TOSHIBA Motor Control Firmware
User Manual
(TMPM4K)

Description

User manual for Toshiba Motor Control Firmware.

Table of Contents

List of Figures

List of Tables

Conventions used in this document

Numerical Values

Hexadecimal number: *0xABC or 0h12F*
Decimal number: *123* or *0d123* (explicitly indicating the decimal numbers)
Binary number: *0b111*

Signals

Active low signals are indicated by a *_N* at the end of the signal name. Example: *RESET_N*.

Assertion of a signal shall mean its activation (transition to the active state). *Deassertion* of a signal shall mean its deactivation (transition to the inactive state).

Bus signals are indicated by *[x:y]* at the end of signal name. Example: *DATA[3:0]* indicates a four bit bus with the individual bus signals *DATA[3], DATA[2], DATA[1] and DATA[0]*.

Registers

Register names are indicated by square brackets *[…]*. Example: *[ABCD]*.

Two or more of the same kind of registers, fields, and bit names are collectively referred to by using a numerical suffix *n*. Example: *[XYZ1], [XYZ2]* and *[XYZ3]* are collectively referred to as *[XYZn]*.

The bit width of a register is expressed as *[x:y]* where x is the number of the most significant bit and y is the number of the least significant bit. Example: *[XYZ][3:0]* indicates a four bit-wide register named XYZ.

The configuration value of a register is expressed by either a hexadecimal number or a binary number. Example: *[ABCD].EFG* = 0x01 (hexadecimal), *[XYZn].XY* = 0b1 (binary).

The following definitions apply for Bytes and Words:

Byte	8 bits
Half Word	16 bits
Word	32 bits
Double Word	64 bits

Unless specified otherwise, registers support only word access.

Register which are indicated to be reserved must not be rewritten. The read value from reserved registers must not be used.

Properties of each bit in a register are expressed as follows:

R	Read only
W	Write only
W1C	Write 1 Clear; the corresponding bit is cleared (=0) when "1" is written to this bit.
W1S	Write 1 Set; the corresponding bit is set (=1) when "1" is written to this bit.
R/W	Read and Write are possible.
R/W0C	Read/Write 0 Clear
R/W1C	Read/Write 1 Clear
R/W1S	Read/Write 1 Set
RS/WC	Read Set/Write Clear; set after read operation, cleared after write operation.

Reading from register bits having a default value of "—" will result in an unknown value.

In case of write accesses to registers containing both read/write (*R/W*) and read-only (*R*) bits, the read-only bits shall be written with their default value. If this default is "—", follow the instructions of each register.

Reserved bits of Write-only (*W*) register should be written with their default value. If this default is "—", follow the instructions of each register.

1. Introduction

The TOSHIBA 3-Phase Motor's Vector Control Solution has two main components:

- A highly scalable and fully configurable Motor Control Firmware designed for the TMPM4K series MCUs, featuring Field Oriented Control (FOC) of up to three motors (1 with hardware Vector Engine and 2 with software vector control).
- The "MCU Motor Studio" PC Tool for Microsoft Windows, utilized for parameter configuration, drive control and real-time logging of various motor parameters in a high-speed Digital Storage Oscilloscope fashion.

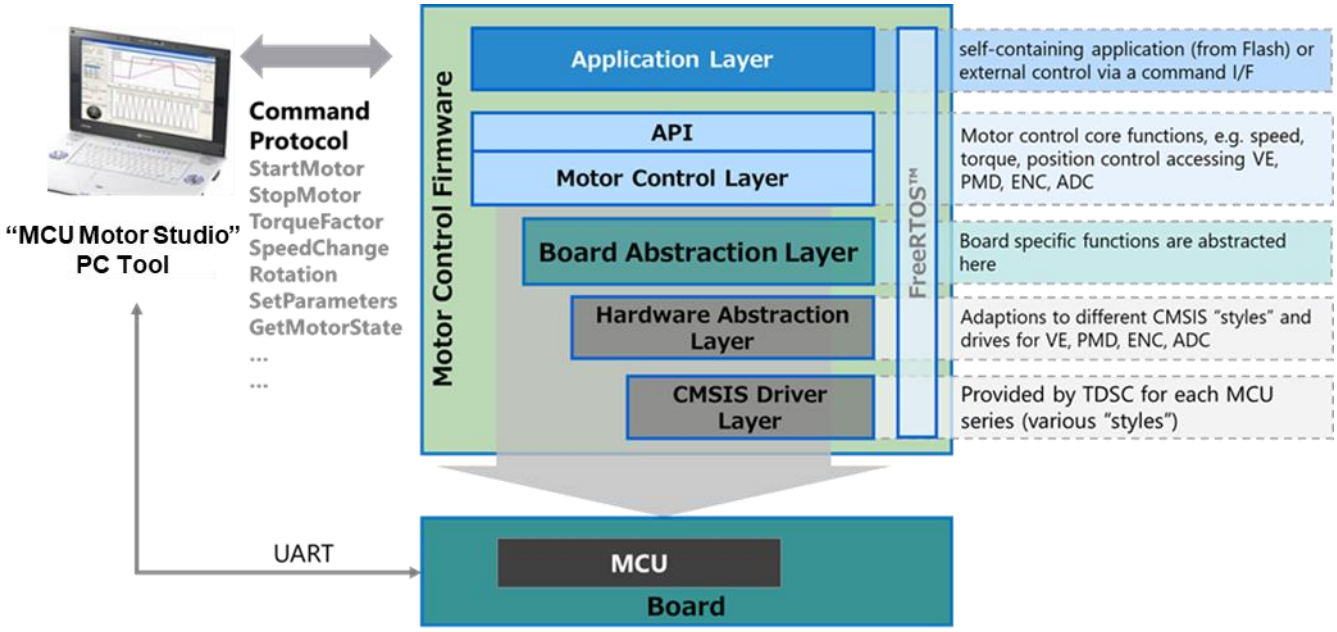

Figure 1.1 TOSHIBA 3-Phase Motor's Vector Control Solution

The main scope of this document is to describe the Motor Control Firmware, its features, components, configuration and usage.

All statements made in this guide are based on version 3.10 of the "MCU Motor Studio" firmware and version 4.1.0 of the "MCU Motor Studio" PC Tool. Minor differences in the configuration, the source code or the GUI look & feel are possible depending on the versions used. However, all design, implementation and operation principles remain unchanged. All examples are demonstrated using the default tool-chain "IAR Embedded Workbench" version 8.50.9.

The terms "Motor Control Firmware" and "Firmware" as used throughout this document are referring to one and the same software entity and are fully interchangeable.

Unlike most competitor's solutions, all TOSHIBA MCUs rely on a built-in hardware accelerator referred as the Vector Engine (VE), which is an enhanced mathematical co-processor highly optimized for the computation needs of a field oriented motor control.

Our Toshiba e-learning Center provides comprehensive technical description of the 3-phase motor Vector Control and the TOSHIBA Vector Engine under:

https://toshiba.semicon-storage.com/eu/semiconductor/knowledge/e-learning/village/vector-1.html

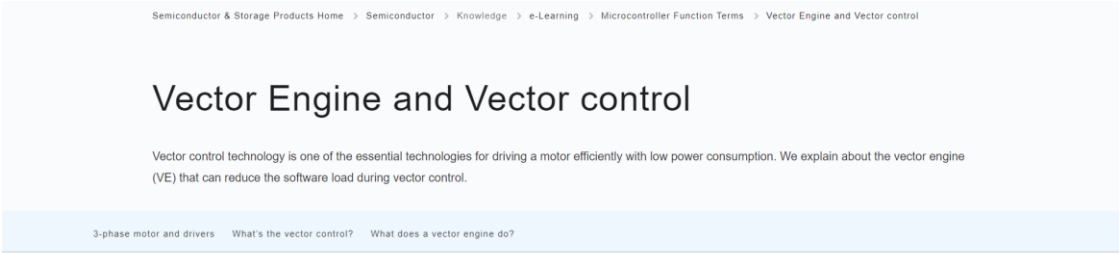

Figure 1.2 eLearning Vector Engine and Vector Control

The main advantage of this solution is the enormous off-loading of the CPU, since all needed mathematical calculations required for the position estimation, speed and torque control are performed with the support of hardware partially or entirely. For a typical use case, the CPU load can go as low as around 17-20% when driving single motor. The free CPU resources can be utilized for various other tasks, such as simple Man Machine Interface (MMI), communication with remote devices, real-time motor data logging for analysis and optimization of the motor regulation, etc.

The position estimation can be performed either with hardware support, given the motor is equipped with optical or magnetic (Hall principle) sensor that permanently monitors the shaft position and rotation. Alternatively, the Motor Control Firmware features angle detection and position estimation utilizing the vector engine's transformation logic or its equivalent software implementation.

The block diagram depicts the standard FOC operation using encoder (optical or magnetic):

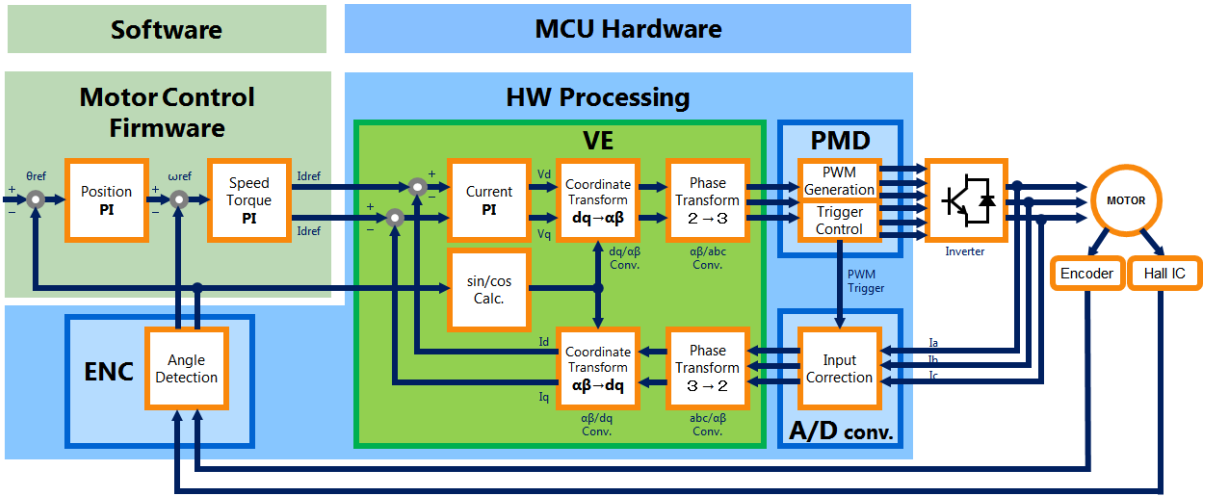

Figure 1.3 Field Oriented Control with Encoder Input Circuit (HW)

The block diagram depicts the FOC operation using Software/Vector Engine (VE) based position estimation and angle detection:

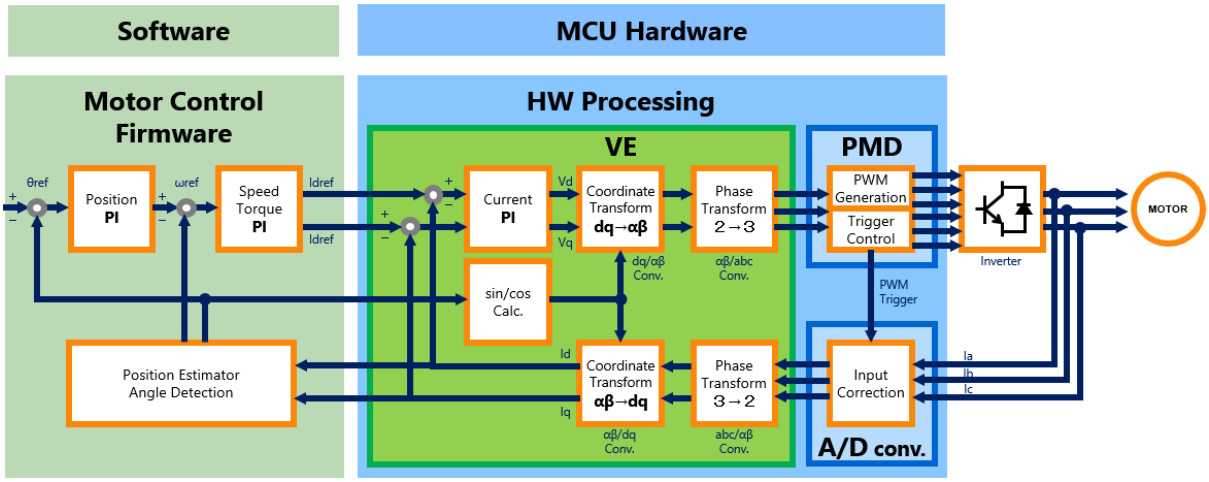

Figure 1.4 Field Oriented Control with Software Position Estimation (SW/VE)

Besides the computational acceleration, all MCUs feature hardware accelerators for the output control referred as Programmable Motor Drive (PMD) and ADC for the feedback path. These require configuration and minimal control in the Motor Control Firmware.

The Motor Control Firmware utilizes the Vector Engine (VE) and the Programmable Motor Drive (PMD) for additional convenient functions like the initial zero-point detection, motor positioning, field stall detection, load dependent speed reduction, controllable motor braking, etc.

A forced (sine-wave) commutation is used for the initial motor start-up and is hardware accelerated utilizing the Vector Engine's transformation and PI logic:

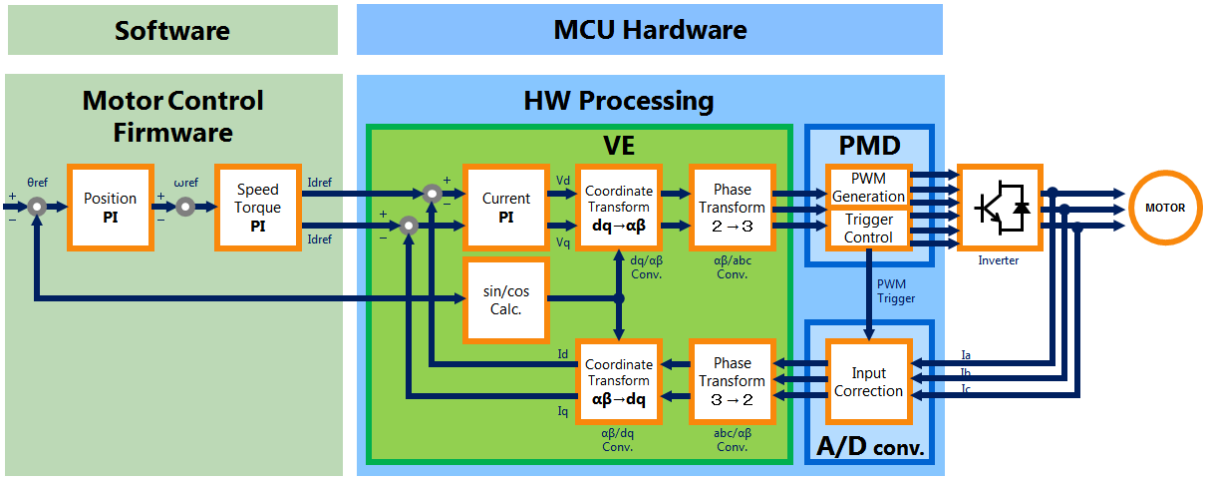

Figure 1.5 Forced Commutation (SW/VE)

The next chapters will detail the functionality, the architecture, project structure, configuration and usage of the Motor Control.

2. Main Features

The table below summarizes the main characteristics and all supported features of the Motor Control Firmware. Some of these are only available for particular family or family members. Such limitation is strictly enforced by the concrete hardware implementation of each MCU within the specific family.

Every single feature is fully implemented and tested for at least one family/family member and can be easily ported to any other:

Table 2.1 Motor Control Firmware Main Features List

"TOSHIBA Motor Control Firmware" Feature List	
Motor Drive Method	Field Oriented Control (Vector Control) using highly optimized mathematical co-processor, the "TOSHIBA Vector Engine"
Control Scheme	Speed / Torque / Position
Current Detection (configurable)	1-Shunt / 3-Shunt/ 2-Sensor(in future releases)
Number of motor channels	Up to 1 channels (V3.10, M4K) (Up to 3 channels in future releases)
PWM Frequency	Variable, 8kHz to 24kHz.
Protection	Over Current Field Stall Detection Over Voltage/Under Voltage Motor Disconnection Detection
Advanced & Convenience Functions	Zero Current Point Detection & Initial Motor Positioning Turn Control / Advanced Turn Control Linear Motion Control Stop Control Stall Recovery Load Dependent Speed Reduction
Encoder	Sensor based: Linear (Optical) / Magnetic (Hall) / Resolver Sensor-less: Advanced Software Positioning
Command I/F (via UART)	Motor, Board & System Parameter Configuration Parameters Read/Store from/into the NVM Motor Start/Stop Speed, Torque & Position Controls Simple Command Sequencer Demo Applications (Slider 2 demo) Controls Digital Storage Oscilloscope (DSO) High-Speed DSO (in future releases) Visualization and CSV Logging (MCU Motor Studio only)
Supported MCU Devices	M4K
Supported Tool-chains	IAR Embedded Workbench (default) / KEIL µVision (in future releases)

Not all of the features can be configured and used simultaneously on any of the family members. It is all dependent on the available memory and in some cases may require optimization of the task footprint.

