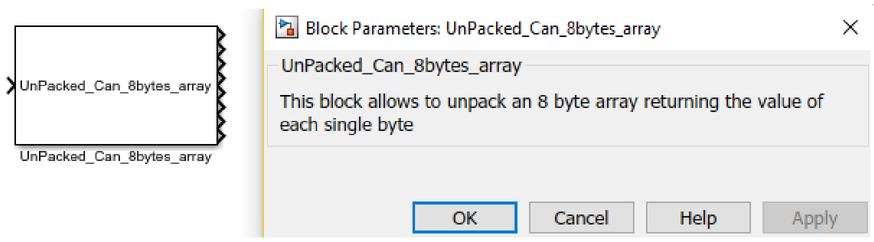


Figure 4.17. *UnPacked_Can_8bytes_array block mask*

- ***Can_Send***: This block allows to send a *Can_tMsg* received as input.

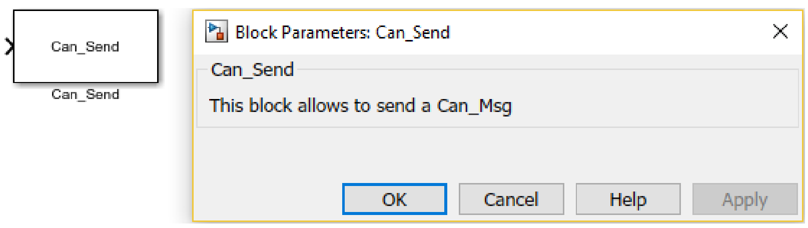


Figure 4.18. *Can_Send block mask*

- ***Can_Receive***: This block receives as input a *Can_Msg_Static* and returns as output a *Can_tMsg* filled with the received values (figure 4.19).

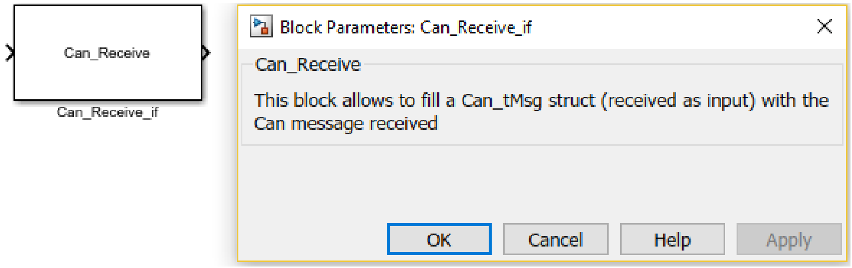


Figure 4.19. *Can_Receive block mask*

4.5.4 PWM

PWM (*i.e.* Pulse-width modulation) is a type of digital modulation that allows obtaining a variable average voltage depending on the ratio between the duration of the positive and the negative pulse (duty cycle).

It's possible to generate PWM using two different modules: ATOM and CCU6. The first one has to be used when single PWM has to be generated while the second one when 3-phase PWM has to be generated.

ATOM

The ARU-connected Timer Output Module (ATOM) is able to generate complex output signals without CPU interaction due to its connectivity to the ARU. In ATOM Signal Output Mode PWM (SOMP) configuration, the ATOM submodule channel is able to generate complex PWM signals with different duty cycles and periods, [8].

Now only the 7 channels of the ATOM0 has been configured. The related Port Pins are: *PORT22_PIN01 (ATOM0_0)*, *PORT22_PIN00 (ATOM0_1)*, *PORT23_PIN05 (ATOM0_2)*, *PORT20_PIN01 (ATOM0_3)*, *PORT22_PIN03 (ATOM0_4)*, *PORT23_PIN00 (ATOM0_5)* and *PORT23_PIN01 (ATOM0_6)*. The *Port Control Mode* of the Port Pins must be configured as *PORT_OUT_PUSHPULL_ALT_1*.

- ***Atom_PWM_Channel_Config***: This block allows to set the desired configuration for the Atom Channel (figure 4.20).

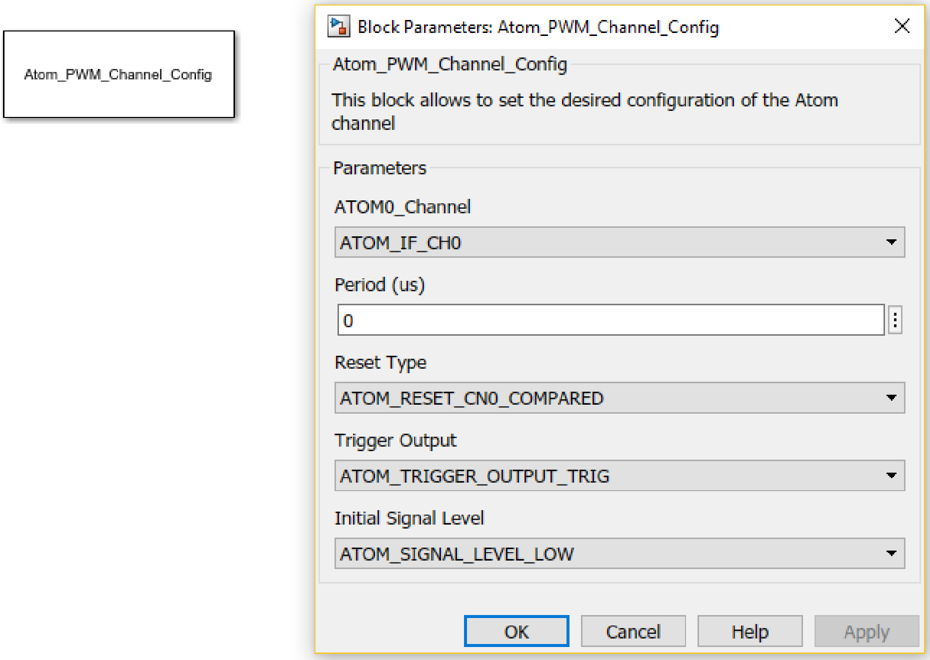


Figure 4.20. *Atom_PWM_Channel_Config block mask*

Parameters:

- *ATOM0_Channel*: Represents the ATOM0 channel that has to be configured;
- *Period (us)*: Sets the PWM period. The value must be written in microseconds;
- *Reset Type*: Allows to select the reset source of CCU0 (Counter Compare Unit 0). It's possible to reset counter register CN0 to 0 on matching comparison with compare value CM0; or when signaled by the trigger signal TRIG_[x-1] of the preceding channel [x-1];
- *Trigger Output*: Defines trigger output selection of module ATOM0_CHx;
- *Initial Signal Level*: Defines if the initial Signal Level is Low or High.

- ***Atom_PWM_SetDutyCycle***: This block allows to set the duty cycle (received as block input) of the selected PWM channel (figure 4.21).

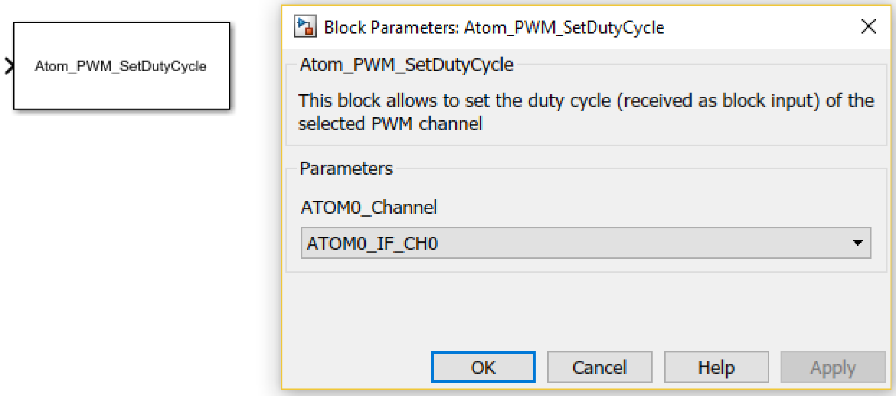


Figure 4.21. *Atom_PWM_SetDutyCycle block mask*

Parameter:

– *ATOM0_Channel*: Represents the ATOM0 channel that has to be configured.

CCU6

The CCU6 unit is made up of a Timer T12 Block with three capture/compare channels and a Timer T13 Block with one compare channel.

The timer T12 block is the main unit to generate the 3-phase PWM signals. A 16-bit counter is connected to 3 channel registers via comparators, that generate a signal when the counter contents match one of the channel register contents. Besides the 3-phase PWM generation, the T12 block offers options for dead-time control.

The related Port Pins are: *PORT02_PIN00 (CC60)*, *PORT34_PIN03 (COUT60)*, *PORT34_PIN04 (CC61)*, *PORT02_PIN03 (COUT61)*, *PORT33_PIN14 (CC62)* and *PORT33_PIN15 (COUT62)*. The *Port Control Mode* of the Port Pins must be configured as *PORT_OUT_PUSHPULL_ALT_7*.

- ***CCU6_PWM_Setup***: This block allows to configure the operating mode of Timer T12, PWM period and dead-time period (figure 4.22).

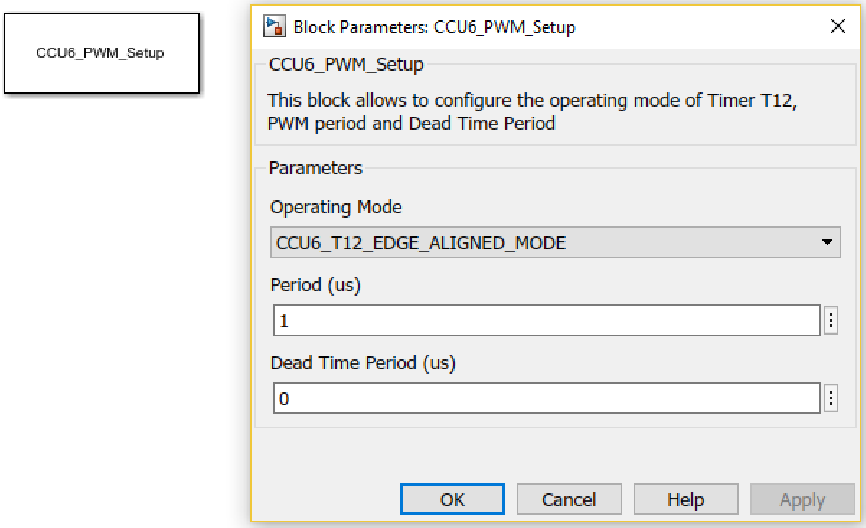


Figure 4.22. *CCU6_PWM_Setup block mask*

Parameters:

- *Operating Mode*: It's possibile to select among Edge Aligned Mode (timer T12 is always counting upwards) and Center Aligned Mode (timer T12 is counting upwards or downwards in order to have a triangular shape);
- *Period (us)*: Sets the PWM period. The value must be written in microseconds;
- *Dead Time Period*: Sets the Dead Time Period. The value must be written in microseconds. No Dead Time insertion if the value inserted is 0.

The CCU6 input clock is 100 MHz so the smallest value that can be entered for both periods is 0.01 us (refers to section 2.2.2).

- ***CCU6_PWM_SetDutyCycle***: This block allows to set run-time the duty cycle of the 3 PWM main channels. The inputs values has to be *double* data types and represent the percentage of ON time (must be in the 0-100 range).

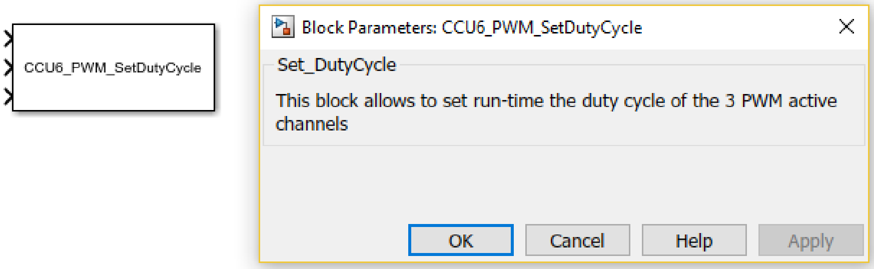


Figure 4.23. *CCU6_PWM_SetDutyCycle block mask*

Chapter 5

Code Generation

Target Language Compiler works with its target files and Real-Time Workshop output to produce code. When generating code from a Simulink model using Real-Time Workshop, the first step in the automated process is to generate a *model.rtw* file. This file includes all of the model-specific information required for generating code from the Simulink model. *model.rtw* is passed to the Target Language Compiler, which uses it in combination with a set of included system target files and block target files to generate the code.

In order to allow Simulink to find the custom System Target File used to generate code for ERIKA RTOS, *erika_rtos* [15] folder (present inside *Getting_Started* folder) must be placed in the MATLAB root folder inside `\rtw\c`. Moreover the *Matlab* folder present in the *Getting_Startedl* folder must become the MATLAB working directory: it contains the S-function source files and the TLC files needed to recognize the blocks of the *Aurix/Arduino-like BSP* Simulink library from Embedded Coder during the Code Generation Process. Without these informations, the *Aurix/Arduino-like BSP* blocks will not be recognized by the tool.

In this chapter will be explained how to create a Simulink model using the *Aurix Arduino-like BSP* library and how to use Embedded Coder to generate code, with the help of some examples (sections 5.2, 5.3). The generated code will be also analyzed in order to better understand the main part. The last section (5.4) is dedicated to the integration of the generated file in the project.

5.1 Software Resources

The following tools are needed:

- Matlab
- Simulink
- Embedded Coder

5.2 First Example: Blinking Led

Generate a task that set ON and OFF a LED with a period of 1 s (500 ms ON and 500 ms OFF)

1. Start Simulink.
2. *C*lick on *New* pane → *Blank Model* and a new window appears.
3. Open the Simulink Library Browser and found *Subsystem* block; name it with the decided task name (*e.g.* *Blinking_Led*) and delete everything inside.