The diagram below gives a brief overview of the control & computation blocks within a typical Field Oriented Control Unit:

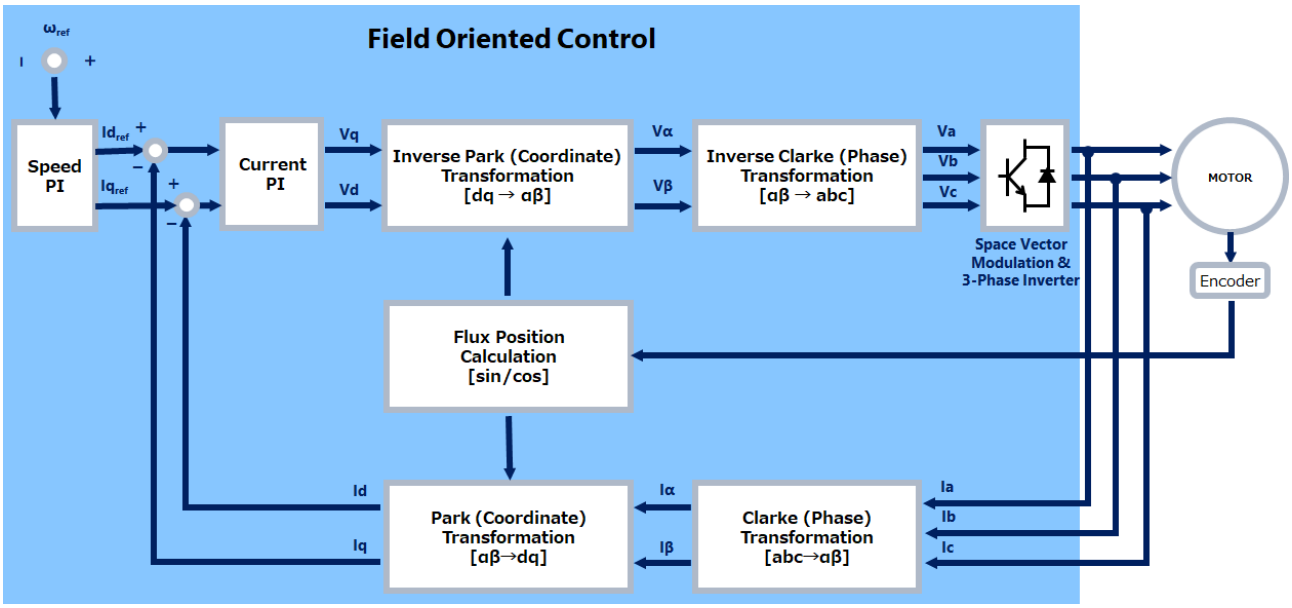

Figure 2.1 Field Oriented Control Block Diagram

The actual computation, control and driving activities split between Firmware (Software) and Hardware Accelerators Engines (VE, PMD and ADC) is not directly exposed to the application. Instead all control, protection and convenience functions are encapsulated in the Motor Control Firmware API and described in the next chapters and a dedicated API specification.

The Firmware is designed to offer high configurability and ease of use, thus only two steps are need for driving the motor. In the first one the user needs to configure various parameters, including:

- Motor characteristics like number of poles, torque constant, motor inductance, rated current, rated torque, etc.
- PI control parameters, like integral and differential coefficients of the flux and torque currents, etc.
- System settings like the PWM frequency and breaking type.
- Board specific settings like MOSFET dead time, bootstrap delays, measurement sensitivity, etc.
- Convenience & protection functionality like stall detection, overcurrent, load dependent speed reduction, etc.
- Advanced functionality – like linear motion control, advanced software positioning, load statistics, etc.
- Target speed or torque for the motor control.

Some configurations shall be done at compile time; others may be done either at compile time or prior/during (motor stop may be needed) the normal operation. Please refer Configuration section. All parameters can be stored in the NVM and will be automatically applied upon start-up if such are found. The configuration can be made static (NVM or compile-time) or dynamic (via the command interfaces or a stand-alone application that runs on top of the Firmware on the used MCU).

Once the configuration is completed, the user only needs to start the motor and set the desired speed, torque or position. The Firmware then performs all the necessary activities to go through all five stages needed to achieve steady rotation, e.g. Stop (if restart is needed only), Initial Positioning, Forced commutation, Change to Steady, Steady Rotation.

2.1. Supported Motor Types

The Motor Control Firmware can be utilized to fully speed control **Brushless DC (BLDC)** or **Permanent-Magnet Synchronous (PMSM)** motors.

Switched Reluctance Motors (SRM) and **Asynchronous AC** can be driven (force commutation) with basic speed control, field oriented controlling is not possible.

Brushed DC motors and **Stepper motors** require different commutation and control techniques/circuits and therefore are not supported.

2.2. Control Functions

The Motor Control Firmware supports three different control modes:

Control by speed (default) – set the target rotational speed in RPM or Hz. The firmware will make the motor revolve permanently with the requested speed.

Control by torque – set the target torque in mNm and desired rotational speed in RPM or Hz. The firmware will ensure the torque on the rotor is permanent and as requested. It will revolve the motor with the desired speed or lower it, given the requested torque cannot be achieved/maintained.

Control by position (optional, needs to be compiled-in) – set the target position of the rotor in either number of physical degrees (full turns inclusive) or in number of sensor ticks, and the desired speed. The firmware will revolve the rotor to make the needed full or partial turns with the desired speed, or the maximal possible one, to achieve the target position. The motor will be stopped and the rotor will be kept at that position.

A Field Oriented Control, as long as possible, is used in either of the control modes. Speed and torque control are always available. The position mode shall be explicitly configured and enabled. There are two variants – the sensor based “linear motion” and the sensor-less “advanced turn control” that utilizes the software position estimator. Each variant can be enabled/compiled-in separately. Both can co-exist and can be freely used by the application, but not simultaneously. Further details are provided in the upcoming chapters.

There are two groups of APIs that are used for controlling the speed of the motor:

Configuration – These are used to set-up the parameters of the motor and the control logic in one go or individually. Parameter changes on the fly are permitted, although this will be internally applied at strictly specified points of time/execution. Additional APIs for status and/or current configuration (single or “in one go”) retrieval are provided. Storage and clearance of the NVM sector(s) containing static parameters are also available.

Motor Operation – Motor start and motor stop can be directly initiated. All other details of positioning the motor, force commutation and entering the FOC stage are taken care internally by the firmware considering the configuration parameters set.

2.3. Protection Functions

The following protection schemes are built-in and configurable at compile-time:

Overcurrent Protection – Hardware realized permanent current monitoring with emergency switch off. The PMD engine stops all control of the power output stage upon emergency. It is enabled by default and can be configured at compile-time in the Power stage configuration’s header file. There is additional software controlled monitoring that is disabled by default and shall only be used if the overcurrent protection is not available on the power stage board (an input signal that has to be fed to the PMDs).

Over and Under Voltage Protection – Hardware supported monitoring of the pre-configurable voltage window. Upon detection the Firmware will trigger a stop via the PMD and the power output stage. This feature is disabled by default and can be configured at compile-time in the user configuration header file.

Motor Disconnection Protection - Hardware supported monitoring of the Iq control current. Upon detection the Firmware will enter the protection handler and abort the FOC control. This feature is disabled by default and can be configured at compile-time in the user configuration header file.

Stall Protection – Hardware supported Field lost detection with configurable reaction. The user can select the Stall recovery policy as described in the convenience function. The feature is disabled by

default and can be configured at compile-time in the user configuration header file.

2.4. Advanced & Convenience Functions

The following advanced and convenience functionality is built-in in the Motor Control Firmware. Some of the function cannot be disabled as these are needed for the proper operation of the speed control:

Zero Current Point Detection – Measurement of the input currents to determine the “0” measurement value of the ADC conversions, e.g. to calibrate the measurement. Performed automatically by the Firmware. Not configurable.

Initial Motor Positioning – DC energization, e.g. applying current to the motor coil, resulting magnetic flux lines that will position the rotor at approximately 90/-90 degrees. Performed automatically by the Firmware. Not configurable.

Turn Control – User can define number of mechanical turns at desired speed and the ramp-up time for achieving it. The Motor Control Firmware will automatically accelerate, perform the number of turns and deaccelerate until full stop. Disabled by default.

Advanced Turn Control – User can define number of full mechanical turns, an additional (and optional) angle in tenth of degrees and a desired speed. The Motor Control Firmware will automatically calculate the ramp, accelerate, perform the number of turns and deaccelerate until full stop reaching the desired final position. The function does not necessarily require an external rotor position sensor. In sensor-less mode the Firmware utilizes the open-loop Advanced Software Positioning computation algorithm for dynamic position estimation. Disabled by default.

Linear Motion Control – only in combination with the built-in encoder and external optical or magnetic sensor for shaft position reading. User can define number of pulses (dependent on the sensor resolution) and desired speed. The Motor Control Firmware will automatically calculate the ramp, accelerate, rotate and deaccelerate until full stop, reaching the desired number of absolute position in number of pulses. Disabled by default.

Stop Control – Three options of the motor stop control are available and can be configured at start-up or on the fly – gentle, short and self-break. Gentle break is fully controlled reduction of the speed with maximum angular de-acceleration. In short breaking mode the Firmware alternates short the upper/lower phases of the motor. Self-break is mode where no control is applied and the motor is drilling down on its own.

Stall Recovery – Permanent monitoring of the Vdi for early recognition and recovery. The user can select between manual (stop) and automatic (re-start) recovery upon Field Stall detection.

Load Dependent Speed Reduction – Automatic reduction of the speed, maintaining the current output upon extra load detection. This feature is disabled by default and can be configured at compile-time in the user configuration header file.

2.5. Channel Control

All TOSHIBA Motor Control MCUs feature at least one Vector Engine (VE) or Advanced Vector Engine (A-VE), allowing effective control of one single motor. The newly introduced M4KN has a single Advanced Vector Engine, but due to its powerful Cortex-M4 can control additional two channels, performing the VE calculation tasks mostly in software, jointly referred as software vector engine(s).

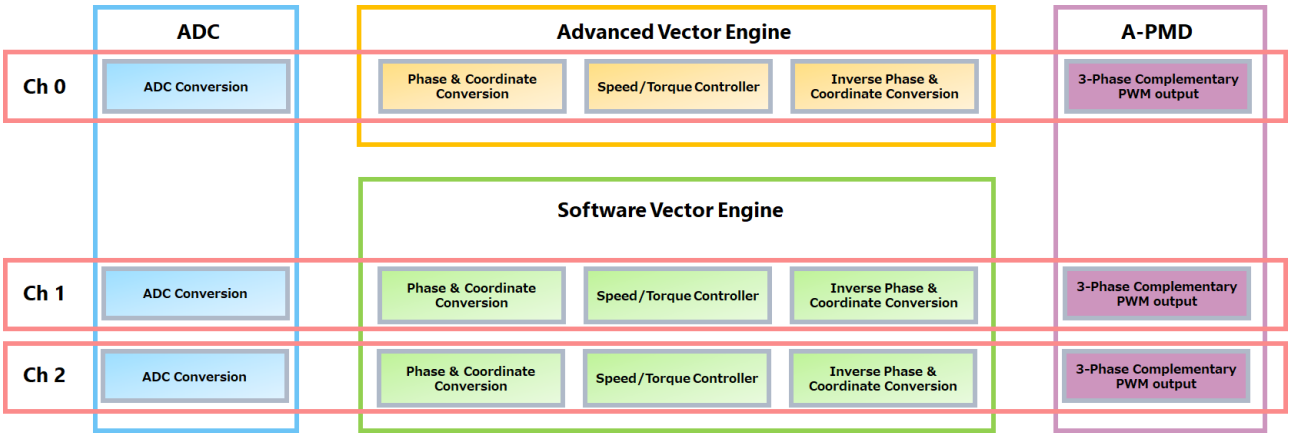

Figure 2.2 Three Channel Field Oriented Control Resource Usage

This chart is solely used to illustrate the calculation (hardware and software resources) split for each individual channel. The software controlled computational operation within Channel 1 & 2 are serialized as these have to be executed on single CPU, running in different RTOS tasks though. Even the A-VE supported control of channel 0 requires certain CPU processing time, thus the very simplified timing chart will look like:

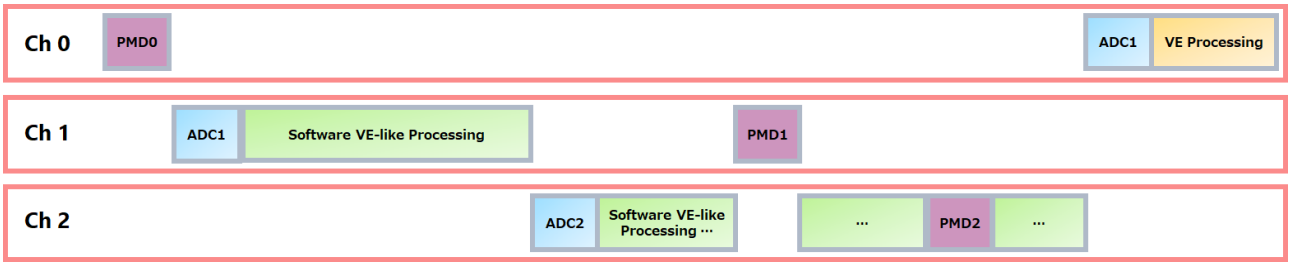

Figure 2.3 Three Channel Field Oriented Control Timing Chart

The channel assignment in the Motor Control Firmware is static and can't be changed. The lowest channel number is always controlled using hardware acceleration, whereas the others are software driven.

Further details on the processing, computation tasks split and timing will be given in Chapter 4.

3. Firmware Architecture

3.1. Layer Structure

Having scalability, extendibility and portability in mind a software design with multilayered structure was implemented. Every layer has its own responsibilities and provides services to the others, typically higher levels, via well-defined interfaces.

The layer independency introduced with this design pattern allows changes or extensions in one layer without direct impact to the operation of all others.

The encapsulation of hardware, software and functionality introduces another level of modularity and improves the testability.

The clear interfaces ease and unify the interaction between subsystems, allowing simple exchange of modules with different versions or implementations of the same.

A top-level view of the firmware architecture is presented below:

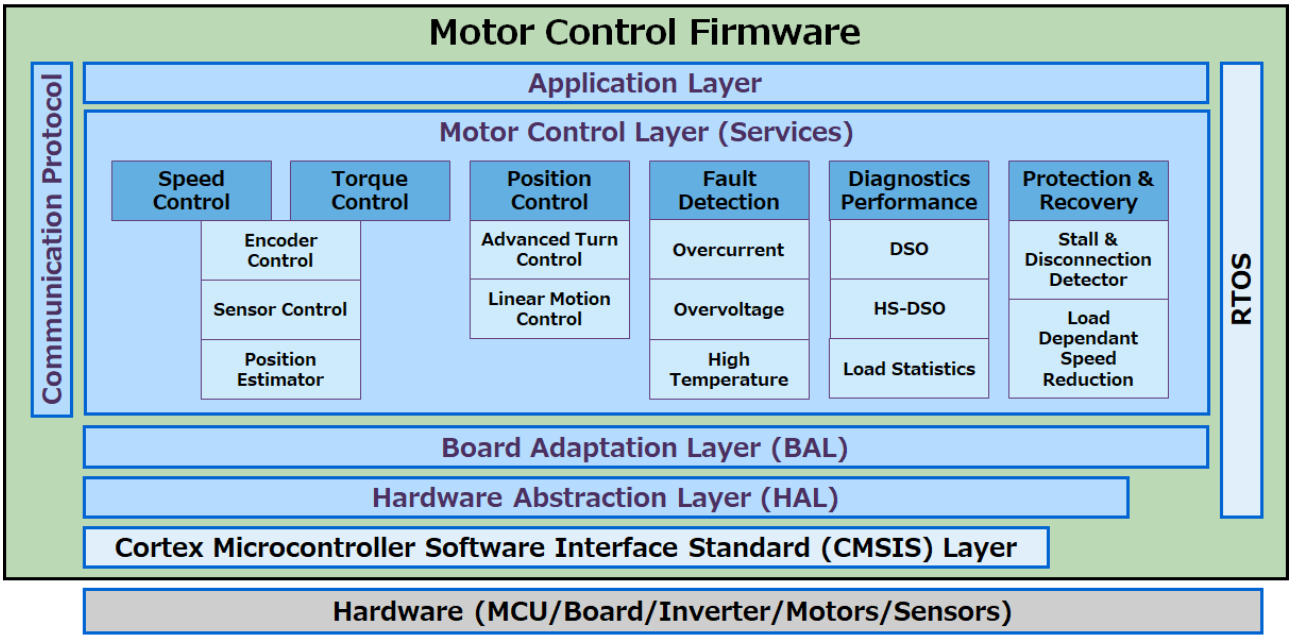

Figure 3.1 Motor Control Firmware Architecture

Application Layer – hosts the particular top-level user control implementation. It may be either a self-containing standalone application (several demos provided) or application allowing external control via communication interfaces such as UART, CAN (in future releases), etc. The “External Motor Control” feature allows external inputs like voltage of PWM duty cycle to be used for setting and achieving the desired speed and direction. A stand-alone application may also, if desired so, accept external control or re-configuration requests/commands. For remote system configuration (e.g. with the MCU Motor Studio tool) the command interface allows not only the change of motor parameter(s), but also system parameter(s) (e.g. PWM frequency) or even board parameter(s) (e.g. MOFSET dead time). Most of the system or board parameters can be changed on the fly without the need of recompiling the Firmware. More details are provided in the “Configuration chapter”

Motor Control Layer – the core functionality, including the tasks and interrupt handlers for speed, torque and position control accessing the MCU’s hardware modules (VE, PMD, ENC, ADC) as well as all needed helper and utilities functions – DSO/HS-DSO, performance measurements, stall detection, user callbacks, etc.

Board Adaptation Layer – has the specific implementations for configuration and operation of any

on-board components like LEDs, switches, EEPROMs, external oscillators, SPI devices, port multiplexers, temperature sensors, etc. Board specific customizations and further configuration, like gain control, measurements sensitivity, type of measurement, etc. are also to done here.

Hardware Abstraction Layer – adapts to different versions (and implementations) of the CMSIS peripheral drivers available for the various MCUs, unifying their access methods and usage.

CMSIS Layer - includes the Cortex Microcontroller Software Interface Standard (CMSIS) compliant peripheral drivers (Device Peripheral Access Layer) dedicated for the particular MCU, adapting the differences in the type of peripherals, number of channels, etc.

FreeRTOS™ – the layer provides means to manage activities and resources in real time. It is a standard Cortex-M3 port of the scalable real time kernel, configured for the requirements and specific usage of the Motor Control Firmware. The current version utilizes only preemptive tasks and mutexes.

3.2. Folder Structure

The folder structure is kept identical, as far as it is possible, with the layered architecture of the Motor Control Firmware, allowing easy navigation and maintenance.