Figure 5.1. *Subsystem Block*

4. Right click on *Subsystem* block → *Block Parameters (Subsystem)* and then select *Main* pane in order to modify the Subsystem parameters: select *Treat as atomic unit* and set the desired *Sample time* in seconds: in this case 0.5 must be written (*e.g.* 500 ms)(figure 5.2).

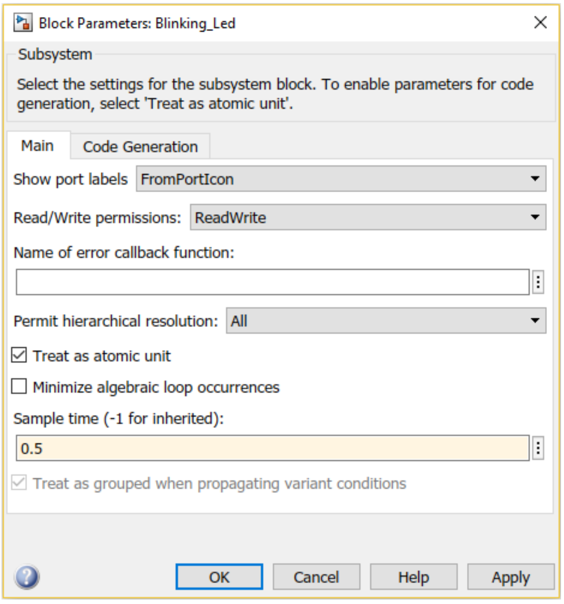


Figure 5.2. *Subsystem Block Parameters: Main pane*

Now select the *Code Generation* pane and set the *Function packaging* to *Nonreusable function* and *Function name options* to *Use Subsystem name*.

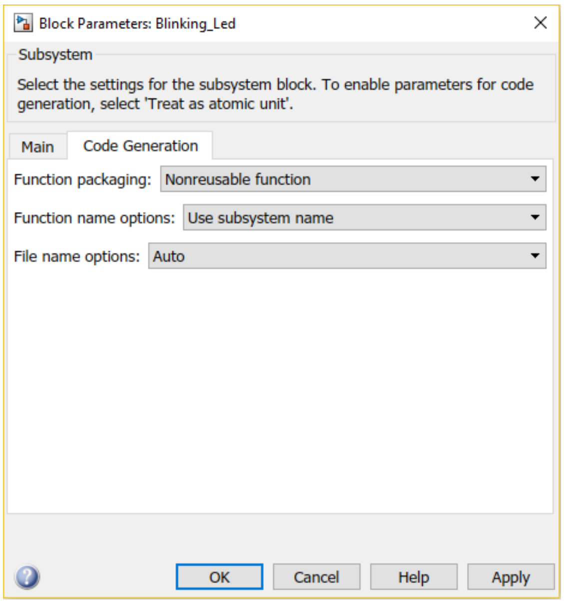


Figure 5.3. *Subsystem Block Parameters: Code generation pane*

5. Drag inside the *Subsystem* block the *Initialize Function* from the Simulink Library Browser and delete everything inside it except for the *Event Listener*. This block is necessary because it's used to "contains" all the initialize block functions.

Figure 5.4. Initialize Function block

6. Open the Simulink Library Browser and find the *Aurix Arduino-like BSP* library. Drag the *PORT_00-34_Conf* block in the *Initialize Function* block and configure the *PORT10_PIN08* to work as *PORT_OUT_PUSHPULL_GPIO*. Look at the figure 5.5 to understand how to configure the other parameters.

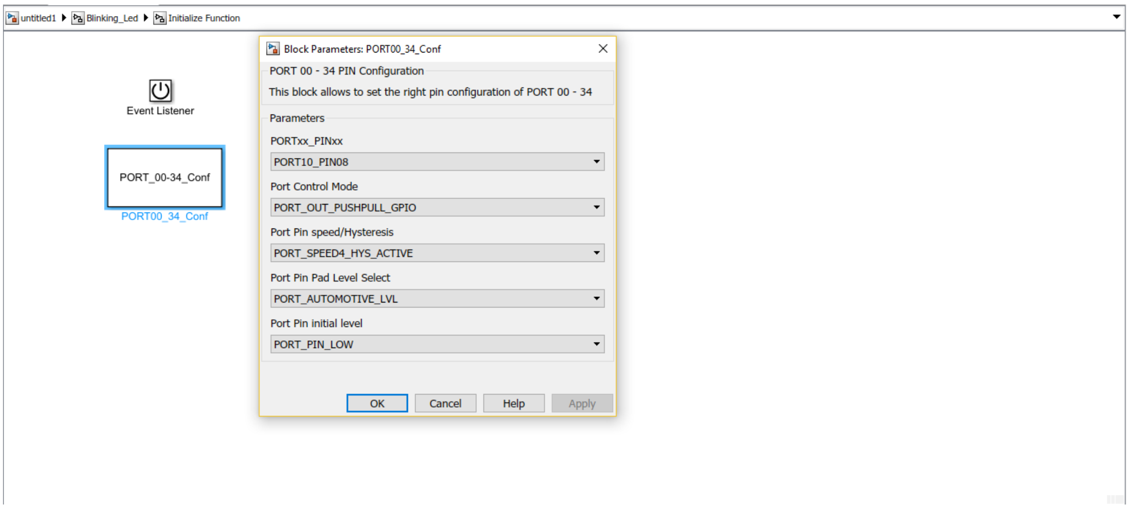


Figure 5.5. *Initialize Function: Port configuration*

7. Inside the *Subsystem* block the *Dio_WRITE_Channel* block is instantiated and configured to write a value on the desired channel, in this case the *DIO_CH_10_08*. This block is connected with some Simulink blocks: *Memory* block with 0 as initial condition, *NOT* block in order to complement the "memory" value and *Data Type Conversion* block to convert the value to the right data type (figure 5.6).