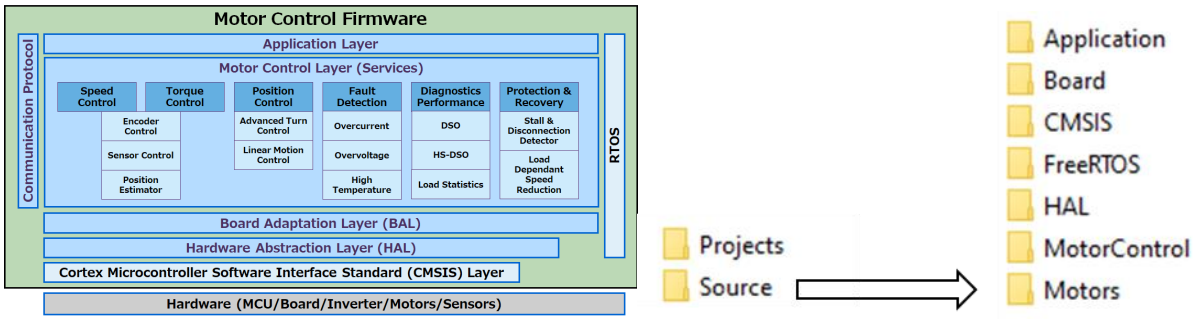

Figure 3.2 Motor Control Firmware Folder Structure

Project – The folder holds all workspace and project related files, including configuration files, resources, and specific implementation (C & H files) for supported tool-chain – IAR Embedded Workbench. There are mostly two configurations for every supported MCU device – Release and Debug (always available). As the names suggest the difference is in the level of debug information available. In some cases, Release may feature different optimization level. The figure below does not list all MCUs and configuration, as these are created at first build/usage and is meant for illustrative purposes:

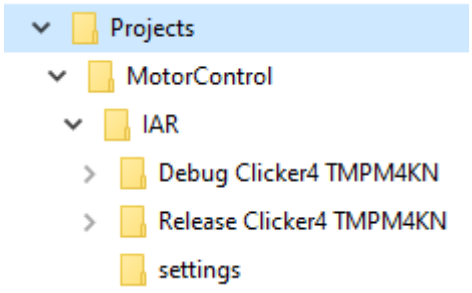

Figure 3.3 Motor Control Firmware Projects Folder Structure

Sources – Implementation of all layers with default configuration for any of the currently supported boards, power output stages, sensors and motors. The individual sub-folders are named upon the layer they implement. The only exception is the extra “Motors” folder, which as the name suggest, contain description of several BLDC motors as C header files. These capture characteristics such as number of poles, speed and current limits, resistance, inductance, encoder type if any, etc.

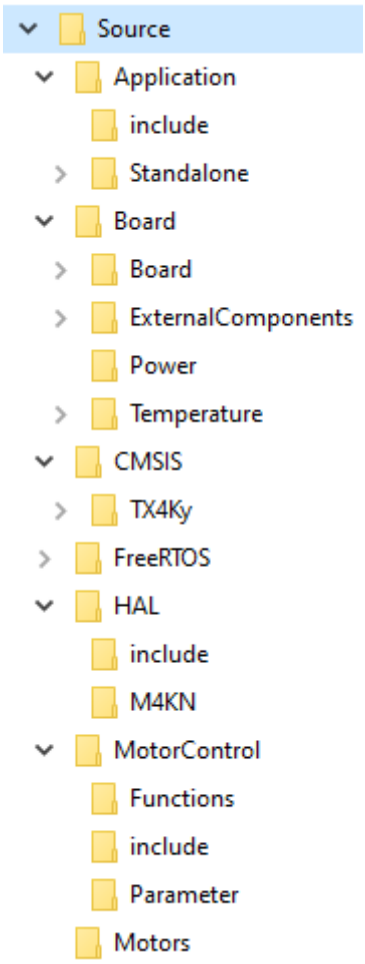

Figure 3.4 Motor Control Firmware Sources Folder Structure

3.3. Applying FreeRTOS Patch

"MCU Motor Studio" firmware uses FreeRTOS Kernel V10.2.1. The project folder structure will not contain the FreeRTOS open source code, user has to download the FreeRTOS open source code and apply patch available in release package.

Please follow the below mentioned procedure to create the folder "FreeRTOS \ source"

1. Download the FreeRTOS from the path " https://sourceforge.net/projects/freertos/files/FreeRTOS/V10.2.1/FreeRTOSv10.2.1.zip/download" and unzip the downloaded file.

   Note:
   a. If required, please accept website cookies to download FreeRTOS kernel.
   b. Newer or different version of FreeRTOS can be used at own risk.

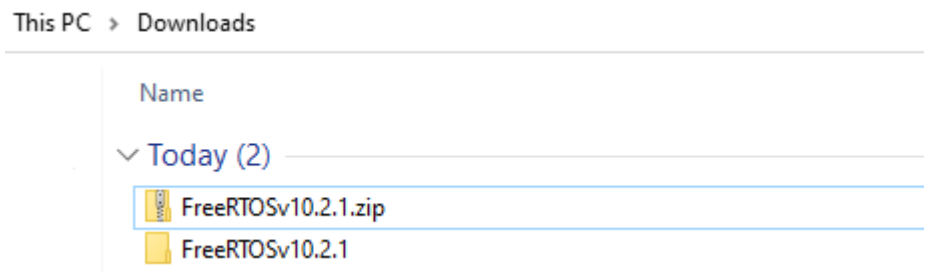

Figure 3.5 Download and unzip FreeRTOS kernel

2. Download Git for windows from the path, “https://git-scm.com/”. Install Git by executing downloaded .exe.

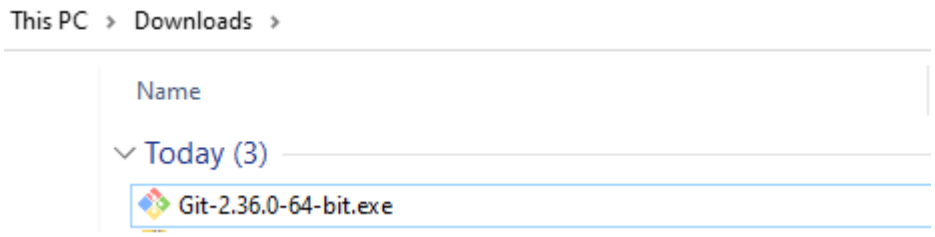

Figure 3.6 Download and install Git

3. Unzip the release package “MCUMotorStudio_1.1.zip”.

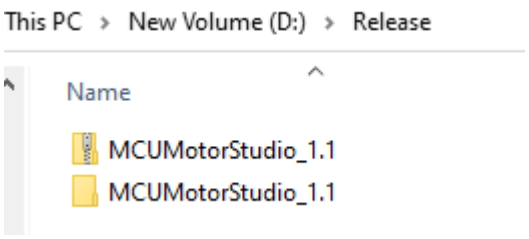

Figure 3.7 Unzip Release package

4. Unzip Firmware source code available at “MCUMotorStudio_1.1\Code”.

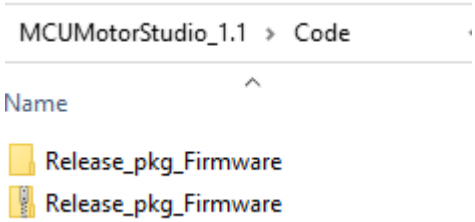

Figure 3.8 Unzip Firmware package

5. Create a temporary folder by name “patch” at any suitable path outside folder “MCUMotorStudio_1.1”.

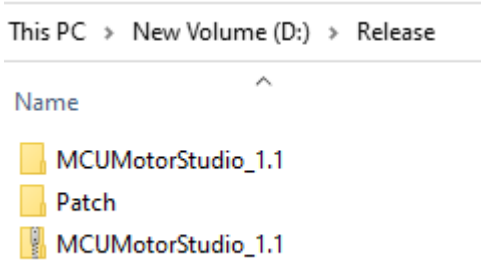

Figure 3.9 Create “patch” folder

6. Copy FreeRTOS source folder “FreeRTOSv10.2.1\FreeRTOS\Source” and patch file “MCUMotorStudio_1.1\Code\Release_pkg_Firmware\FreeRTOS.patch” to “patch” folder as shown below.

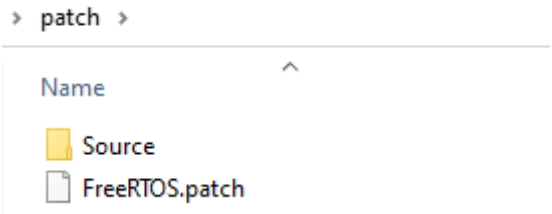

Figure 3.10 Copy FreeRTOS source and patch file

7. Open Git Bash application by right clicking on “patch” folder and clicking “Git Bash Here”.

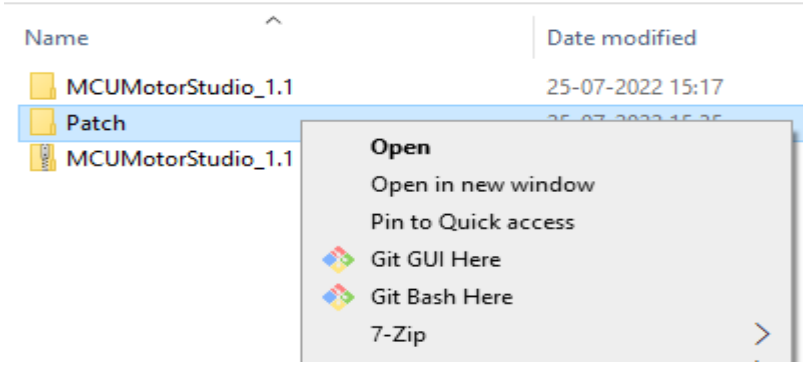

Figure 3.11 Open Git Bash

8. Execute following command in Git Bash. Git bash works on linux style commands.

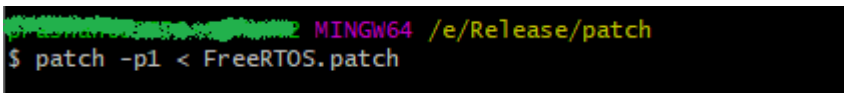

Figure 3.12 FreeRTOS patch command

9. If the patching is successful, message as shown below will appear.

Figure 3.13 FreeRTOS patch message

10. Finally copy “\patch\source” folder to “MCUMotorStudio_1.1\Code\Release_pkg_Firmware\Source\FreeRTOS” as shown below.

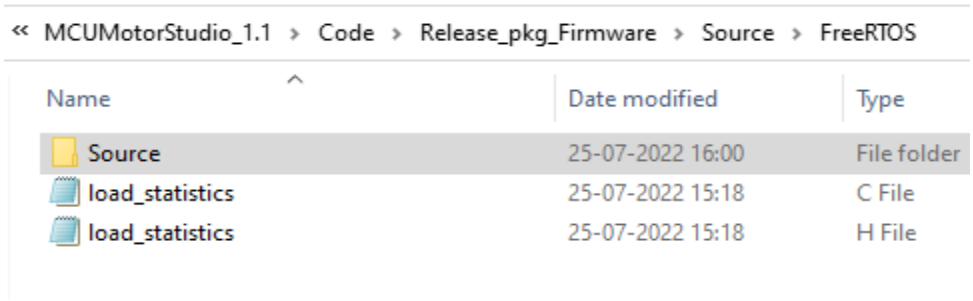

Figure 3.14 FreeRTOS patch message

11. Downloaded FreeRTOS kernel and temporary folder “patch” can be deleted after patching procedure.

3.4. Project Structure

The project structure for each individual target is kept identical with the layered structure of the Motor Control Firmware with one nesting level only, due to the limitation of the default development tool used (IAR Embedded Workbench).

All general configuration source files and the **main.c** are grouped together at the project top level. Besides the improved visibility and faster navigation through the various system parameters, it eases the set-up, tuning, reconfiguration and debugging of the currently selected and active project configuration.

C/C++ Compiler and Linker pre-processor definitions on project configuration level are and shall be avoided as much as possible, instead these shall be put in the corresponding configuration header files. The rule allows portability and eases the usage of several different tool-chains.

There are however a few exclusions from the rule. These affect the C/C++ Compiler symbol definitions only and are limited in their number. The desired configuration shall be reflected in the **Options** of the Motor Control project, sections **C/C++ Compiler:**

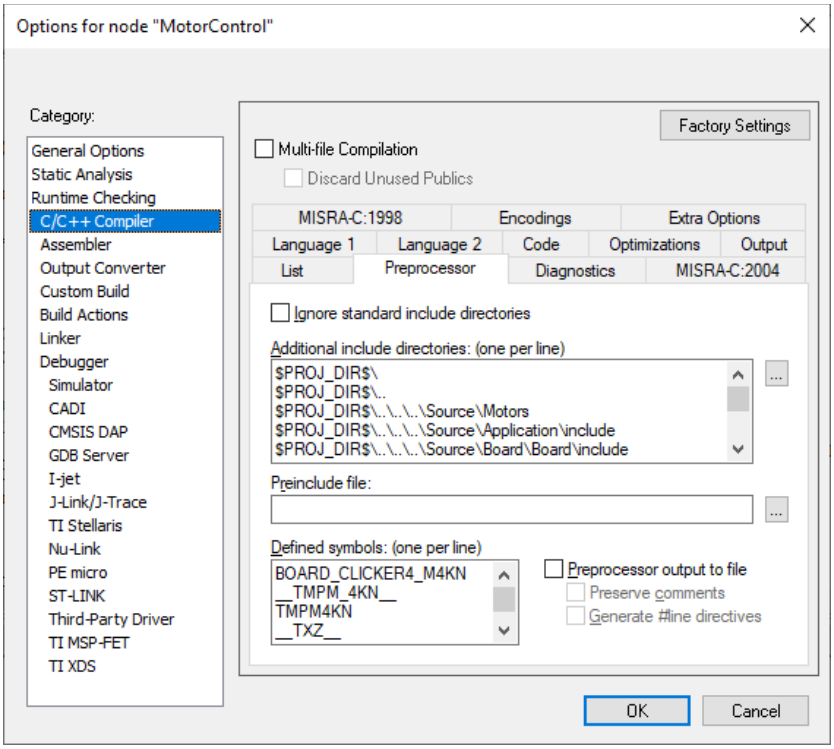

Figure 3.15 Motor Control Firmware C/C++ Compiler pre-processor definitions

Debug configuration (DEBUG) – enabling debug output and pre-processor assertions. Shall not be used with the release configuration. May be used in the debug configuration as addition to the source-level debugging

Board and MCU selection (BOARD_CLICKER4_M4KN, __TMPM_4KN__, TMPM4KN) – Some boards can be equipped with different MCUs from one family, due to the board design and/or the pin compatibility. For that reason, a top level configuration needs to be defined. The MCU selection requires two different definition styles. The underscored macro needs to be defined only if legacy sources files are to be included in the build

CMSIS tool-chain selection (_IARCMSIS_) – legacy definitions in the CMSIS drivers. Can be omitted with M4K

Board system set-up (__TXZ__) – M1K group specific definition for board configuration. Currently unused and can be safely omitted

Currently no Linker pre-processor definitions on project configuration level are done. In general, such are to be avoided, as these are very much tool-chain specific.

There is at least one project configuration, with debug mode enabled, for each MCU of TMPM4K families. For most MCUs a project with release mode configuration is defined as well. The default project configuration is the Clicker4 TMPM4KN on “Clicker 4 for TMPM4K” reference board in release mode:

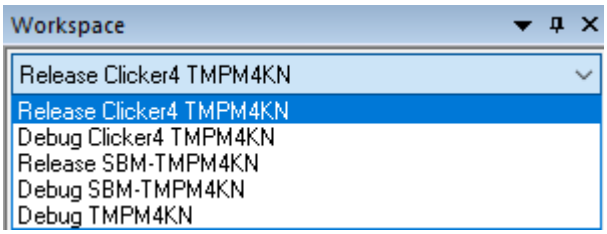

Figure 3.16 Motor Control Firmware IAR Project Configurations

The typical top-level workspace window is illustrated below:

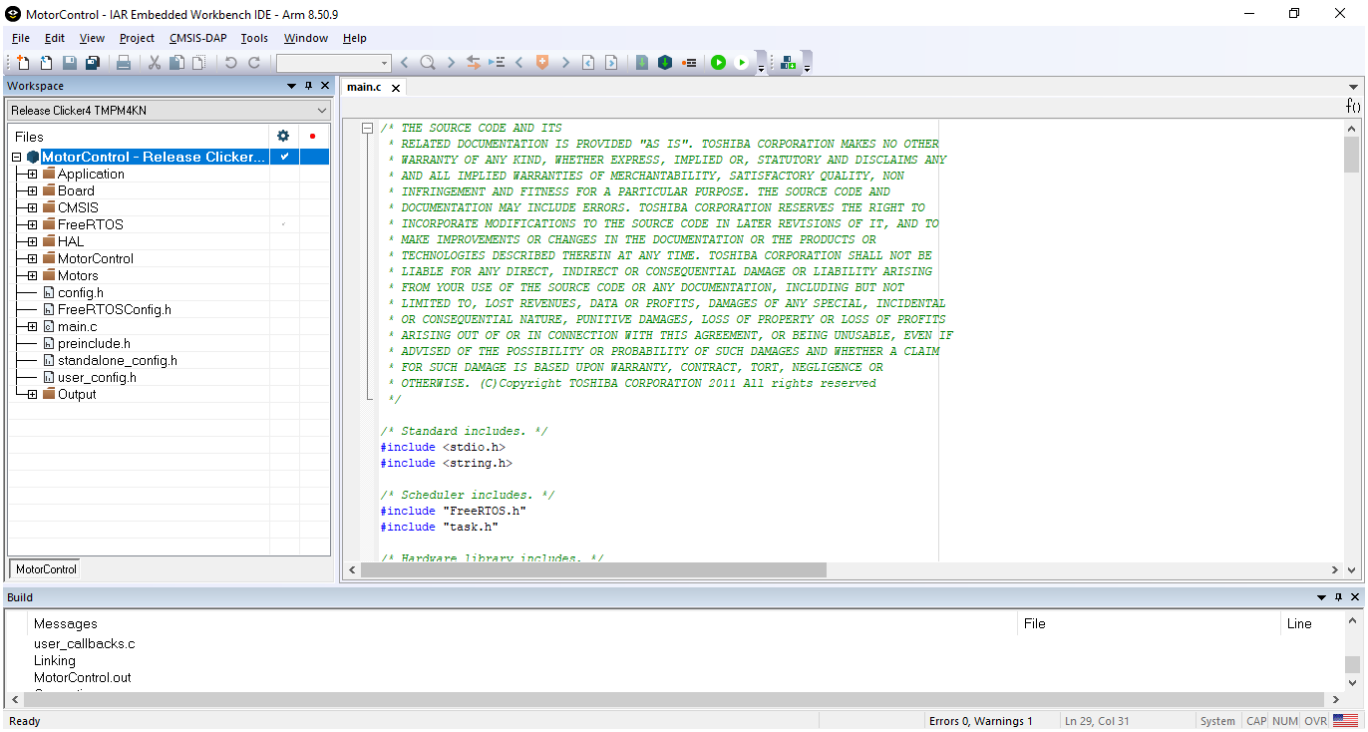

Figure 3.17 Motor Control Firmware IAR Workspace

The screen capture above is illustrative. The actual look of the IAR Embedded Workbench workspace may slightly differ from system to system and user to user, mainly depending on the used IAR Embedded Workbench version and the particular user configuration/preferences. The project/file structure will be identical for the selected active configuration, e.g. “Release Clicker4 TMPM4KN” in the example above. Note: It is recommended to use “IAR Embedded Workbench” version 8.50.9.

3.5. Configuration Files

Each project regardless of the release/debug configuration offers number of configuration options that are mandatory and need to be adjusted for the proper operation. Depending on the selected project, a number of additional options are also available.

The following common configuration files (C header) are project independent and any modification shall be avoided. Changing parameters and options in these require deep understanding of the firmware

structure and will directly affect core firmware or FreeRTOS behavior and/or characteristics. Caution shall be taken as any change affects all projects and may result exceeding of the available RAM memory:

FreeRTOSConfig.h – Real-time kernel configuration and scaling adopted to the Motor Control Firmware requirements, selecting the scheduling policy, the communication and synchronization objects, tasks configuration, memory usage and limitations, Task API functions inclusion/exclusion and the interrupt service routine mapping. Detailed description of each parameter can be found in the "Configuration" section(s) of the FreeRTOS documentation.

config.h – Common Motor Control Firmware configurations specifying the memory limits, individual task stack size and priority, clock selection, MCU and board dependent definitions and basic sanity check on the complete configuration

The following common configuration file (C header) is applications specific and shall be adjusted to each project configuration (debug or release):

user_config.h – configure the usage of watchdog timer, oscillation frequency detection, the exact control loop frequency, various convenience and protection functions, number of control channels, exact motor type, override the default board for the particular motor and the usage of the DSO/HS-DSO

standalone_config.h – configuration specific to the selected stand-alone application (this will be supported in future releases), allowing overriding of the used number of control channels, exact motor types and board. Various application related configurations like button control (in addition to the default command interface), power loss detection and safe demo parking (if supported in hardware and software), etc. It contains configurations for the four demonstrator projects that are delivered with the Firmware and may be extended accordingly.

Further details can be found in the upcoming "Configuration chapter".

3.6. Command Interface

A strict request/answer serial communication protocol between the external controller and the Motor Control Firmware is used to configure and control the motor(s) on the particular board.

The Toshiba MCU Motor Studio Control GUI is the default controller utilizing UART with 115200, 8N1 to send requests and evaluate the Firmware responses. Only one request can be processed at a time. The next one can be dispatched after the previous one was answered or timed out.

The supported commands, their format, number and type of parameters and all possible answers are detailed in its Protocol Specification.

The communication protocol is realized at the application layer and uses UART by default. Other serial or parallel communication interfaces, like SPI, CAN, etc. can be integrated and used instead. Such adaptation requires moderate changes solely in the **protocol.c** file - replacing all UART specific initialization, transmit, receive and error handling functionality with the ones of the newly selected interface.

3.7. Data Logging

The Firmware supports in combination with the "MCU Motor Studio" tool logging of the most important control and motor parameters in a Digital Storage Oscilloscope (DSO) (or High Speed DSO, in future releases) fashion.

Up to 8 parameters, like rotational speed Theta (θ), electrical angle Omega (Ω), flux axis current Id, torque axis current Iq, phase voltages Va, Vb, Vc, phase current Ia, Ib, Ic, etc. can be selected for simultaneous logging. Please refer to the "MCU Motor Studio" User Guide for the full list of supported parameters.

The firmware will perform a snapshot of the selected signals at a defined spread factor. An optional

trigger condition, including threshold value, can be configured to set the sampling start condition. The signals are sampled with constant user defined spread factor that ranges from 1 to 255, where 1 means each PWM cycle, 2 every second PWM cycle and so on.

The tick duration and total recording time are also fed back. The tick duration is based on the PWM frequency, whereas the total recording time depends on the tick duration, the spread factor and the number of signals.

Alternatively, simplified statistics with five control parameters is provided as extended motor state that can be polled via the dedicated command protocol request. The parameters are fixed, alternating the target/current control value and non-control value depending on the selected control method – by speed or by torque. MCU Motor Studio has built-in support for this state retrieval via its statistics tab, featuring an optional CSV logging capability. The definition of the **MotorExtendedStateSettings** can be found in **api.h**, which logically belongs to the Motor Control layer and contains definitions of parameters and APIs which realize all functionality as invoked by the Application layer based UART command interface protocol. The following parameters are used:

ActualControlledParameter – being either the actual rotation speed in RPM/Hz or the actual torque of the Motor in Ncm. Negative values shall be used for counter clockwise (CCW) rotation only when controlled by speed.

TargetControlledParameter – being either the desired rotation speed in RPM/Hz or the desired torque of the Motor in Ncm. Negative values shall be used for counter clockwise (CCW) rotation only when controlled by speed.

Current – Applied current in mA

NonControlParameter – being either the actual torque of the Motor in Ncm or the actual rotation speed in RPM/Hz. Negative values indicate counter clockwise (CCW) rotation

Timestamp – a relative timestamp measure in number of system timer ticks

ControlMethod – Control by Speed (default) or by Torque

3.8. Error Handling

The common error handling & recovery strategy utilizes simple API return values allowing check against successful or failure completion of the requested task. The error code is signed integer and in most of the cases does not provide deterministic information of the exact malfunction. Depending on the error type and the application logic a recovery may be possible and shall be implemented and enforced by the user.

Due to memory limitations most validity checks on the input data and parameters in each operation are done with an assert macro and in debug configuration only. These are blocking calls resulting operation abort and may result irresponsive system from the communication protocol point of view.

If a protection function is configured and activated, the dedicated error handler will be given control and the desired action will be carried out. As most of the failures are not recoverable the handler will simply trap the execution or perform controlled/emergency motor stop.

3.9. Comment styles

Although both C and C++ style comments are generally allowed, the native C commenting style is preferred and used thought for better portability and readability.

The Firmware implementation adopts the Doxygen standard specification, allowing automatic API documentation generation from annotated C/C++ sources. For details please refer to the official tool website: https://www.doxygen.nl/manual/.

As some of the layers, namely RTOS and CMSIS are not under full implementation control, these may be exception of the rule.

4. Detailed Layer Description

4.1. Application Layer

The application layer specifies and implements the top-level services and the communication protocols for user interaction – drive control, data exchange, status retrieval and diagnostic collection. There are two major methods for system operation further described in the subsequent chapters – built-in and external.

As for every layer, both the implementation and its project representation are grouped in single and dedicated folder, naturally called **Application**. The subfolder **include** hosts all top-level application specific definitions and configurations. The **Standalone** folder contains the full implementation of built-in reference stand-alone applications.

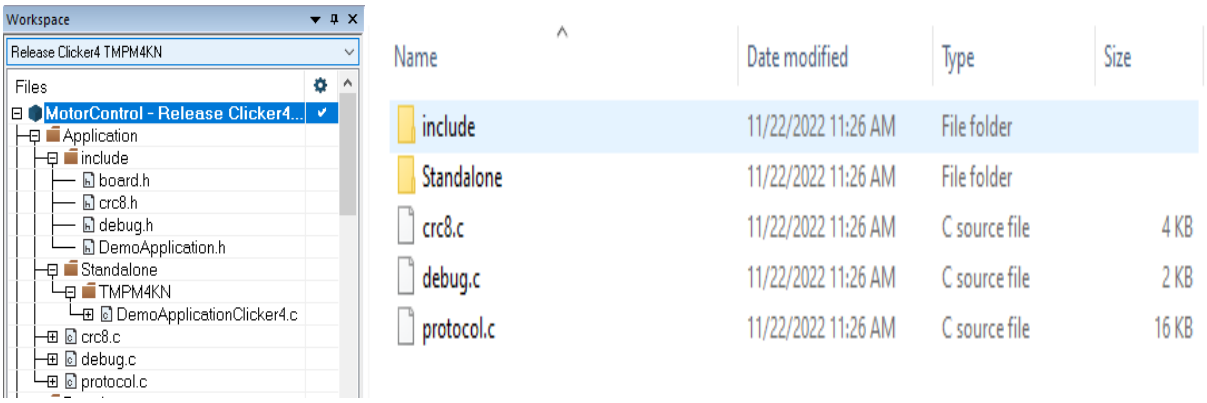

Figure 4.1 Application layer project and folder structure

The following C source and header files are part of the layer implementation:

board.h – the top level generic board definition file, specifying the minimal initialization and versioning related APIs. It abstracts the underplaying main board and is inherited and partly overridden by the Board layer hardware specific definitions.

debug.c and **debug.h** – several levels of code instrumentation and print-statement based debugging is supported. All these are implemented via printf and assertion macros and configured via C/C++ preprocessor definition **(DEBUG)** or direct inclusion/exclusion via **#if** and/or **#ifdef** preprocessor directives.

The communication protocol is implemented as a single RTOS task handling message reception/transmission and various communication errors in dedicated generic state machines that are executed in the interrupt handler context of the selected communication interface. Although the generic protocol is intermixed with the particular communication interface (UART) implementation, the decoupling or porting to another one, like SPI, CAN, etc. is rather straightforward. It requires simple replacements of an initialization routine, namely **uart_init()** within **protocol_init()**. Additionally the reception buffer retrieval **HAL_UART_GetReceivedData ()** and transmission buffer filling **HAL_UART_SendData()** in the corresponding state machine has to be adjusted to the used communication module. Finally, the error specific status check and interrupt clearance shall be exchanged in the generic error handling state machine. All protocol definitions and declarations are placed in **protocol.c** and **protocol.h**.

The default communication interface uses a simple 8-bit cyclic redundancy code (CRC) for single bit error detection over the command and data payload. It is a table driven implementation of the CRC-8 as defined by the System Management Bus (SMBus) Specification, version 2.0. All CRC definitions and functionalities are provided in **crc8.c** and **crc8.h**.

4.1.1. External Control (in future releases)

No dedicated application is executed on the MCU, all commands are fed from external interfaces that may, although not recommended, be intermixed. Currently two types of control interfaces are implemented – basic and extensive.

4.1.1.1. Basic Speed Control

Speed and direction commands are input via small number of selected IOs, utilizing non-protocol based mapping between the IO functionality and the control parameter, like PWM duty or ADC input measurement results representing the requested target speed. All these are implemented in one single file, the **external_speed_control.c**. Advanced features usage and dynamic configuration is limited or cannot be used.

The assigned IOs for direction and speed input shall be defined on base board configuration level, in the main board specific header file under **~\MotorControl\Source\Board\Board\config**. The following definitions may be carried out:

Channel – the mandatory definition **ESC_CHANNEL** specifies at compile-time the single channel that shall be controlled via external signals. Multiple channels and dynamic selection is not supported.

Direction – the general-purpose IO that shall be used as input for direction setting is described via the **SPEED_CONTROL_CWCCW_PORT** and **SPEED_CONTROL_CWCCW_PIN** definitions. Logical “1” on the input is interpreted as counter-clockwise (CCW). The definitions are mandatory.

Error status – the optional definition of the general-purpose IO that shall be used as output for signaling an error detection may be specified via **SPEED_CONTROL_FAULT_PORT** and **SPEED_CONTROL_FAULT_PIN**. The output will be driven, given at least one of all supported error states, is detected and entered.

Rotation indication – an optional definition of the general-purpose IO, which shall be used as output for signaling a motor rotation, may be specified via **SPEED_CONTROL_FG_PORT** and **SPEED_CONTROL_FG_PIN**. The output will be alternated 6 times pro electrical turn, as the speed change detection is based on the sector information provided by the Vector Engine. The feature is not yet available for the software controlled channels.

Target Speed – several partly non-mutual exclusive options for external speed setting are available, these are compiled-in by providing a valid definition for either ADC or PWM configuration:

ADC Control – the target speed is based on analogue voltage measurement and scaling. The **SPEED_CONTROL_ADC_PORT**, **SPEED_CONTROL_ADC_PIN**, **SPEED_CONTROL_ADC_CHANNEL**, **SPEED_CONTROL_ADC_REG** definitions describe the ADC channel/pin and its configuration. The **SPEED_CONTROL_ADC_HANDLER** and **SPEED_CONTROL_ADC_IRQ** specify the interrupt to be used.

PWM Control – three different modes, referred as “PWM Duty”, “Servo Min to Max” and “Servo 0 to Max” are supported. The difference is in the way the captured PWM signal is interpreted – either the duty is scaled to speed or the servo signal pulses are converted to negative/positive speed or positive speed only. The **SPEED_CONTROL_PWM_PORT**, **SPEED_CONTROL_PWM_PIN** and **SPEED_CONTROL_PWM_TMRB** definitions specify the used PWM channel/pin. **SPEED_CONTROL_PWM_HANDLER**, **SPEED_CONTROL_PWM_IRQ**, **SPEED_CONTROL_PWM_HANDLER2** and **SPEED_CONTROL_PWM_IRQ2** define the used interrupt and handlers for the signal capture.

The basic external control was initially implemented for an older firmware architecture. Its configurability and extensibility is limited and may require rework in the future.

4.1.1.2. Extensive Speed Control (not supported)

Protocol based, utilizing (by default) a configurable HS-UART channel for commands and data exchange. This method allows parameters change and full usage of all supported control modes and advanced

functionality. The protocol implementation is encapsulated in **protocol.c**, complemented by simple error correction is **crc8.c**. The command and parameter definitions are part of the Motor Control layer, **api.c**, **api.h** and **api_internal.h** in particular.

All supported commands, their parameter list and usage are described in the “MCU Motor Studio Protocol Specification”.

4.1.2. Standalone Demo

The Standalone demo can be driven in two different ways:

1. Using Slider and Buttons (does not require connection to PC Tool)
2. Demo Control Window of the PC Tool.

4.1.2.1. Demo using Slider and Buttons

The demo can be operated without the need of Motor Studio PC tool. The demo uses “Slider 2 click”, connected to the MicroBus on Clicker 4 MCU board. The “Slider 2 click” along with buttons, can be used to demonstrate Start, stop and speed control.
The demo can be enabled from “Projects\MotorControl\ standalone_config.h” by Uncommenting the macro **DEMO_CLICKER4_SLIDER2**.

standalone_config.h

```
#ifndef _STANDALONE_CONFIG_H_
#define _STANDALONE_CONFIG_H_

#include <stdint.h>

/* Standalone applications for TMPM4KN */
#if ( (defined __TMPM_4KN__) || (defined __TMPM_4KN__) )

/* Select standalone application */
//#define DEMO_CLICKER4_SLIDER2                                   /*!< Activate "Slider 2 Clicker" demo on Clicker 4 */

#if (defined DEMO_CLICKER4_SLIDER2)

#define STANDALONE_APP

#define STANDALONE_APP_TASK_STACK_SIZE        (200)

#ifdef DEMO_CLICKER4_SLIDER2
/* Demo configuration */
#define USE_DEMO_CONTROL                                        /*!< Activate the demo related commands in the UART command protoco
#define USE_SLIDER_2_CLICKER                                    /*!< Use the Slider 2 Clicker - for speed control */
#define USE_LED                                                 /*!< Activate LED signaling */
/* Motor configuration*/
#define MOTOR_CHANNEL_0 "motor_define_ACT_42BLF01_3sh_opa.h"    /*!< Motor to be used */
#endif /* DEMO_CLICKER4_SLIDER2 */

#endif /* (defined DEMO_CLICKER4_SLIDER2)*/

#endif  /* TMPM4KN   */
```

Figure 4.2 Stand-alone application configuration in standalone_config.h

4.1.2.2. Demo Control Window on PC Tool

The Demo Control window provides a basic control for the demo application, offering several buttons for state change and a simple state/transition status indication. However, the set speed is still controlled via the **Slider and the direction via button B5**.
Pressing the Run button will start the motor and the active state will become “Running …”:

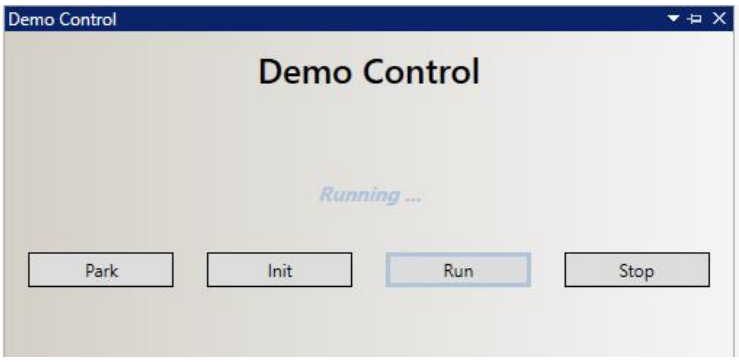

Figure 4.3 Demo control Window in "Running"

Pressing the Stop button will stop the motor and the active state will become "*Stopping …*". As soon as the motor stands still, it will be changed to "*Idle*":

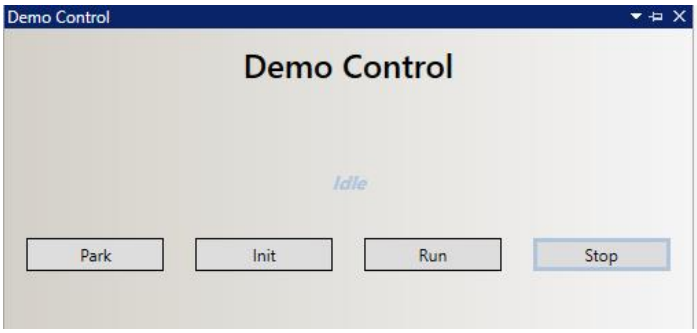

Figure 4.4 Demo Control Window in "Idle"

As both **Park** and **Init** states are not supported, pressing the corresponding buttons will result no action. LEDs L2, L3, L5 and L6 will be updated accordingly. Mixed usage of Run/Stop and B6 buttons is supported.

4.2. Board Adaptation Layer

The board adaptation layer specifies and implements the generic framework for integration and customization of the various hardware platforms the firmware is to be executed on. The Motor Control Firmware logically differentiates between the following platform components, regardless of their physical location (on one and the same PCB or dedicated PCBs):

Base (main) boards – the main logical component defining the microcontroller interconnectivity, used frequency, connectors, expanders and status indicators. Simply referred as board in the description and the implementation.