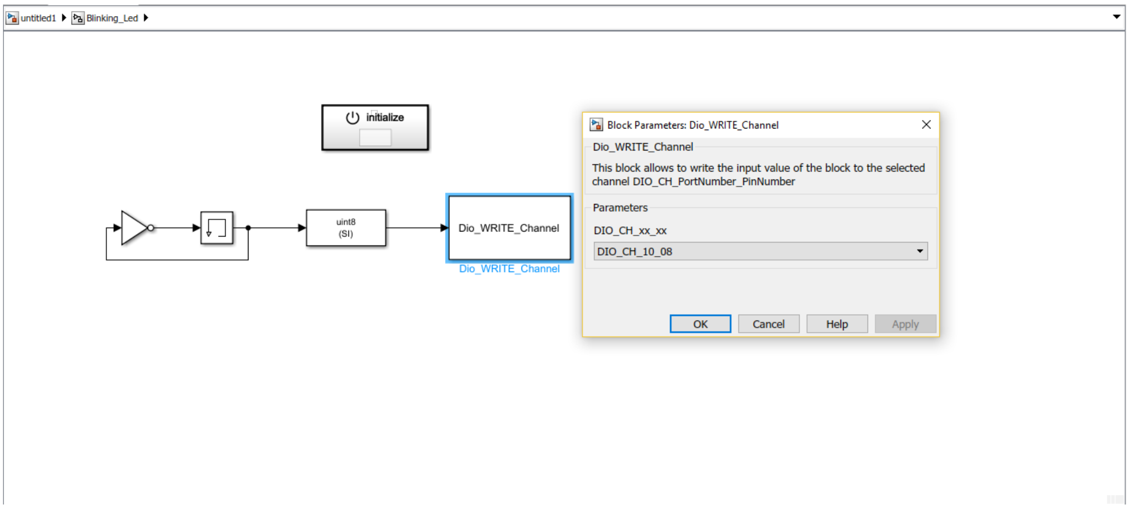


Figure 5.6. *Model Design*

8. Once the model is created, click on  (*i.e. Model Configuration Parameters*) to configure the Solver Type and set the System Target File:
   - *Solver*: Select *Fixed-step* in Solver selection;
   - *Hardware Implementation*: Select *Infineon* in Device vendor and *TriCore* in Device type; it's useful only to define the correct data type size;
   - *Code Generation*: Set System Target File as *mbd_erika_rtos.tlc*, select *Generate code only* and deselect *Generate makefile* (figure 5.7).

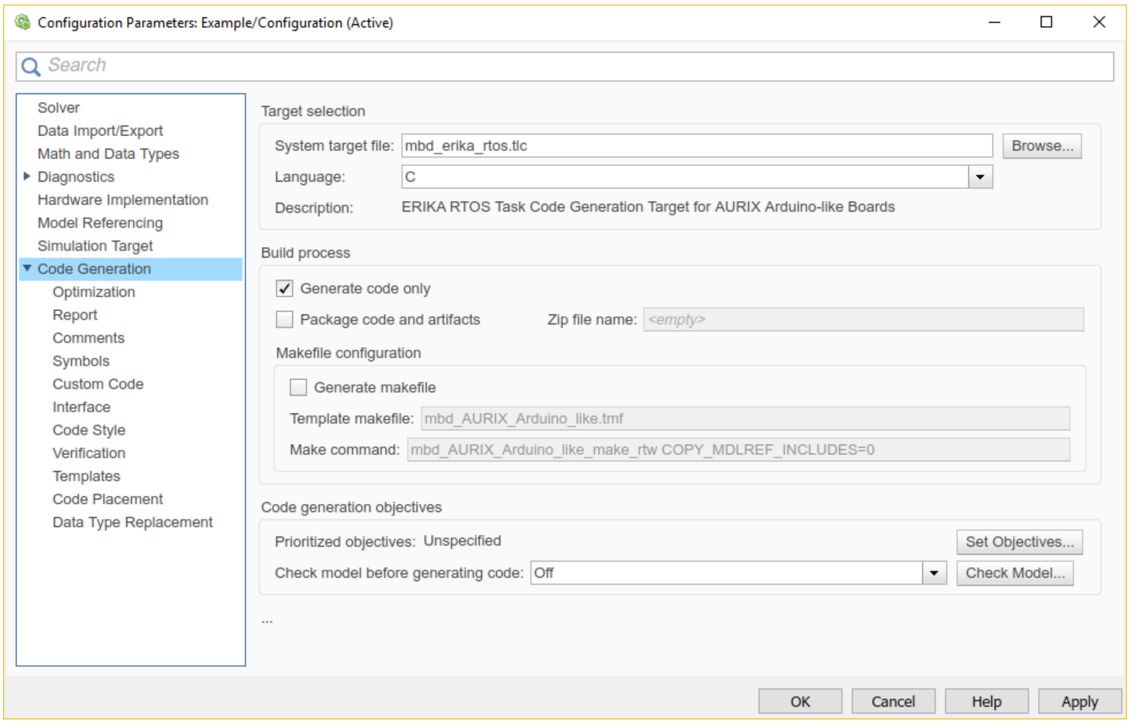
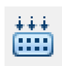


Figure 5.7. *Set parameters for the Code Generation*

9. *erika_rtos* folder must be added to MATLAB search path. Run the `addpath("path");` instruction on the MATLAB command line where the arguments "path" represents the position of *erika_rtos* folder: for example ***addpath('C:\ProgramFiles\MATLAB\R2018b\rtw\c\erika_rtos');***;

10. Save and name the model (*e.g. Example*) and then click on  to build the model.

If everything is done correctly, the *Code Generation Report* appears. The auto-generated files will be placed in the *"modelname"_task* folder (in this case *Example_task* folder) inside MATLAB working directory.

Have a look to the auto-generated files:

- **oscfg.oil.c** contains the ERIKA RTOS configuration. *Blinking_Led* task is here configured (figure 5.8):

```
TASK Blinking_LED {
  CPU_ID = "cpu0";
  PRIORITY = 1;
  AUTOSTART = TRUE;
  STACK = PRIVATE {
    SYS_SIZE = 512;
  };

  ACTIVATION = 1;
  SCHEDULE = FULL;

  /* Event managed by the Task */
  EVENT = ScheduleEvent_Blinking_LED;
};

EVENT ScheduleEvent_Blinking_LED {
  MASK = AUTO;
};
```

Figure 5.8. *Oil file: Task configuration*

The ALARM is configured in order to generate the EVENT *ScheduleEvent_Blink_LED* that wake up the task every CYCLETIME (in this case 500 ms).

```
ALARM Alarm_Blinking_LED {
  COUNTER = system_timer_cpu0;
  ACTION = SETEVENT {
    TASK = Blinking_LED;
    EVENT = ScheduleEvent_Blinking_LED;
  };

  AUTOSTART = TRUE {
    ALARMTIME = 250;
    CYCLETIME = 500;
  };
```

Figure 5.9. *Oil file: ALARM configuration*

- **OSTasks0_ErikaOs.c** contains the extented task implementation (figure 5.10). Task is waiting for the "arrive" of the event configured in the .oil file, so in this case it is the *ScheduleEvent_Blink_LED* event generated by the ALARM: the step-function (*i.d. Example_Blinking_LED()*) is called every 500 ms.

```c
/* Task definition: Blinking_LED */
unsigned int volatile Blinking_LED_count;
TASK(Blinking_LED)
{
  EventMaskType mask;

  /* Task Body */
  while (1) {
    WaitEvent(ScheduleEvent_Blinking_LED);
    GetEvent(Blinking_LED, &mask);
    if (mask & ScheduleEvent_Blinking_LED) {
      ClearEvent(ScheduleEvent_Blinking_LED);

      /* call MBSD auto-generated code step function */
#ifdef __BSWL_COMPILE__

      Example_Blinking_LED();

#endif

      /* Increment execution Counter */
      Blinking_LED_count++;
    }
  }
```

Figure 5.10. *Extended task implementaion*

- The main function in ***Example.c*** are the initialize function (that "calls" all the function blocks inside the *Initialize Function*) *Example_Blinking_LED_Init()* (figure 5.11) and the *Example_Blinking_Led()* (figure 5.12) corresponding to the step function

```c
/* System initialize for atomic system: '<Root>/Blinking_LED' */
void Example_Blinking_LED_Init(void)
{
  /* SystemInitialize for Atomic SubSystem: '<S1>/Initialize Function' */

  /* S-Function (PORT00_34_Conf): '<S2>/PORT00_34_Conf1' */
  PORT00_34_Conf(39U, ((uint8_T)4U), ((uint8_T)4U), ((uint8_T)1U), ((uint8_T)1U));

  /* End of SystemInitialize for SubSystem: '<S1>/Initialize Function' */
}
```

Figure 5.11. *Init Function*

```
/* Output and update for atomic system: '<Root>/Blinking_LED' */
void Example_Blinking_LED(void)
{
  uint8_T rtb_DataTypeConversion;

  /* DataTypeConversion: '<S1>/Data Type Conversion' incorporates:
   *  Memory: '<S1>/Memory'
   */
  rtb_DataTypeConversion = Example_DW.Memory_PreviousInput;

  /* S-Function (Dio_WRITE_Channel): '<S1>/Dio_WRITE_Channel1' */
  Dio_WRITE_Channel(39U, rtb_DataTypeConversion);

  /* Update for Memory: '<S1>/Memory' incorporates:
   *  Logic: '<S1>/NOT'
   */
  Example_DW.Memory_PreviousInput = !Example_DW.Memory_PreviousInput;
}
```

Figure 5.12. *Step Function*

- ***Example.h*** and ***Example_private.h*** contain the declaration of all the generated functions while ***rtwtypes.h*** and ***Example_types.h*** contain the definition of the basic types. It's not important to analyze this file because are useful only to avoid compilation error after the integration in the Project.

  Read the last section 5.4 in order to understand how to integrate the generated files in the proect.

5.3 Second Example: 3-phase PWM generation

Generate a center-aligned 3-phase PWM with a period of 100 us and dead time of 30 ns. The value of the duty cycle of the 3 main channels, change according to the value returned by one of the ADCs of the target board: whenever the value is greater than a threshold of 2702 (i.e. 3.3 V), a 3-phase PWM with duty cylce of 30% is generated otherwise with a duty cycle of 70%.

1. Start Simulink.
2. *C*lick on *New* pane → *Blank Model* and a new window appears.
3. Open the Simulink Library Browser and found *Subsystem* block; name

it with the decided task name (*e.g. PWM_Gen*) and delete everything inside.

Figure 5.13. *Subsystem Block*

4. Right click on *Subsystem* block → *Block Parameters (Subsystem)* and then select *Main* pane in order to modify the Subsystem parameters: select *Treat as atomic unit* and set the desired *Sample time* in seconds: in this case 0 must be written because the task has to be activated by Timer12 Period Match (section 2.2.2)

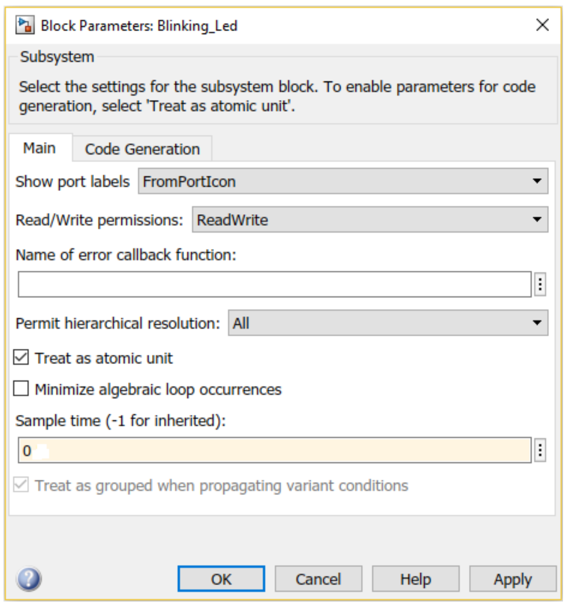


Figure 5.14. *Subsystem Block Parameters: Main pane*

Now select the *Code Generation* pane and set the *Function packaging* to *Nonreusable function* and *Function name options* to *Use Subsystem name*(figure 5.15)

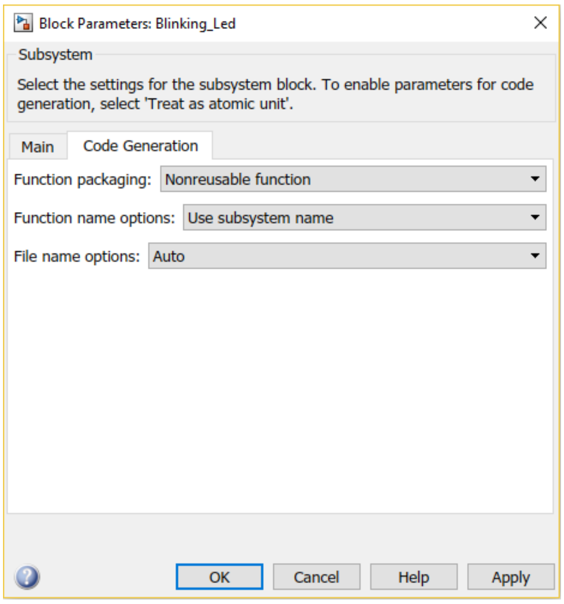


Figure 5.15. *Subsystem Block Parameters: Code generation pane*

5. Drag inside the *Subsystem* block the *Initialize Function* from the Simulink Library Browser and delete everything inside it except for the *Event Listener*. This block is necessary because it's used to "contains" all the initialize block function

Figure 5.16. Initialize Function block

6. Open the Simulink Library Browser and find the *Aurix Arduino-like BSP* library. Drag six *PORT_00-34_Conf* blocks in the *Initialize Function* block and configure the *PORT02_PIN00*, *PORT34_PIN03*, *PORT34_PIN04*, *PORT02_PIN03*, *PORT33_PIN14*, *PORT33_PIN15* to work as *PORT_OUT_PUSHPULL_ALT_7*. In this way pin P02.00, P34.04 and P33.14 are configured to be the CC60, CC61, CC62 outputs respectively (the 3-phase PWM main channel) while P34.03, P02.03, P33.15 are configured to be COUT60, COUT61, COUT62 outputs respectively (the 3-phase PWM complementary channel). Look at the figure 5.17 to

understand how to configure the other parameters.