Power (output) stages – logical unit for all power components, including but not limited to the inverter circuitry (H-Bridge/MOSFETs), current and DC link measurement circuitry and overvoltage/overcurrent protection circuitry.

Temperature sensing solutions – the temperature sensing solution in terms of characteristics, interconnectivity and thermal characteristics are grouped here.

External components – all other optional components, mainly interface and communication related like keypads, LCD displays, transceivers, etc.

The folder and the folder structure follows the logical split. All board adaptation related files are grouped in single and dedicated folder **Board** that contains sub-folder for each of the platform component types, e.g. **Board, ExternalComponents, Power** and **Temperature**:

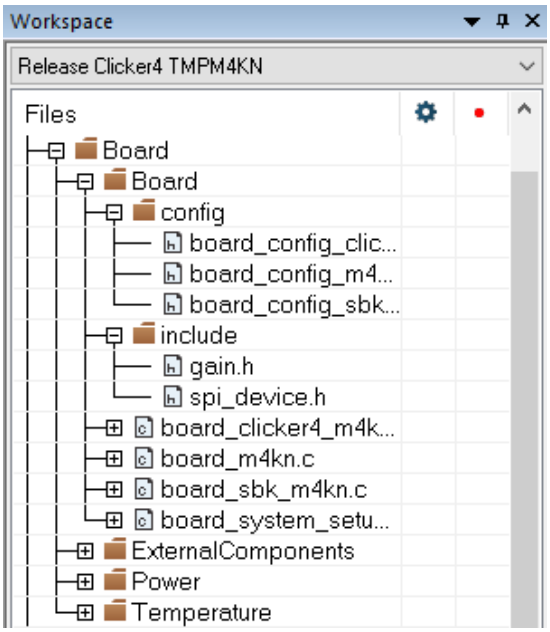

Figure 4.5 Board adaptation layer project structure

All initialization and configuration routines shall be added to the **board_system_setup.c** and will be invoked from **main.c** at the system initialization stage.

4.3. CMSIS Layer

The layer encapsulates the Cortex Microcontroller Software Interface Standard (CMSIS) compliant drivers provided by TOSHIBA for each family/family member. These allow a vendor-independent hardware abstraction of the used ARM Cortex microprocessor.

All CMSIS layer files are grouped in one single folder, named **CMSIS**, for both the project and folder structure. It is further sub-divided into individual families, typically using same microprocessor core: TXZ+.

There are two types of implementation files – system and startup. As the name suggest startup is pure assembly file that is providing the entry point, vector table and start sequence. System contains the basic core initialization functionality needed for proper start-up.

A header file containing definitions for the exceptions/external interrupts and all peripheral modules for each family or member is also supplied. It contains registers, bits and access macros definitions.

Direct usage of the access macros defined on the CMSIS driver layer is depreciated in favor of the interface provided by the corresponding hardware abstraction layer driver.

There are no CMSIS drivers for some of the available peripheral modules, either as these are not yet standardized by ARM or these are not currently supported and supplied by TOSHIBA.

The CMSIS project structure in IAR Embedded Workbench is depicted below:

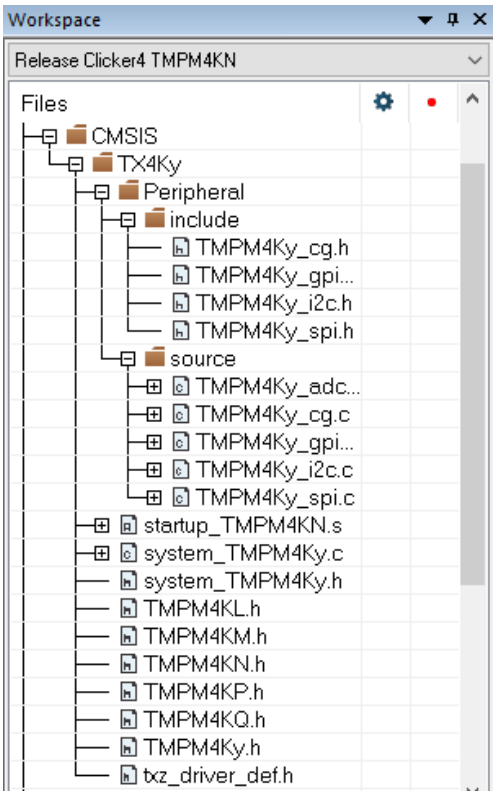

Figure 4.6 CMSIS layer project structure

4.4. Hardware Abstraction Layer (HAL)

The hardware abstraction layer specifies and implements light weighted driver services for full abstraction of the underlying microcontroller peripherals. Both the implementation and its project representation are grouped in single and dedicated folder - **HAL**. The subfolder **include** hosts definitions and configurations of all peripheral modules, including the ones that are not currently used by the firmware. Number of family specific (M4KN) sub-folders hold the driver and services implementation:

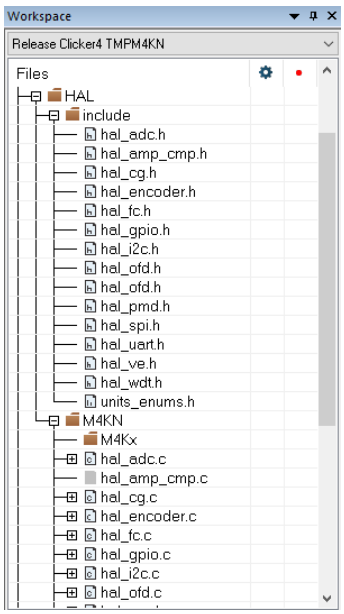

Figure 4.7 Hardware abstraction layer project structure

The hardware abstracted peripheral drivers are based, extend and unify the access to the CMSIS driver layer. Differences such as number of supported channels, base addresses and supported functionality are internally handled and hidden from the service and the application layers.

Any cross-module functionality usage, such as clock gating or IO configuration, is to be exclusively done via the HAL layer interfaces. Only the module internal implementation may refer to the lower layers, CMSIS in particular.

4.5. FreeRTOS Layer

Although not mandatory the usage of a RTOS for resources (mainly CPU time) management and prioritization is highly recommended. The Motor Control firmware adopts the FreeRTOS in a minimal configuration. The entire RTOS implementation and the system load statistics, as it is RTOS specific, are grouped in this layer.

All user and application related configuration are grouped in **FreeRTOSConfig.h** on top level. For further customization or adaptation please refer to https://freertos.org/.

4.5.1. Load Statistics

The firmware utilizes the RTOS idle hook and a software counter to dynamically measure the CPU load in terms of “number of times the idle hook is entered per fixed amount of time”. A reference mean value is measured with the default firmware configuration and non-active channel control, representing the no-load condition with maximal number of entries. During normal operation the reference software counter value and the currently captured one are compared and the actual load is calculated.

Please note that the reference “zero load” point represented by the **MAX_IDLE_COUNTER_NO_LOAD** definition, shall be measured and adjusted for each particular configuration, family member and tool-chain. Otherwise the load indication might be imprecise or even incorrect.

The implementation is encapsulated in **load_statistics.c** and **load_statistics.h**.

4.6. Motor Control Layer

The Motor Control Layer is a services layer. It defines and implements independent software components, providing various functionalities to the upper layers, the application layer in particular.

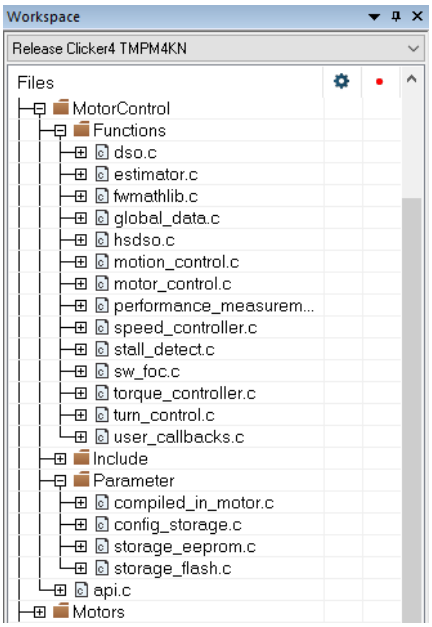

Figure 4.8 Motor Control layer project structure

The implementation and its project representation is grouped in the Motor Control folder, which is further divided into **Include, Parameter** and **Functions**. The definitions and the APIs of each software component are provided in one or several header files collected under **Include. Parameter** contains all definitions and implementation related to motor, encoder, systems, board and PI regulation parameters,

their initialization and storage. **Functions** is the container for the implementation of each supported service.

Due to the complexity of each software component/service, these will be described in dedicated sub-chapters.

4.6.1. Motor Controller

4.6.1.1. Hardware & Software Control Types

The control of a motor channel is performed using the dedicated hardware accelerators, the Advanced Vector Engine/Vector Engine (A-VE/VE), the Programmable Motor Driver (PMD) and the ADC. Typically, one instance of each accelerator is needed per channel. Depending on the family member a maximum of two instances of the Vector Engine are available, which limits the number of hardware controlled channels. A sequential usage of a single Vector Engine for multiple-channel control might be possible, but with severe limitations, the PWM update frequency (control execution rate) being one of these. A better alternative for increasing the number of supported channels is to perform the Vector Engine computation in software. It significantly increases the CPU load and is mostly used with the Cortex-M4 based TXZ/TXZ+ families.

The Motor Control firmware supports one hardware (motor 0) and up to two software controlled (motor 1 and motor 2) channels. A second hardware channel can be added for selected families/members. The assignment is static.

4.6.1.2. Processing Loops

The top-level control of the motor drive is performed in two independent loops.

4.6.1.2.1. Interrupt Loop

A periodic lag angle compensation and control current adjustment is performed in the context of the Vector Engine interrupt handler for the hardware controlled channels and in the context of the ADC completion interrupt handler for the software controlled ones. The occurrence rate is determined by the set PWM update frequency. The main hardware control handler **IRQ_Common()** is defined in the vector engine's HAL layer driver. The handlers **INTADBPDB_IRQHandler()** and **INTADCPDB_IRQHandler()** of the software control are defined in **sw_foc.c**.

A typical **Interrupt loop** control cycle consist of:

PI control – the reference d-axis and q-axis currents are regulated with the selected gain and the corresponding axis voltage is being calculated.

Inverse coordinate transformation – Inverse park transformation to get the time variant representation of the two phase voltages.

Phase transformation and space vector conversion – two to three phases transformation and voltage calculation, including internal sector information preparation.

Output Control – calculation and adjustment of the duty for the PMD wave generation and the trigger point for the next ADC conversion.

Input processing – retrieval of the ADC measurement results, the three phase coil currents and DC link voltage and pre-processing these to fixed point format.

Input current conversion – three to two phase and time variant to time invariant conversion, resulting the actual Id and Iq which will be then fed to the PI controller in the next cycle.

All these are pure mathematical operations that are performed in the vector engine for the hardware controlled channels. The exact computation steps are described in the Vector Engine's Reference Manual, Chapter "Description of Tasks". As the algorithms and used formulas are well documented and

fixed, a full replication in software was implemented for the cost of extra CPU load. This can be found in **sw_foc.c**.

Final processing is performed in the end of each cycle. In the case of hardware controlled channels this is signaled with a VE completion interrupt generated after the completion of the last task in the currently selected schedule, typically the **Input current conversion**. As the software controlled channels cannot utilize the vector engine the control cycle is shifted. It starts with **Input processing** and end with **Output Control** completion that result ADC conversion at the selected trigger point. The cycle end is signaled by the ADC conversion completion interrupt generation.

The final processing of the hardware control type includes:

Vector Engine Configuration – set/re-set the output mode, schedule and start task

Measurement Results Evaluation – current and DC link measurement results are being read, post-processed and stored.

Motor Power Calculation – current motor power consumption is being calculated

Software Overcurrent Protection (if configured-in) – current values of Iq and Id are checked and if exceeding the pre-calculated threshold, the emergency stage is enforced

Angle & Speed Calculation – performed in Forced and FOC stages only. The control method specific, either **SpeedController_Omega_Theta()** or **TorqueController_Omega_Theta()**, is invoked. Please refer to the Control Methods sub-chapter for further details.

Current Calculation – depending on the control method, either **SpeedController_Calculate_Id_Iq_ref ()** or **TorqueController _Calculate_Id_Iq_ref ()** are invoked, if the current stage is one of Forced or FOC. Details can be found in the Control Methods sub-chapter.

Drive/Break Control – dependent on the current stage the control values are set to either zero (Break, Stop, Emergency) or the newly calculated reference values for angular speed, lag angle and torque-axis current.

Disconnect Detection (if configured-in) – current values of **Ia, Ib** and **Ic** are monitored and compared towards the expected. If it is below a percentage for certain number of control cycles, a motor disconnection will be signaled and the emergency stage is enforced

Position Estimation – count the number of electrical turns and sector changes to estimate the current position in case there is no external sensor connected to the rotor or a higher precision is needed.

The final processing of the software control type includes similar steps but shifted as it has different cycle completion/start point.

4.6.1.2.2. Control Loop

The top-level speed or torque control with optional position estimation is performed on regular intervals in the control loop. It is initialized in a dedicated RTOS task, but executed in the context of a periodic timer interrupt with a rate of approximately 1ms. The generic handler **MotorControl_Loop ()** is defined in **motor_control.c** and serves both the software and the hardware controlled channels.

The handling in this loop is again stage dependent. The basic principle is however that the actual control parameter (angular speed or torque) is being calculated (estimated), compared with the set one and corrected accordingly. As the interrupt loop is executed with the PWM update frequency the speed change is not recognizable in it.

Furthermore, the stage changes, especially Forced<->FOC, are normally decided and triggered within this loop.

4.6.1.3. Processing Stages

Independent of the selected control method, the processing goes through several stages:

Zero Current Measurement Stage – this is single scheduled vector engine task; whose purpose is to calibrate the current measurement. The input current is set to 0, the ADC conversion results are read and the mean ADC conversion result that is captured sequentially for number of cycles is taken as the actual “zero” value. The default ADC conversion result representing a zero is 0x7FF. It may vary dependent on various factors like temperature, supply voltages, tolerance of the external components, etc. Typically, it will be re-adjusted and set to a value in the range of 0x800 to 0x820.

Bootstrap Stage – in this stage the low side MOSFETs of the H-bridge are switched on, while keeping the high side off. This allows the charging of the bootstrap capacitors via a diode and resistor to the **Vdd** supply voltage and effectively ensuring the needed potential difference for driving the high side.

Initial Positioning Stage – a permanent non-moving electromagnetic filed is generated by gradually increasing the q-axis current component until the maximum calculated one **MotorControl[channel].PreCalculation.IqCurrentForInitposition** is reached. The target angular speed is set to 0, the angular position to -/+ 90 degrees depending on the direction. A minimum of **Position delay** time is awaited. The firmware does not check whether this time was sufficient and the rotor has reached and is holding its start position.

Forced Stage – a sine-wave open-loop commutation with constant current **Iq start** until either the set speed is or the **Change speed** is reached. In case of the later a switching to the **FOC stage** will be done

FOC Stage – the final stage in which a field-oriented control is performed. It requires sufficient back EMF and is typically possible whenever certain predefined/preconfigured rotational speed is achieved. The angle of the phase current vector lags behind that of the back electromotive force (EMF) vector due to the motor phase inductance. A correction is performed in each cycle. The difference of the set rotational speed and the calculated/estimated one is checked in the control loop and result a change of the Iq reference as input to the PI regulation.

Break Stage – controlled stop of the motor in the short breaking mode, e.g. alternating short of the upper and lower side

Stop Stage – the default idle stage with the motor standing still, no outputs driven and no drive control.

Emergency Stage – the stage is similar to the zero current measurement in terms of executed schedule/tasks, but its purpose is to actively drive the MOSFETs to inactive, enforcing active overcurrent protection and resulting motor stop.

Most stages are further divided in sub-stages reflecting the need of certain initialization/preparation or clean-up upon change from one stage to another.

4.6.1.4. Control Methods

4.6.1.4.1. Speed Controller

The firmware will go through initial positioning, forced mode and if achievable (speed target is above the Change speed) will reach the FOC stage:

The overall control looks like:

Forced mode – constant torque-axis current, the **Iqstart**, is set and maintained in the interrupt loop. The angular speed is gradually increased with the maximal angular acceleration in the control loop, while the lag angle is compensated in the interrupt loop.

FOC mode - torque-axis current, the **Iq**, is monitored and adjusted in the interrupt loop. The angular speed is estimated in the control loop, adjusted and maintained indirectly via the lag angle compensation

in the interrupt loop.

The specific implementation is done in **speed_controller.c**. The generic control loop is provided in **motor_control.c**. The implementation of the interrupt loop is control type dependent - in **hal_ve.c** for the hardware controlled channels and **sw_foc.c** for the software controlled ones.

4.6.1.4.2. Torque Controller

The method is meant to ensure stable torque output while the motor is revolved with a desired speed. If the speed can't be maintained than it will be reduced.

The firmware will go through initial positioning, forced mode and if achievable (the speed that can be maintained is above the Change speed) will reach the FOC stage:

Forced mode – constant torque-axis current **Iq**, whose value is dependent on the torque (constant) factor of the motor, is applied. The angular speed is gradually increased with maximum the maximal angular acceleration in the control loop, while the lag angle is compensated in the interrupt loop.

FOC mode – the constant torque-axis current **Iq** is further supplied. The angular speed is estimated in the control loop, and reduced if the lag angle can't be compensated or requires **Iq** increase.

The specific implementation is done in **torque_controller.c**. The generic control loop is provided in **motor_control.c**. The implementation of the interrupt loop is control type dependent - in **hal_ve.c** for the hardware controlled channels and **sw_foc.c** for the software controlled ones.

4.6.1.4.3. Speed Estimator

Implemented in **estimator.c** and often referred as Position estimator in the code comments and older documentation, this service is a pure mathematical computation of the PI regulation error that is applied to the currently set angular speed to determine the actual one. The calculation considers the dropouts caused by the resistance and inductance of the coils, the induced voltage, the proportional and integral errors and the set reference (desired) angular speed.

The formulas behind are commonly known and properly described in the code comments. The main equation is: **Ed = Vd - R*Id + ωest* Iq *Lq**

4.6.2. Turn Control/Advanced Turn Control

Turn Control is a service for sensor-less rotor positioning performed in units of whole physical turns, which was initially developed for the TX03 and TXZ series. The Advanced Turn Control is an extension brought with TXZ+ family that adds support for partial turns in terms of physical degrees (internal calculation is done in electrical) and improves the drive profile calculation and parameterization. The latter is automatically used for all families at present, as it is backward compatible - simply keep the additional angle to 0.