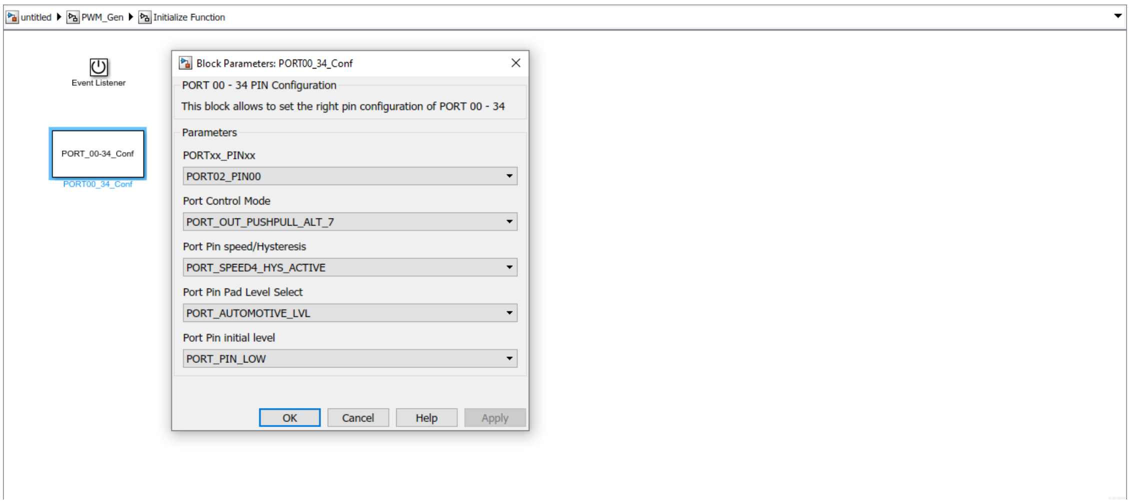


Figure 5.17. *Initialize Function: Port configuration*

Figure 5.17 shows how to configure the *PORT02_PIN00* and the configuration process must be repeated for the other five port pin.

Now drag *Adc_StartBackgroundConvertion* block and select *TRUE* in order to enable the Adc background convertion (figure 5.18).

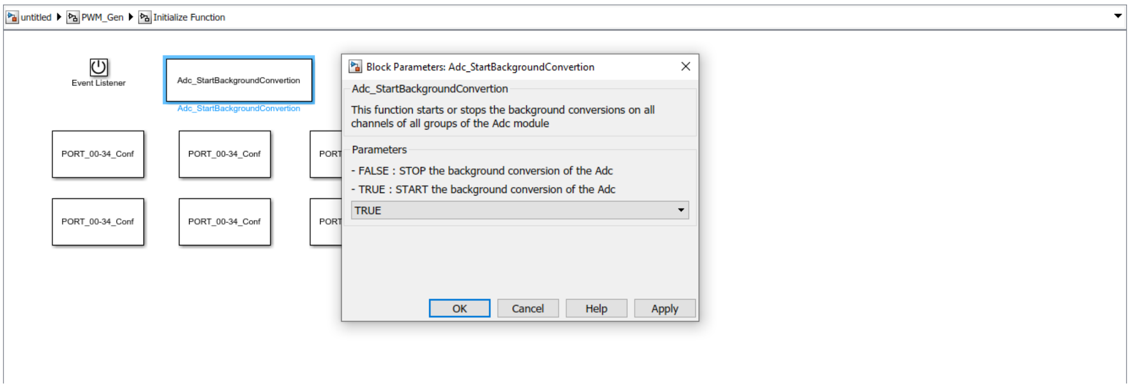


Figure 5.18. *Initialize Function: Start Adc Background Convertion*

In order to configure the CCU6 Operating Mode, PWM period and dead time value, the *CCU6_PWM_Setup* must be instantiated (figure 5.19)

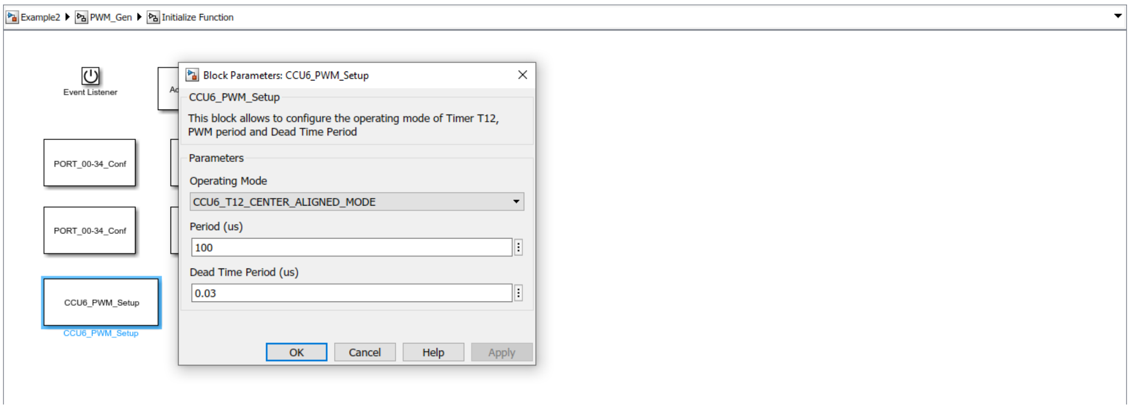


Figure 5.19. *Initialize Function: CCU6 configuration*

7. Inside the *Subsystem* block the *Adc_Read* block is instantiated and configured to reaturn the value converted by the desired Adc group channel, in this case the *ADC_IF_GR2_CH0*. This block is connected with the *Switch* Simulink block with a threshold set to 2072: the inputs of this block are two Constant *double* value 30 and 70. The output of the switch is used to set the Duty Cycle of the 3-phase PWM by using the *CCU6_PWM_SetDutyCycle* block.

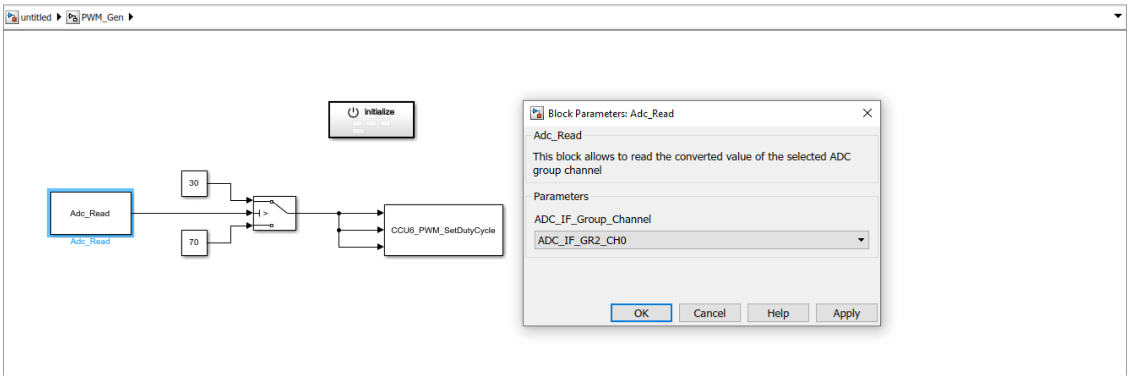


Figure 5.20. *Model Design*

8. Once the model is created, click on  (*i.e. Model Configuration Parameters*) to configure the Solver Type and set the System Target File:

   – *Solver*: Select *Fixed-step* in Solver selection;

– *Hardware Implementation*: Select *Infineon* in Device vendor and *TriCore* in Device type; it's useful only to define the correct data type size;
– *Code Generation*: Set System Target File as *mbd_erika_rtos.tlc*, select *Generate code only* and deselect *Generate makefile* (figure 5.21).

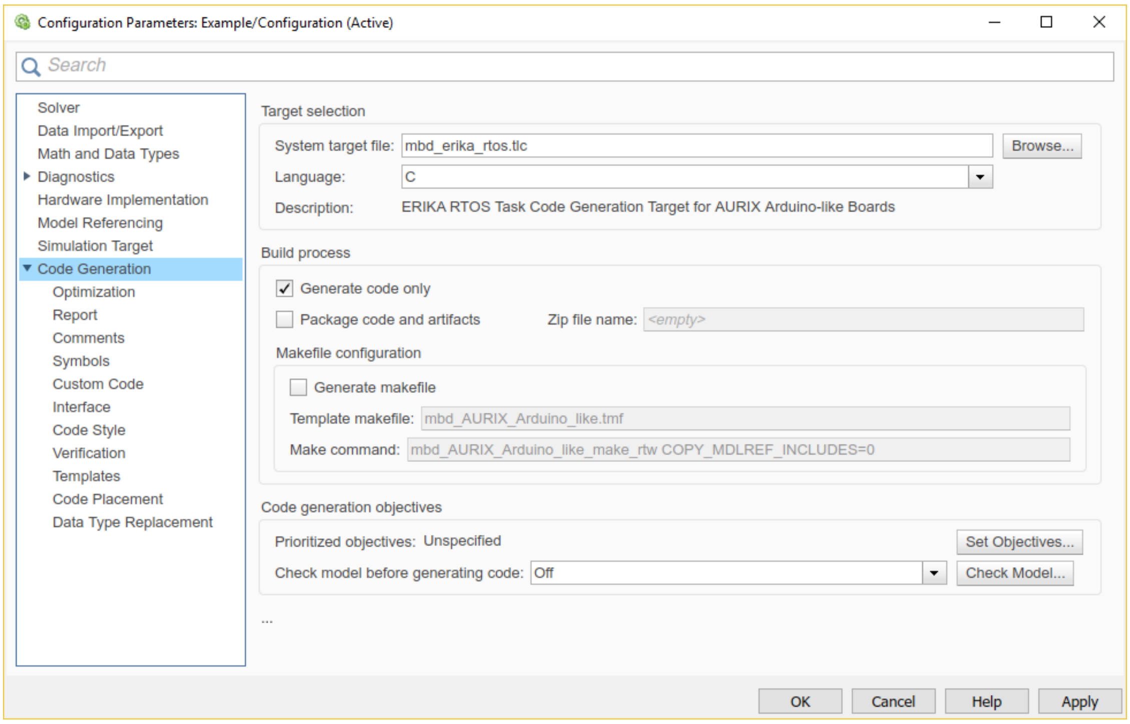


Figure 5.21. *Set parameters for the Code Generation*

9. *erika_rtos* folder must be added to MATLAB search path. Run the `addpath("path");` instruction on the MATLAB command line where the arguments "path" represents the position of *erika_rtos* folder: for example ***addpath('C:\ProgramFiles\MATLAB\R2018b\rtw\c\erika_rtos');***;

10. Save and name the model (*e.g.* *Example2*) and then click on  to build the model.

If everything is done correctly, the *Code Generation Report* appears. The auto-generated files will be placed in the *"modelname"_task* folder (in this case *Example2_task* folder) inside MATLAB working directory. Have a look to the auto-generated files:

- **oscfg.oil.c** contains the ERIKA RTOS configuration. *PWM_Gen* task is here configured:

```
TASK PWM_Gen {
  CPU_ID = "cpu0";
  PRIORITY = 1;
  AUTOSTART = TRUE;
  STACK = PRIVATE {
    SYS_SIZE = 512;
  };

  ACTIVATION = 1;
  SCHEDULE = FULL;

  /* Event managed by the Task */
  EVENT = ScheduleEvent_Period_Match;
};

EVENT ScheduleEvent_Period_Match {
  MASK = AUTO;
};
```

Figure 5.22. *Oil file: Task configuration*

- **OSTasks0_ErikaOs.c** contains the extented task implementation (figure 5.23). Task is waiting for the "arrive" of the event configured in the .oil file, so in this case it is the *ScheduleEvent_Period_Match* event generated by the CCU6 Timer12 (section 2.2.2): the step-function (*i.e. Example2_PWM_Gen()*) is called every period match, so in this case every *(PWM period/ 2 - 1) us* because the selected CCU6 Operating Mode is Center Aligned (section 2.2.2).

```
/* Task definition: PWM_Gen */
unsigned int volatile PWM_Gen_count;
TASK(PWM_Gen)
{
  EventMaskType mask;

  /* Task Body */
  while (1) {
    WaitEvent(ScheduleEvent_Period_Match);
    GetEvent(PWM_Gen, &mask);
    if (mask & ScheduleEvent_Period_Match) {
      ClearEvent(ScheduleEvent_Period_Match);

      /* call MBSD auto-generated code step function */
#ifdef __BSWL_COMPILE__

      Example2_PWM_Gen();

#endif

      /* Increment execution Counter */
      PWM_Gen_count++;
    }
  }
}

;
```

Figure 5.23. *Extended task implementaion*

- The main function in ***Example.c*** are the initialize function (that "calls" all the function blocks inside the *Initialize Function*) *Example2_PWM_Gen_Init()* (figure 5.24) and the *Example2_PWM_Gen()* (figure 5.25) corresponding to the step function.

```c
/* System initialize for atomic system: '<Root>/PWM_Gen' */
void Example2_PWM_Gen_Init(void)
{
  /* SystemInitialize for Atomic SubSystem: '<S1>/Initialize_Function' */

  /* S-Function (Adc_StartBackgroundConvertion): '<S2>/Adc_StartBackgroundConvertion' */
  Adc_StartBackgroundConvertion(((uint8_T)2U));

  /* S-Function (CCU6_PWM_Setup): '<S2>/CCU6_PWM_Setup' */
  CCU6_PWM_Setup(((uint8_T)2U), 100.0, 0.03);

  /* S-Function (PORT00_34_Conf): '<S2>/PORT00_34_Conf' */
  PORT00_34_Conf(19U, ((uint8_T)11U), ((uint8_T)4U), ((uint8_T)1U), ((uint8_T)1U));

  /* S-Function (PORT00_34_Conf): '<S2>/PORT00_34_Conf1' */
  PORT00_34_Conf(148U, ((uint8_T)11U), ((uint8_T)4U), ((uint8_T)1U), ((uint8_T)
    1U));

  /* S-Function (PORT00_34_Conf): '<S2>/PORT00_34_Conf2' */
  PORT00_34_Conf(149U, ((uint8_T)11U), ((uint8_T)4U), ((uint8_T)1U), ((uint8_T)
    1U));

  /* S-Function (PORT00_34_Conf): '<S2>/PORT00_34_Conf3' */
  PORT00_34_Conf(22U, ((uint8_T)11U), ((uint8_T)4U), ((uint8_T)1U), ((uint8_T)1U));

  /* S-Function (PORT00_34_Conf): '<S2>/PORT00_34_Conf4' */
  PORT00_34_Conf(145U, ((uint8_T)11U), ((uint8_T)4U), ((uint8_T)1U), ((uint8_T)
    1U));

  /* S-Function (PORT00_34_Conf): '<S2>/PORT00_34_Conf5' */
  PORT00_34_Conf(144U, ((uint8_T)11U), ((uint8_T)4U), ((uint8_T)1U), ((uint8_T)
    1U));

  /* End of SystemInitialize for SubSystem: '<S1>/Initialize_Function' */
}
```