A field stall or device lift recognition is automatically performed. Limited recovery is possible, but for the price of position information lost.

The service is disabled by default and can be added at compile-time by enabling the **USE_TURN_CONTROL** macro in the top-level user configuration **user_config.h**.

It relies on a position estimation purely performed in software. It is referred to as "Advanced Software Positioning" and described in the next sub-chapter.

4.6.2.1. Advanced Software Positioning

The Advanced Software Positioning is an open-loop sensor-less solution for precise rotor positioning utilizing the Cortex-M4 computational power in combination with the dedicated motor control hardware accelerators.

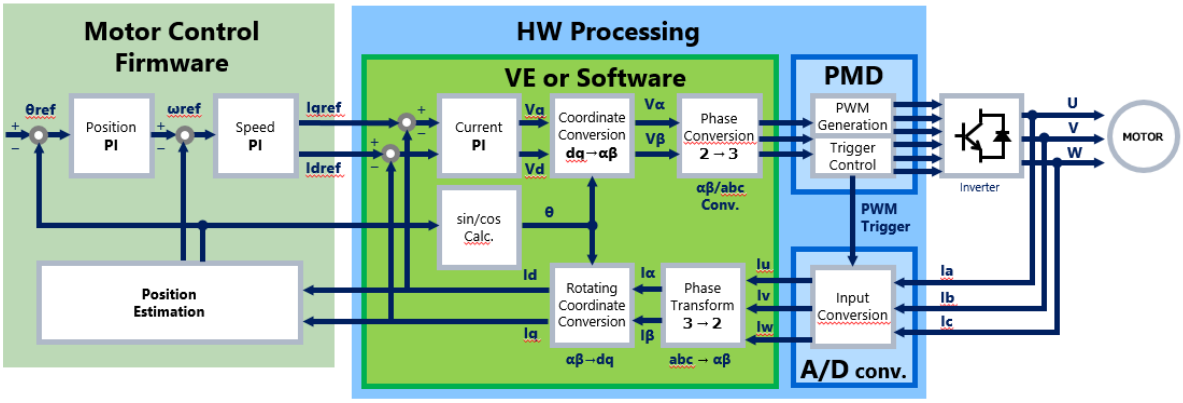

Figure 4.9 Advanced Turn Control Functional Diagram

The actual angular speed is dynamically calculated by the software speed estimator. It uses the periodically sampled Iq & Id values in combination with the reference/set ones to determine the induced current. Considering the exact motor characteristics (resistance and inductances), an estimation of the actual rotation speed, based on the calculated inducted current, is performed. The sampling interval is function of the used PWM rate, typically 62.5µs.

The new electrical angle **ɵ** (theta) is the position change for the fixed sampling interval ΔT with the current rotation speed ωest (omega estimated):

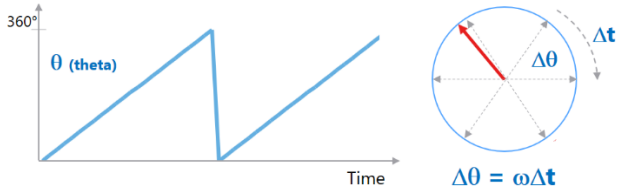

Figure 4.10 Full physical turn counting

The current position information is computed in the interrupt loop where all the need information, namely current sector, estimated speed, lag angle and angular speed, are available and actual.

4.6.2.2. Drive Profile & Configuration

Most of the drive functionality is implemented in the **turn_control.c**, including the exact profile calculation with ramp-up, ramp-down times and the final approach. Only one command may be served at a time per channel, as the service runs in separate RTOS task. A motor has to be stopped before a new command can be dispatched. Abort is always possible.

A typical drive profile looks like:

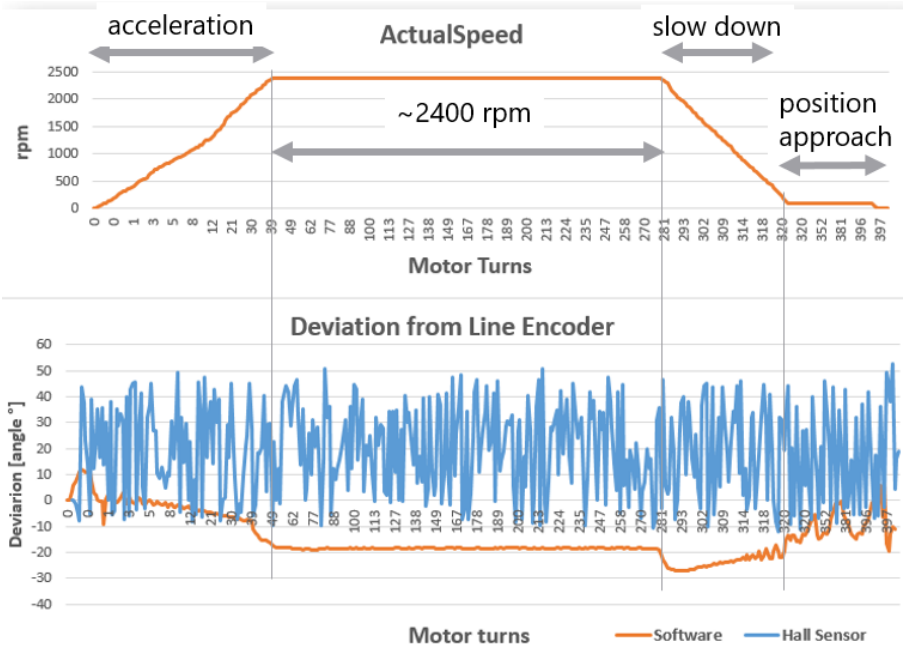

Figure 4.11 Advanced Turn Control Ramp & Driving Profile

The operation starts with pre-processing of the positioning request. The firmware will determine and configure the most suitable drive profile for the current request. All these operations are performed in **CalculateTurnRampAndTurn()**.

The following parameters will be calculated, taking into consideration the units of the request parameters (e.g. Speed in Hz or RPM, etc.):

Maximal Acceleration Rate – the rate of angular speed increase that is possible for the selected number of full turns and additional angle. It may be below the maximum supported by the motor.

Maximum Drive Speed – if the requested speed in not achievable for the number of turns that have to be done, it will be gradually reduced to the maximal possible one. The calculation includes the entire drive path (speed-up, revolve, slow down, final approach, stop at position).

Time to Speed-up/Slow-down – used for calculating the number of turns that the rotor will do while increasing or decreasing the speed

Full Turns for Speed-up/Slow-down – the number of physical turns for the speed up and slow down

Approach Speed – the maximal speed after slow down at which the motor can be precisely positioned and stopped without overshoot.

Full Turns to Approach – number of full physical turns until the approach speed is set and final positioning is attempted.

Series of wait loops (suspend and wait) are implemented in the positioning RTOS task of the service. Each of these secure the completion of a particular section of the drive profile - speed-up and revolve, slow-down, final approach, position and stop. Once the position is reached the motor will be held still, supplied with a minimum current to warrant a hold moment/sufficient torque for the load.

The current firmware implementation is tested with various scenarios and performs quite well in all of these. Fine tuning for border cases, especially low number of turns or fractions may be needed.

The drive profile calculation is based on some parameters that very much influence the time to position and its precisions:

MIN_APROACH_SPEED, MAX_APROACH_SPEED and **APROACH_SPEED_FACTOR** – configure and tune the actually used approach speed, specifying the minimum, maximum and the reduction factor.

SMOOTH_ACC_UP_FACTOR, MIN_SMOOTH_ACC_UP and **DEFAULT_SMOOTH_ACC_UP** – control of the acceleration and de-acceleration rate. Currently a symmetric one is used. Separate de-acceleration factor may be introduced as it will allow further increase of the precision and faster operation.

LOW_SPEED_TURNS_FACTOR and **DEFAULT_LOW_SPEED_TURNS** – define the safety margin for the final approach in terms of physical turns.

The drive profile may be further divided into more sections, especially for the smooth and controlled position approaching. The current implementation is illustrating the fundamental approach that may be taken. The current configuration reflects an experimental set-up that was found to be a good compromise between speed and precision.

The macro **ADVANCED_TUNR_CTRL_DEBUG** can be enabled in the file scope level. It will allow in combination with the semi-hosting (Terminal IO) printing of the values at important navigation points as depicted on Figure 4-10.

The following set-up was used to validate and or tune the parameters of the service:

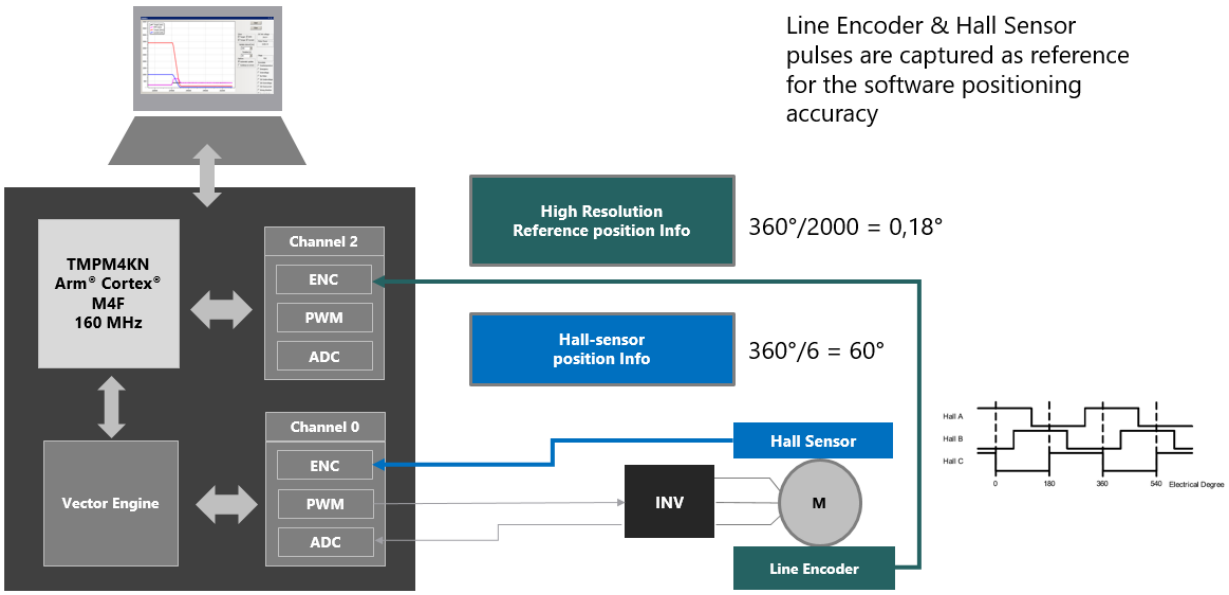

Figure 4.12 Utilizing MCU Motor Studio and sensor to validate the Advanced Software Positioning precision

A single pole-pair motor equipped with hall sensor and line encoder was connected to channel 0 of the TMPM4KN reference model. The hall sensor inputs were fed to the Advanced Encoder input of channel 0, the line encoder was connected to the input of the Advanced Encoder input of channel 2. These were configured to count the number of pulses as low and high resolution reference respectively. The number of pole pairs was kept to 1 to have direct mapping between electrical and physical turns. In the end of each positioning command the software calculated position was compared with the reference information.

The firmware statistics feature was modified in a way that instead of the controlled parameter (set and actual speed or torque), the q-axis current and the non-controlled parameter (actual torque or speed) were replaced by the currently calculated software position, the current hall value and the current line encode value.

4.6.3. Linear Motion Control

The Linear Motion Control is a precise positioning service, which unlike the Advanced Turn Control, requires external sensor for its operation. This is a closed-loop positioning solution that ensures fixed precision at higher system cost.

The target for the service is not set in terms of full physical turns and additional angle, but in number of sensor pulses. Therefore, the precision and partly the drive profile are strongly dependent on the used sensor.

Like the Advanced Turn Control, the service runs in separate RTOS task for each channel. A command may only be dispatched, if the motor is not running and the previous command is already completed. Abort is always possible.

Most of the drive functionality is implemented in the **motion_control.c**, complemented by the Advanced Encoder processing and interrupt handling as provided by the corresponding HAL Layer driver.

The drive control is not solely based on waiting stages. An event triggered approach via callbacks is taken. The service sets a new event as number of pulses that need to be reached. The Advanced Encoder interrupt handler is configured to invoke a dedicated routine **MotionControl_IRQ_Callback()** upon an event. This routine will configure the next one, re-engage the Advanced Encoder and notify the service task.

The drive profile is very similar to the one used in the Turn Control/Advanced Turn Control service. However, there is an additional factor that is to be considered. It is the mapping between the Advanced Encoder counter's value and rotor's position. The firmware differentiates between:

Absolute position – it is the one to one representation of the Advanced Encoder pulse counter. The position is set to 0 at the beginning, thus negative values would indicate rotation in CCW direction. The absolute position may be reset during the normal operation, reverting the value of both the Advanced Encoder pulse counter and the internal state variable.

Relative position – a fixed offset to the absolute position. A pre-configured start value is mapped to the absolute position zero.

Unlike the Advanced Turn Control, the Linear Motion Control allows limit definitions – minimal and maximal relative positions that may not be exceeded in either direction.

The relative position configuration is performed in the individual motor configuration file via the **MOTOR_ENCODER_MIN_COUNT, MOTOR_ENCODER_MAX_COUNT** and **MOTOR_ENCODER_START_COUNT** macros. These fields are mandatory even if the service is disabled.

No configuration parameters for precise tuning of the drive profile, like the ones in Advanced Turn Control (approach speed factor and minimums, smooth acceleration and low turn factor), are available at present. As the drive profile and section split is identical, such can be easily added if the application requires dynamic adjustment.

The simultaneous usage of the Advanced Turn Control & Linear Motion Control is not supported, although possible as long as these are not used for control of one and the same channel. Further limitation may be the amount of RAM memory that is needed for the individual task.

4.6.4. Stall Detector

The simple service is implemented in **stall_detector.c** is executed in own RTOS task, supervising the d-axis voltage drops below certain configurable threshold.

Whenever a stall is detected, the motor will be stopped in short-brake mode. After a decent wait time (hardcoded to 500ms) and only if the system recover mode configurable via **SYSTEM_RESTART_MODE** is enabled a restart with the previously set speed/torque will be attempted.

4.6.5. DSO

Digital Storage Oscilloscope (DSO) is a service allowing logging of various system, motor and control parameter during normal operation without stopping the system. It is relatively less intrusive. It is enabled in the top-level user configuration via the **USE_DSO** macro, which is the firmware default.

Various parameters will be sampled at the end of control cycle in the interrupt loop or in the channel's PMD interrupts and will be latched in a dedicated buffer using the **DSO_Log()** function. The configuration of the number (up to 8) and exact parameters is performed via the **ConfigDsoLog()** that is mapped to a communication protocol command, but can also be called directly which is not the typical usage.

The collected data can be retrieved via the **GetDsoLogData()** which is directly mapped to a communication protocol command, although it may be used internally as well.

The size of the buffer and the resulting maximum capacity is set by the **BOARD_DSO_SIZE** define in the top-level configuration file **config.h**.

After data collection completion, it may be retrieved or a new collection can be started overwriting the existing data. The exchange of the collected data is done via the URAT used for serial communication protocol.

The implementation is done in the file pair **dso.c/dso.h**.

DSO can be also used to log any user defined parameter (of type signed short) in the firmware . Replace **user_dummy** with user defined parameter which needs to be logged in function **DSO_Fill_up_dataVE()** if hardware VE is used or **DSO_Fill_up_dataSWFOC()** if software VE is used.

```
LOG_IF_SELECTED(user_dummy,                    User_1,     0, 0        );
LOG_IF_SELECTED(user_dummy,                    User_2,     0, 0        );
```

Figure 4.13 Use of DSO to log user defined variable

4.6.6. HS-DSO (in future releases)

The High-Speed Digital Storage Oscilloscope (HS-DSO) is a variant of the DSO, where the data is collected at transferred "real-time" using separate UART channel and much higher baud rates. The UART selection is done via the **HSDSO_COMMUNICATION_UART_CHANNEL** macro in the individual board specific configuration file. Additionally, a high-speed capable cable/adapter (FTDI C232HD-EDHSP-0) and host system driver is needed for the flawless operation. The feature is disabled by default and can be compiled in using the **USE_HSDSO** macro in the top-level user configuration file **user_config.h**.

The implementation is done in the C/H file pair **hsdso.c** and **hsdso.h**.

4.6.7. Performance Measurement

The execution of important core functions and or interrupt handlers might be time critical or could requires significant time. In order to understand the dynamics and the limits of the system, as well as to optimize or properly configure the interrupt priorities an intrusive performance measurement concept was introduced. It is based on a high-resolution free running counter that is captured at various execution points, mostly at entry or exit of a function. The recorded timer values and calculated difference are stored in log buffers and overridden at the next capture. The following general fields are available for several interrupt handlers and/or core functions:

Start Time – counter value captured at entry of the function

End Time – counter value capture upon exit of the function

Processing Time – the difference between the start and end time

Occurrence Rate – the difference between two successive Start Times

The resolution is 6.25 ns, as the firmware utilizes the 32-bit free running counter of the Debug Watch and Trace (DWT) module running at 160MHz. the implementation can be found in **performance_measurement.c**.

4.6.8. Global Data

All shared definitions are packed in single C file **global_data.c** under the Motor Control layer. This includes system, channel, motor, encoder, regulation and operational parameters that are stored in numerous descriptor tables, defined as arrays of structure elements. Although the usage of global variables (external linkage) is generally not recommended, the memory limitation on some family members have enforced it.

4.6.9. Software Mathematical Library

The firmware features a support library containing functions for some common mathematical functions, such as trigonometry and exponentiation, as well as the motor controls specific transformations and conversions.

It is pure software implementation in a single file **(fwmathlib.c)** utilizing the CPU and its FPU for all computation. Caution shall be taken as some function use assembly intrinsic to directly compute the product of multiplication, subtraction, etc. and these may need additional effort when ported to different toolchain or environment/compiler.

The generic mathematical operations such as **pow10()** are used throughout the entire motor controller. The transformations and space modulation are solely utilized by the software replication of the vector engine as implemented in **sw_foc.c**.

The following operations are implemented as C functions: pow10, sine, cosine, Clark transformation, Park transformation, Inverted Park transformation, Space Vector Modulation, Software Limiter. Helper functions for minimum, maximum and sorting of three values complement the functionality list.

4.6.10. User Callbacks

The firmware allows user specific actions for channels that are utilizing the vector engine for the input and output processing, e.g. are hardware controlled. The application can register implementation specific functions for various Tasks of Schedule 0, Schedule 1 and Schedule 9. If such is defined, it will be called back upon task completion. It introduces an easy way to influence or adjust the input parameters (scaling for example) and the computation in the single steps of the control cycle or to post process the output of a computation task. Furthermore, the user callback mechanism allows dynamic re-scheduling and/or rearranging of individual tasks or schedules, as long as the hardware implementation permits it.

The callback table is defined as an array of **UserCallback** elements, the **CallbackTable[]**. Each element in the table specifies the user defined function to be called, the next schedule to be executed, the next task in the schedule that is to be started. Current definition and the supported callbacks are illustrated below:

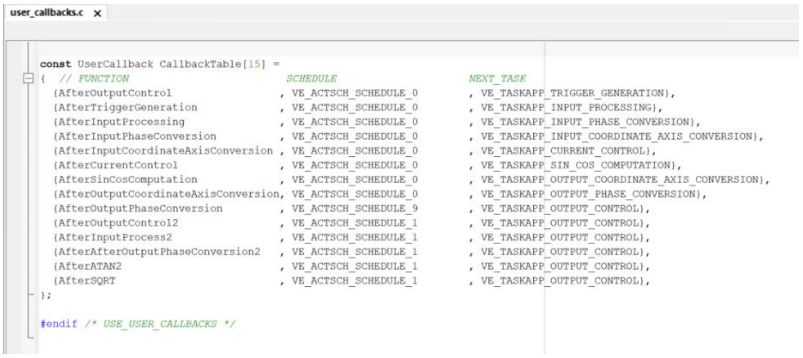

Figure 4.14 Supported user callback functions

As the implementation is not fully configurable, please do not change or rearrange the table, especially the first, second and last entries.

An example usage would be to alter the next task value, jumping over a single task in the schedule for which the computation was already done (in software). Another typical usage is to change from Schedule 0 to Schedule 1, allowing automatic execution of the remaining sequence.

The entire implementation is captured in single C/H file-pair **user_callbacks.c/user_callbacks.h** and is configured in via the **USE_USER_CALLBACKS** macro in the top-level user configuration file **user_config.h**.

Please refer to the Vector Engine's Reference Manual, Chapter "Schedule Management" for further details on the scheduling and tasks.

4.6.11. Watchdog Usage (in future releases)

The firmware implements configurable usage of the built-in watchdog timer for supervision and system recovery upon malfunction or deadlocks. Disabled by default, it can be configured in by uncommenting the **USE_WDT** macro in the top-level user configuration file **user_config.h**.

The initialization is performed by the **BOARD_SetupWDT()** function defined in **board_system_setup.c**. The configuration parameters are defined with the structure **configWDT**. The time-out value may need adjustment depending on the selected features and number of channels configured in.

The clearance is done in the overridden FreeRTOS idle handler **vApplicationIdleHook()**.

5. Configuration

There are different approaches for the initial set-up. The one described here is the recommended and proven in our experience rich practice.

Regardless of the selected approach, all of the system components described in this chapter shall be configured or at least checked for consistency.

A back-up of the project or any reused and/or modified file shall be kept to allow recovery and/or usage of any predefined standard project/configuration.

External power supplies with current and voltage limiters shall be used during the initial set-up and configuration, in order to avoid short currents, hardware damages or person injuries!

5.1. Configuration Concept

The firmware relies of numerous header files for specifying the used features, channels, boards, their configuration parameters and interconnections. There are two major groups:

Global configuration files – those describe and/or configure generic characteristic of the firmware, the RTOS, the used boards/motors and the standalone application (if applicable). Three major files fall in this group – **config.h, user_config.h** and **standalone_config.h**. All these are defined on top project level.

Individual (usage specific) configuration files – these provide definitions and configurations for particular type of board, channel, motor or external component. Additionally, these may be further modified or cloned to meet the specific requirements of individual board, motor or component instance of a given type. An example would be a generic motor definition that is further refined for a motor variant with gearbox or a variant with different reduction ratio of the gearbox, etc. These files are specified in the corresponding project folders: **~\MotorControl\Source\Motors\Board\, ~\MotorControl\Source\Motors**, etc.

C/C++ preprocessor macros are defined on project level for the few specific cases where header file inclusion is not applicable, as it might result circular dependencies.

5.2. Initial Project Set-up

5.2.1. Project Configuration & Components

The Firmware comes with predefined project configuration for every member of the various TOSHIBA Motor Control MCU families (TMPM4K). It is configured for usage with the default board and motors used for that particular MCU, based on the available and commonly used Evaluation Kit, Reference Model or Partner board.

It is recommended to start with a direct reuse of the debug configuration for the selected MCU. Alternatively, a copy of that configuration can be made. The exact procedure is very much tool dependent. Under IAR Embedded Workbench it is done via **Project->Edit configurations** … as illustrated below:

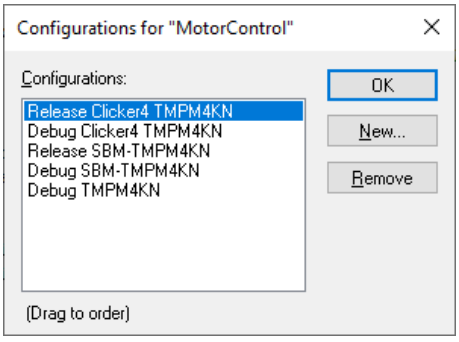

Figure 5.1 New/clone project configuration

5.2.2. Board Configuration

Numerous board, sensor, shield, click-on board and BLDC motor configuration files, all tested with at least one of the MCU's platforms, are integral part of the package. Re-usage of these components either "as-is" or as template/copy is highly encouraged.

5.2.2.1. Base (main) Board Configuration

The used main board shall be defined as macro in the C/C++ Compiler pre-processor definitions, for illustrative purposes the "BOARD_USERGUIDE_M4KN" is introduced:

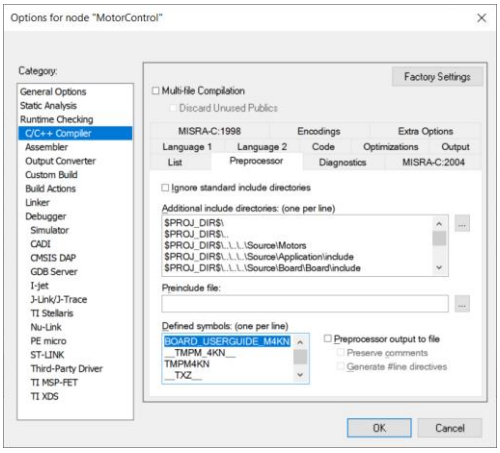

Figure 5.2 Adding new board configuration

The board and its configuration has to be made known throughout the project, adding specific definitions in the **config.h**. The following additional parameters shall be set:

BOARD_NAME – string to identify the board, mainly used in the communication protocol. Human readable name, board ID, part number or nick name is preferable. Limited to 20 characters.

BOARD_CONFIG_HEADER_FILE – specifies the header file describing the required board parameters, detailed description of these will follow later on

.

BOARD_PWR_HEADER_FILE_0/ BOARD_PWR_HEADER_FILE_1/ BOARD_PWR_HEADER_FILE_2 - specifies the header file describing the required power board (output stage) parameters for the corresponding channel. Simply omit the definition for the unused ones. It is possible to define/use different power stages for the different channels.

An example may look like (channel 0 & channel 2 used only):

```
/* An example */
#ifdef BOARD_USERGUIDE_M4KN
#define BOARD_NAME               "USERGUIDE_M4KN"                    /* Example Board used in the User's Guide */
#define BOARD_CONFIG_HEADER_FILE "board_config_sbk_m4kn.h"           /* Re-use existing one, make sure to make a back-up copy of the original fi

#ifndef BOARD_PWR_HEADER_FILE_0
#define BOARD_PWR_HEADER_FILE_0  "example_m4kn_pwr.h"                /*! Powerboard to be used */
#endif /* BOARD_PWR_HEADER_FILE_0 */

#ifndef BOARD_PWR_HEADER_FILE_2
#define BOARD_PWR_HEADER_FILE_2  "example_m4kn_pwr.h"                /*! Powerboard to be used */
#endif /* BOARD_PWR_HEADER_FILE_2 */

#endif /* BOARD_USERGUIDE_M4KN */
```

Figure 5.3 Example board configuration entry in config.h

Unless already existing, the board specific configuration file shall be put to the board folder of the package under **~\MotorControl\Source\Board\Board\config**. The **board_config_clicker4_m4kn.h** file in the example configuration is already existing as illustrated below:

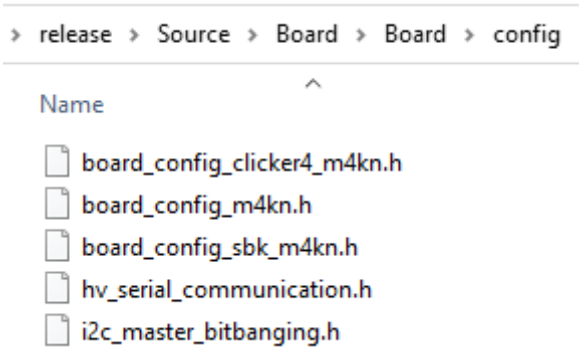

Figure 5.4 Base board specific configuration file location

The following parameters need to be specified in each base board specific configuration file, e.g. board_config_clicker4_m4kn.h in our example:

BOARD_USE_EXTERNAL_OSCILLATOR – specifies whether the internal high speed oscillator or an external one is used as main clock supply.

BOARD_EXTERNAL_OSCILLATOR_FREQUENCY – the frequency of the external oscillator in Hz if used.

BOARD_USE_PLL – usage of the internal clock multiplication circuit (PLL) for speeds up to 160 MHz.

BOARD_GAIN_CURRENT_MEASURE – a configurable current measurement gain may be applied. The parameter is used as index to select an option from the gain table **gaintable[]**, which must contain at least one valid value. The selected gain is later on used in calculating the maximal values for the current and voltage sensing.

AINx_3PHASE_U / AINx_3PHASE_V / AINx_3PHASE_W – specify the analog input ports for the phase voltage (current) measurement of each channel. Definition shall be provided for all supported channels, even unused. In that case it is allowed to reuse the inputs of the used channel(s) or any unused one.

AINx_VDC – the analog input port for the DC link voltage measurement of each channel/power board. Definition shall be provided for all supported channels, even unused. In that case it is allowed to reuse the input of the used channel(s) or any unused one.

VDC_MEASUREx_REG – specify the ADC register to be assigned for DC link voltage measurement results. The values here will be used in the PWM duty calculation for output control and the “Software Over/Undervoltage Detection” if configured-in (via **user_config.h**).

AINx_TEMPERATURE – the analog input port for the temperature measurement if supported and configured in the individual power board configuration header file. Definition shall be provided for all supported channels, even unused. In that case it is allowed to reuse the input of the used channel(s) or any unused one.

TEMPERATURE_ADCx / TEMPERATURE_REGx – ADC unit and result register for temperature control of each channel. No definition is needed for the unused channels

SERIAL_COMMUNICATION_UART_CHANNEL – UART channel for the communication protocol that might be used for external application control, interaction with MCU Motor Studio, etc.

HSDSO_COMMUNICATION_UART_CHANNEL – UART channel for the HS-DSO data exchange if configured-in (via **user_config.h**).

There are several LED related definitions, which need valid definition, only if one of the **USE_LED / USE_RGB_LED** configuration option is enabled in **user_config.h**. In that case either **led.c** or any other

relevant implementation has to be included in the build.

Finally any board specific API declarations or exported symbols may be defined, besides the mandatory **BOARD_Detect_Revision()** which is used in the communication protocol.

For every base board an implementation has to be provided under **~\MotorControl\Source\Board\Board**. It is advisory that both the configuration and implementation file names are same, for example **board_clicker4_m4kn.c** and **board_clicker4_m4kn.h**.

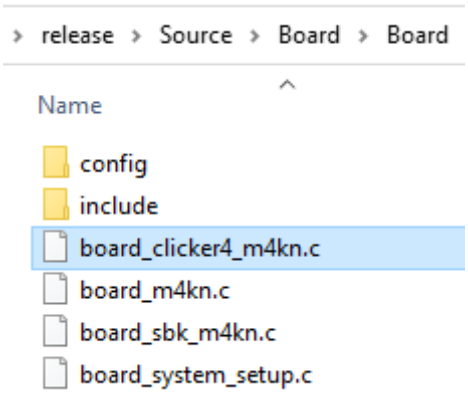

Figure 5.5 Base board specific implementation file location

5.2.2.2. Power Board (stage) Configuration

For each power board defined under **BOARD_PWR_HEADER_FILE_x**, a configuration file shall be created or copied to the power board folder of the package, e.g. **~\MotorControl\Source\Board\Power**. One definition file may be reused for multiple channels, given one and the same is used for these. No implementation file is required, although such may be defined and included in the build if needed.

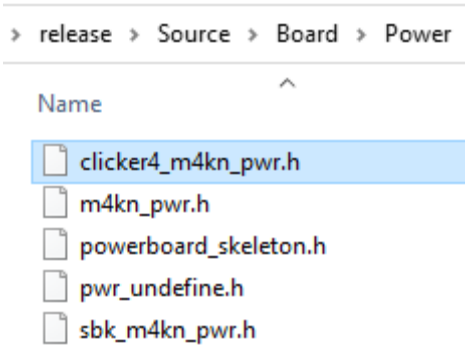

Figure 5.6 Power board specific configuration file location

The following parameters need to be specified:

USE_EMERGENCY_SIGNAL – commented out by default, this definition specifies that a low active emergency signal is available on the power board and can be fed to the corresponding EMGx input

USE_OVERVOLTAGE_SIGNAL – commented out by default, this definition specifies that a low active overvoltage signal is available on the power board and can be fed to the corresponding OVVx input

Both signals are handled with highest priority. If detected, the PMD will automatically and immediately disable all 6 PWM outputs. Recovery is possible by following a special EMG/OVV return procedure. which is currently not implemented/supported in the Firmware.

BOARD_NAME_PWR – string to identify the power board, used in the communication protocol. Human readable name, board ID, part number, nick name or similar is preferable. Limited to 20 characters.

BOARD_DEAD_TIME – The minimal wait time in ns, needed to ensure that the MOSFETs are switched off. It is advisable to start with conservative timings, far above the typical value given in the transistor specification. Few hundreds of ns up to one thousand is normally good initial value. The final one shall still secure sufficient margin for proper operation under all temperature conditions and voltage/frequency fluctuation.

BOARD_BOOTSTRAP_DELAY – the time needed to fully charge the bootstrap capacitors through the low-side MOSFETs of the H-bridge. It is recommended to use conservative timing and allow sufficient margin to ensure proper operation under all conditions.

BOARD_SENSITIVITY_CURRENT_MEASURE_VALUE – sensitivity of the current measurement circuit. MCU Motor Studio features simple calculator for a typical circuit as depicted below:

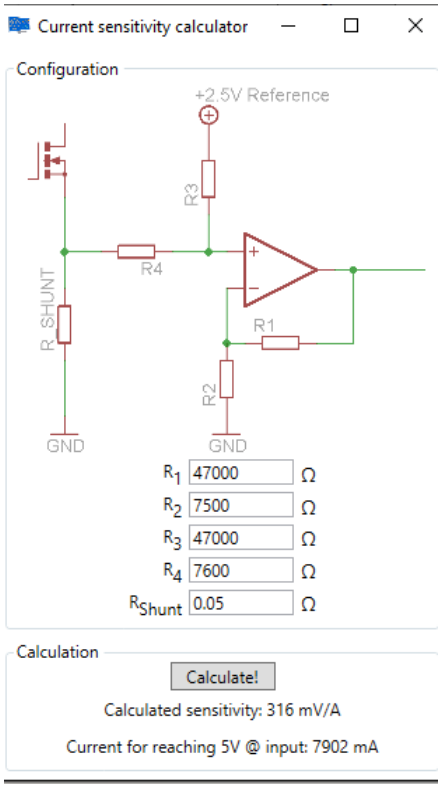

Figure 5.7 Current sensitivity calculator

BOARD_SENSITIVITY_CURRENT_MEASURE_UNIT – unit of the measurement sensitivity, normally mV/A

BOARD_MEASUREMENT_TYPE – select one of the supported types, defined as CURRENT_MEASUREMENT enumeration in api.h. The following values are possible at present: CURRENT_SHUNT_1, CURRENT_SHUNT_3 and CURRENT_SENSOR_2. Any other symbolic or numerical definition will lead to run-time assertion. Edit the value in "**clicker4_m4kn_pwr.h**".

```
#define BOARD_MEASUREMENT_TYPE                  CURRENT_SHUNT_3                 /* [NONE]                                                */
#define BOARD_CURRENT_MEASURE_DIRECTION         CURRENT_MEASUREMENT_INVERTED
#define BOARD_SENSITIVITY_VOLTAGE_MEASURE_VALUE  77                             /*              - Sensivity of voltage measurement circuit */
```

Figure 5.8 Board Measurement type

BOARD_CURRENT_MEASURE_DIRECTION – select the current sensing type. The two supported values **CURRENT_MEASUREMENT_NORMAL** and **CURRENT_MEASUREMENT_INVERTED** are defined in **api.h**, **CURRENT_MEASUREMENT_ORIENTATION** enumeration. Any other symbolic or numerical definition will lead to run-time assertion.

BOARD_SENSITIVITY_VOLTAGE_MEASURE_VALUE – sensitivity of the voltage measurement circuit. MCU Motor Studio features simple calculator for a typical circuit as depicted below:

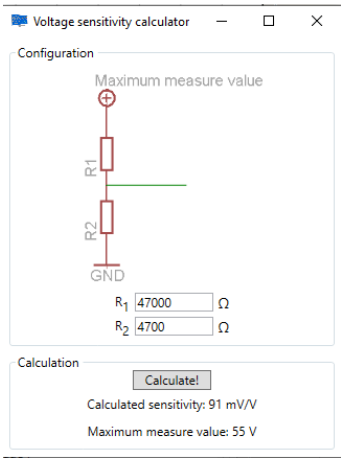

Figure 5.9 Voltage sensitivity calculator

BOARD_SENSITIVITY_VOLTAGE_MEASURE_UNIT – unit of the measurement sensitivity, normally mV/V.