Figure 5.24. *Init Function*

```c
/* Output and update for atomic system: '<Root>/PWM_Gen' */
void Example2_PWM_Gen(void)
{
  real_T rtb_Switch;

  /* Switch: '<S1>/Switch' incorporates:
   *  Constant: '<S1>/Constant'
   *  Constant: '<S1>/Constant1'
   *  S-Function (Adc_Read): '<S1>/Adc_Read'
   */
  if (((uint16_T)Adc_Read(((uint8_T)1U))) > 2702) {
    rtb_Switch = 30.0;
  } else {
    rtb_Switch = 70.0;
  }

  /* End of Switch: '<S1>/Switch' */

  /* S-Function (CCU6_PWM_SetDutyCycle): '<S1>/CCU6_PWM_SetDutyCycle' */
  CCU6_PWM_SetDutyCycle(rtb_Switch, rtb_Switch, rtb_Switch);
}
```

Figure 5.25. *Step funtion*

- ***Example2.h*** and ***Example2_private.h*** contain the declaration of all the generated functions while ***rtwtypes.h*** and ***Example2_types.h*** contain the definition of the basic types. It's not important to analyze this file because are useful only to avoid compilation error after the integration in the project.

Another important thing to be done before the integration of the generated files in the project, is the definition of the *SetEvent* function inside the CCU6 Interrupt handler. This function is used to generate the *ScheduleEvent_Period_Match* in order to activate the generated task each CCU6 Period Match.

Open the *CCU6.c* file inside `TricoreBswl\BSWL\Kernel\MCAL_TC277TF\CCU6` folder and uncomment the *SetEvent* function at code line *241*. Furthermore the name of the generated task must be written inside the function. Look at the figure 5.26 to better understand.

```
IFX_INTERRUPT( CCU6_SR0_Handler,0, CFG_IRQ_CCU6_0_PRIO_SR_0 )
{

    if( pxCCU6->IS.B.T12PM == 0x1 )
    {
        SetEvent(PWM_Gen,ScheduleEvent_Period_Match);   /* Set the Event in order to run the task */
        pxCCU6->ISR.B.RT12PM = 0x1;                        /* Reset Timer T12 Period-Match Flag  */
    }
    if( pxCCU6->IS.B.T12OM == 0x1 )
    {
        pxCCU6->ISR.B.RT12OM = 0x1;                        /* Reset Timer T12 Period-Match Flag  */
        pxCCU6->TCTR4.B.T12STR = 0x1;                      /* Enable Shadow transfer to T12PR */

    }
}
```

Figure 5.26. *CCU6.c file*

Read the last section 5.4 in order to understand how to integrate the generated files in the proect.

5.4 Integrate the Generated File in the Project

The auto-generated files are easily integrable in the basic software but some manual interventions are necessary to make everything works. First of all the Project must be "aware" that the auto-generated files will be integrated in the basic software: a **MATLAB** variable **must** be defined in *bswl_presence.h* file. This is a necessary step to adjust the values passed by Matlab to BSP functions: mask popup type (refers to previous chapter) treats every choice as incremental integer number starting from 1.

The *OIL* file, the *OS_Tasks0_ErikaOs* file, the model C file and all the header files have to be copied and pasted in the right Basic Software folder:

- *oscfg.oil.c* must be renamed in *oscfg.oil* and copied inside the `TricoreBswl\PrjCfg\ErikaOsCfg\OilFile` folder
- *OS_Tasks0_ErikaOs* inside `TricoreBswl\BSWL\OS\OsTask_ErikaOs\cpu0` folder
- all the other files in `TricoreBswl\ASWL` except for *ert_main.c* that is never used.

The model initialize function and the header file that contains its declaration must be manually added in the *cpu0_main.c* file: the correct code location is indicated by the comments in the file. Look at the following code (figure 5.27 refers to *First Example* (section 5.2) to better understand where the header file *Example.h* (*i.e.* the header file containing the delcaration of all the generated functions) and the initialize function *Example_initialize()* has to be added.

```
#ifdef __BSWL_COMPILE__
#include "bswl_if.h"
#include "Imcal_if.h"

/*Add here the .h file auto-generated by MATLAB*/
#include "Example.h"

#endif

#define BSW_START_SEC_CODE
#include "MemMap.h"

int main( void )
{
    StatusType status;
    #ifdef __BSWL_COMPILE__

    Imcal_Init();

    /* Add here the initialize functione auto-generated by MATLAB */
    Example_initialize();

    #endif
    StartCore(OS_CORE_ID_1, &status);
    StartCore(OS_CORE_ID_2, &status);

    StartOS(DONOTCARE);

    return 0;
}

#define BSW_STOP_SEC_CODE
#include "MemMap.h"
```

Figure 5.27. *cpu0_main.c file*

Now it's time to generate configuration files for ERIKA RTOS and to compile the operating system (section 3.2.3). Then it's the turn to import the Project in Eclipse (section 3.2.2) in order to build it (section 3.2.4) and program and debug the target board (section 3.2.5).

Bibliography

[1] Broy M., Kirstan S., Kremar H., Schätz B., Zimmermann J., *What is the benefit of a model-based design of embedded software systems in the car industry?*, Germany

[2] The Mathworks Inc, *How Small Engineering Teams Adopt Model-Based Design*, The Mathworks

[3] Kautz O., Roth A., Rumpe B.,*Achievements, Failures, and the Future of Model-Based Software Engineering*

[4] The Mathworks Inc, *Simulink® Developing S-Functions*, The Mathworks

[5] The Mathworks Inc, *Target Language Compiler™ For Use with Real-Time Workshop®*, The Mathworks

[6] The Mathworks Inc, *Real-Time Workshop® For Use with Real-Time Workshop Embedded Coder*, The Mathworks

[7] The Mathworks Inc, *Embedded Coder® User's Guide*, The Mathworks

[8] Infineon Technologies AG, *TC27x C-Step*, Infineon Technologies

[9] Violante M., *Operating Systems for Embedded Systems - Course Slides*, a.y. 2016/2017

[10] Evidence S.r.l., *ERIKA Enterprise Manual Real-time made easy*, version: 1.4.4, 2012

[11] OSEK, *OSEK/VDX - Operating System*, version: 2.2.3, 2005

[12] OSEK, *OSEK/VDX - System Generation - OIL: OSEK Implementation Language*, version: 2.5, 2004

[13] Future Technology Devices International Limited, *User Guide for FTDI FT_PROG Utility*, 2016

[14] HighTec, *A Getting Started to Free TriCore ™ Entry Tool Chain*, HighTec

[15] Cottone F., *A Model-Based Design Embedded Software Development Methodology for an OSEK-Compliant RTOS - Master Thesis*, Politecnico di Torino, Luglio 2019