BOARD_POLL / BOARD_POLH – the active level of the low-side / high-side MOSFETs respectively.

BOARD_VDC_CHANNEL_x_VALUE / BOARD_VDC_CHANNEL_x_UNIT – optional, typically used for board bring-up, troubleshooting of control issues or whenever there are instabilities in the voltage measurement. If defined, the voltage measurement results are overridden with the specified fixed value. Please note that no DC link voltage ripple compensation can be done with such clamping. These definitions are commented out be default.

5.2.2.3. On-board Temperature Sensor Configuration

The temperature sensing is configured on power board level, as the sensors are normally located there. No restriction or requirement related to the channels is enforced. Mixing power boards with and without temperature sensing or not using an available sensor is allowed. Using different sensors within one project is however not yet supported. It requires minimal customization of the firmware a.

Temperature monitoring and over temperature protection will be automatically performed for each channel with temperature sensing. This behavior is not configurable.

The following minimal configuration has to be done on power board level, e.g. in the header file specified under **BOARD_PWR_HEADER_FILE_x**:

USE_TEMPERATURE_CONTROL – enable the temperature sensing for that particular power board and the channel it is connected to

TEMP_SLOPE – the slope for the over temperature protection recovery in degree Celsius

XXXXXX (TDK_NTCG163JF103FT1) – user selectable definition for the type of the sensor used. It shall be used for temperature table definitions and any specific or additionally required implementation, extending the generic temperature control.

The generic handling of temperature measurement and over-temperature protection is located under **~\MotorControl\Source\Board\Temperature** and the temperature folder of the workspace as illustrated bellow:

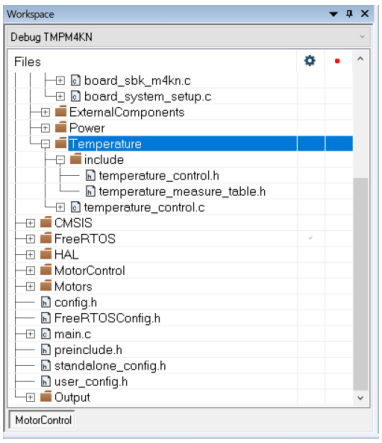

Figure 5.10 Temperature project structure

The measured voltage to temperature relation is captured in a table, which is sensor type specific and located in **~\MotorControl\Source\Board\Temperature\include\temperature_measure_table.h**. Only one table shall exist pro sensor and project configuration. The typical resolution is 5 ℃, but may be précised by simply extending the table and strongly depending on the capabilities of the sensor. In the below example the resolution is set to 10 ℃. The first value is the value that will be read from the sensor, when the temperature is as specified by the second. The firmware automatically interpolates the actually read value to the closest specified one:

```
#ifdef TDK_NTCG163JF103FT1
static const temp_table temperature[]={
                                    {0xE48, -20},
                                    {0xCFB, -10},
                                    {0xB8E,   0},
                                    {0x9F6,  10},
                                    {0x862,  20},
                                    {0x6DB,  30},
                                    {0x58B,  40},
                                    {0x468,  50},
                                    {0x380,  60},
                                    {0x2C7,  70},
                                    {0x235,  80},
                                    {0x1B7,  90},
                                    {0x159, 100}
                                   };
#endif /* TDK_NTCG163JF103FT1 */
```

Figure 5.11 Example entry in temperature_measure_table.h

5.2.2.4. External Components Configuration

Various external components (others then temperature sensing elements/circuits) are defined and used in combination with some of the base or power boards. Those include keypads, led drivers, programmable gain amplifiers, etc.

There is a project group in dedicated folder, following the same logic an structure as the base board.

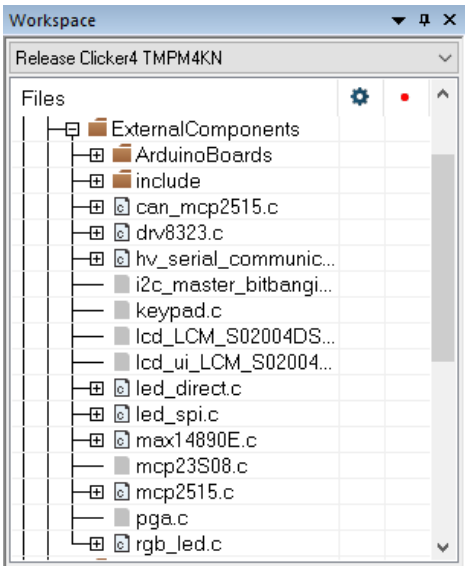

Figure 5.12 External components project structure

All needed definitions shall be placed under **~\MotorControl\Source\Board\ExternalComponents\include** or a component specific definition sub-folder like the depicted **AdruinoBoards**. In the latest case the project configuration shall be updated with the newly introduced additional include directories. It is recommended to add the header files to the tool-chain workspace viewer.

The implementation file(s) shall be copied to **~\MotorControl\Source\Board\ExternalComponents** and included in the project. The needed initialization shall be invoked from the generic **board_system_setup.c**.

5.2.2.5. Board Build-related Configuration

All relevant board, power board and external components implementations have to be included in the build.

The definitions and implementation files of all supported standard platforms/MCUs are included in the build by default and for completeness. The unused symbols and code sections will be removed during the linkage process.

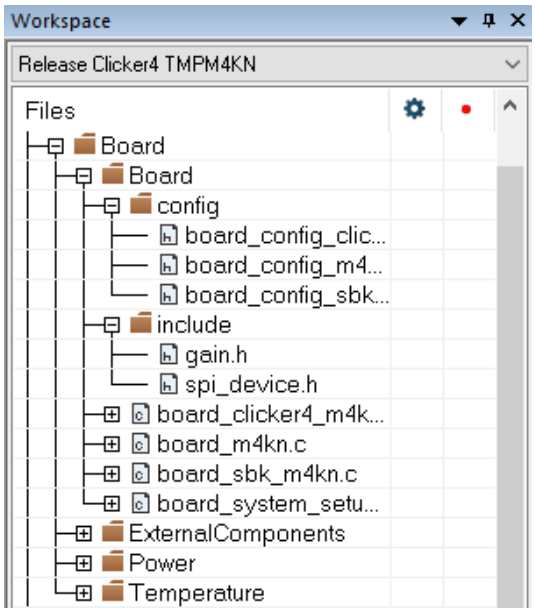

Figure 5.13 Board & components build

Excluding the unused files and configurations for the customized build and/or application is highly recommended.

5.2.3. Channel & Used Features Configuration

As a next step the configurations file **user_config.h** shall be adjusted to the number of channels and motors used. For each channel a configuration file containing the motor specific definitions shall be introduced, assigning it to the macros **MOTOR_CHANNEL_0, MOTOR_CHANNEL_1** and **MOTOR_CHANNEL_2** respectively. If the motors and their configuration are same for several channels, then one single header file can be used for these. If a channel is not to be used, simply do not specify any include file for it. (Note: only MOTOR_CHANNEL_0 is supported**)**

The legacy parameter **MOTOR_CHANNEL_FOR_STORAGE** specifies the single channel whose parameters shall be stored in the NVM, whenever the option is configured-in and there is no sufficient space to store all configured channels.

Given a standalone application (in future releases) is used, all channel related definitions shall be done in **standalone_config.h**. The same principles apply.

```
//#define MOTOR_CHANNEL_0 "motor_define_ACT_42BLF01_1sh_opa.h"          /*!< Motor to be used */
//#define MOTOR_CHANNEL_0 "motor_define_ACT_42BLF01_1sh_ti.h"           /*!< Motor to be used */
#define MOTOR_CHANNEL_0 "motor_define_ACT_42BLF01_3sh_opa.h"          /*!< Motor to be used */
//#define MOTOR_CHANNEL_0 "motor_define_DB59S024035-B_Joint2.h"         /*!< Motor to be used */
//#define MOTOR_CHANNEL_0 "motor_define_Dematek_BL4220B.h"              /*!< Motor to be used */
```

Figure 5.14 Typical channel definition

Besides channel definition, **user_config.h** is used to configure the set of features that shall be supported by the particular build of the firmware:

VE_CONTROL_LOOP_FREQUENCY - specifies the frequency of the control loop. It shall be between 1kHz and 10kHz. Typical & recommended value is 1kHz.

USE_WDT - use the internal Watchdog for system monitoring, commented out by default.

USE_OFD - make use of Oscillation Frequency Detector, commented out by default.

USE_INTERNAL_MOTOR_PARAMS - use the compiled-in motor parameters, default option.

USE_LED / USE_RGB_LED - activates the LED signaling for various status indicators, disabled by default.

USE_SERIAL_COMMUNICATION – enable the communication protocol via the selected UART, enabled by default.

USE_CONFIG_STORAGE – enable if the parameters have to be retrieved from the non-volatile memory, instead of the default and typically present compiled-in ones. The type of memory and the access functions are dependent on the below listed two definitions. One of these shall always be defined, if configuration storage is to be used. Otherwise the firmware will not be compliable. The feature may not be available in all its variants on all platforms and MCUs.

USE_CONFIG_STORAGE_EEPROM / USE_CONFIG_STORAGE_FLASH – the parameters are stored in EEPROM / Flash. These two options are mutual exclusive.

USE_DSO – up to 8 parameters can be logged during run-time and retrieved via the communication protocol. The define enables the collection of data and is on by default.

USE_HSDSO – disabled by default, this is a high-speed option for data collection, which is exchanged via dedicated and separately configurable UART channel and requires special high-speed UART adapter and PC driver for proper operation. At present available with MCU Motor Studio only.

USE_LOAD_STATISTICS – enabled by default this option is used to estimate the current CPU load based on “idle load” values that were measured by the firmware in a “special” mode/configuration – compiled with no features, not driving any motors and having the idle task active only.

MEASURE_MOTOR_CONTROL_PROCESS_TIME – intrusive option that captures the actual 32-bit free running counter value of the Debug Watch and Trace (DWT) module to log the start/end time and occurrence rate of various core functions and IRQ handlers. Disabled by default, it shall be used with caution and avoided in normal operation.

USE_ENCODER – using the advanced encoder counters to capture the input of speed sensors attached to the motor. It may be overridden by the motor configuration field “encoder usage”, thus it is enabled by default and may be kept in this state.

USE_STALL_DETECT – activates the **Vqi** monitoring for stall detection in FOC mode, regardless whether encoder is used or the firmware runs sensor-less. The drop/raise rate of **Vqi** may be measured with the help of the DSO for each particular motor and the implementation in **stall_detect.c** may be refined, at least the fall/raise factor as defined in **JUMP_STALL_DETECT_PERCENTAGE** may need to be adjusted. It defaults to 70%

USE_SW_OVER_UNDER_VOLTAGE_DETECTION – the firmware shall monitor the values produced by the DC link voltage measurements and enforce protective actions, instead of relying to the external overvoltage signal.

USE_MOTOR_DISCONNECT_DETECTION – monitor the measured **Ia, Ib, Ic** to determine the presence of a motor on each of the available channels.

USE_LOAD_DEPENDANT_SPEED_REDUCTION – **Iq** monitoring and speed reduction upon threshold exceeding detection. The threshold is pre-calculated based on the maximal **Iq** normed to Ampers and the desired speed reduction ratio (percentage).

USE_SW_OVERCURRENT_DETECTION – the firmware shall monitor the square power of the direct and torque component and enter emergency handling, if a certain threshold is reached, e.g. $(\mathbf{Id}*\mathbf{Vd})^2 + (\mathbf{Iq}*\mathbf{Vq})^2 <$ THRESHOLD. The threshold value is pre-calculated upon system initialization, based on the characteristics of the used boards and motors.

USE_CAN – enable the CAN communication task and operation with the MCP2515. Available only for selected platforms, may be further extended.

USE_TURN_CONTROL – Turn Control / Advanced Turn Control features shall be included in the build. Encoder usage parameters in the motor configuration has to be adjusted accordingly, e.g. **MOTOR_ENCODER_TYPE** shall be set to **4** for the Advanced Turn Control.

USE_MOTION_CONTROL – compile-in the Linear Motion feature. Encoder usage parameters in the motor configuration has to be adjusted accordingly, e.g. **MOTOR_ENCODER_USAGE** shall be set to **4**, “Event IRQ counting”.

USE_TORQUE_CONTROL – enables the control by torque as optional drive method, default option.

USE_USER_CALLBACKS – override the automatic Vector Engine handling. User has to register/provide implementation of the callback functions, as defined in **user_callbacks.c**. These shall perform the desired handling of each scheduled VE step – input processing, input phase conversion, etc.

USE_EXTERNAL_SPEED_CONTROL – allows basic control of speed and direction via user-defined GPIOs for selected legacy platforms. Depreciated in favor of the far superior control offered by MCU Motor Studio.

5.3. Motor Parameter Configuration

Number of BLDC motor characteristics need to be defined for each motor currently configured in the build via the **MOTOR_CHANNEL_x** macro. Typically, these parameters can be found in the datasheet,

some like the inductance may be measured as well.

MOTOR_POLE_PAIRS – number of poles divided by 2.

MOTOR_DIRECTION – specifies the directions the motor can revolve in.one of CW and CCW. One of **MOTOR_CW_ONLY, MOTOR_CCW_ONLY, MOTOR_CW_CCW** or their numeric representation shall be used. The enumeration **MOTOR_DIRECTION** is defined in **api.h**.

MOTOR_ANGULAR_ACC_MAX_VALUE / MOTOR_ANGULAR_ACC_MAX_UNIT – maximal angular acceleration and unit, typically rad/s^2.

MOTOR_TORQUE_FACTOR_VALUE / MOTOR_TORQUE_FACTOR_UNIT – torque constant, typically in mNm/A.

MOTOR_RESISTANCE_VALUE / MOTOR_RESISTANCE_UNIT – motor winding resistance.

MOTOR_INDUCTANCE_VALUE / MOTOR_INDUCTANCE_UNIT – motor windings inductance.

MOTOR_SPEED_LIMIT_VALUE / MOTOR_SPEED_LIMIT_UNIT – maximal/rated angular speed.

MOTOR_SPEED_CHANGE_VALUE / MOTOR_SPEED_CHANGE_UNIT – minimal speed at which FOC may be entered, e.g. the measured back-EMF is sufficient for filed-oriented control. This value needs to be optimized or adjusted and is not specified by the datasheet.

MOTOR_POSITION_DELAY_VALUE / MOTOR_POSITION_DELAY_UNIT – wait time until the rotor is settled in its initial position. This value is typically given in ms and is not specified in the datasheet.

MOTOR_ID_START_VALUE / MOTOR_ID_START_UNIT – Value of the direct component. Typically kept as 0mA.

MOTOR_IQ_START_VALUE / MOTOR_IQ_START_UNIT – start current/current to be applied during the forced commutation phase, typically in mA.

MOTOR_IQ_LIM_VALUE / MOTOR_IQ_LIM_UNIT – maximal allowed/rated value of the direct current component.

MOTOR_ID_LIM_VALUE / MOTOR_ID_LIM_UNIT – maximal allowed/rated value for the quadrature component of the current.

MOTORID – string to identify the power board, used in the communication protocol. Human readable name, board ID, part number, nick name or similar is preferable. Maximal 20 characters.

5.4. Encoder Configuration

The proper usage of the Advanced Encoder Input Circuit requires various parameters of the external sensor that is connected to the motor (if applicable). These are all motor specific and therefore were made part of the motor configuration header files, as specified by the **MOTOR_CHANNEL_x** macro:

MOTOR_ENCODER_TYPE – one of the values defined in **api.h** under **ENCODER_TYPE** or its numeric representation shall be used. Selectable are single pulse, hall sensors with 2 and 3 pulses, encoders with and without index (Z) pulse and the AMS AS5145H.

MOTOR_ENCODER_USAGE – specify the intended usage among none, speed detection, event counting and interrupt on “at position”. The **ENCODER_USAGE** enumeration is defined in **api.h**.

MOTOR_ENCODER_COUNT – resolution of the sensor attached in number of pulses generated per turn, typical 6 for hall sensors.

MOTOR_ENCODER_MIN_COUNT – value used in the standalone applications and/or in the Linear Motion to define the lower border. It shall be specified in number of pulses.

MOTOR_ENCODER_START_COUNT – a relative value mapped to the initial value of the Advanced Encoder counter.

MOTOR_ENCODER_MAX_COUNT – value used in the standalone applications and/or in the Linear Motion to define the upper border. It shall be specified in number of pulses.

MOTOR_ENCODER_LM_APPROACH_RPM – Linear Motion specific parameter, indicating the maximal approach speed given in RPM.

MOTOR_ENCODER_RECEIVER – the type of receiver connected between the used encoder and the MCU. Possible values are single ended, differential, resolver, none. These as defined in **api.h**, **ENCODER_RECEIVER** enumeration.

MOTOR_ENCODER_RECEIVER_MODE – select the operation mode for the signals (A, B and Z) of the differential encoder between TTL and RS-422.

MOTOR_GEAR_FACTOR – the reduction ratio of the gear attached to the motor. Shall be set to “1” if no gear is used.

5.5. PI configuration

All configuration parameters of the proportional-integral regulation are naturally part of the motor configuration header file:

CONTROL_ID_KI_VALUE / CONTROL_ID_KI_UNIT – integral coefficient for **Id** regulation in V/As unless other specified by the unit.

CONTROL_ID_KP_VALUE / CONTROL_ID_KP_UNIT – proportional coefficient for **Id** regulation in V/A unless other specified by the unit.

CONTROL_IQ_KI_VALUE / CONTROL_IQ_KI_UNIT – integral coefficient for **Iq** regulation in V/As unless other specified by the unit.

CONTROL_IQ_KP_VALUE / CONTROL_IQ_KP_UNIT – proportional coefficient for **Iq** regulation in V/A unless other specified by the unit.

CONTROL_POSITION_KI_VALUE / CONTROL_POSITION_KI_UNIT – integral gain for position control.

CONTROL_POSITION_KP_VALUE / CONTROL_POSITION_KP_UNIT – proportional gain for position control.

CONTROL_SPEED_KI_VALUE / CONTROL_SPEED_KI_UNIT – integral gain for speed control.

CONTROL_SPEED_KP_VALUE / CONTROL_SPEED_KP_UNIT – proportional gain for position control.

5.6. System Configuration

The following system related configuration are motor/channel specific and are therefore specified in the motor configuration header file:

SYSTEM_PWM_FREQUENCY_VALUE / SYSTEM_PWM_FREQUENCY_UNIT – the PWM output rate / frequency of the VE interrupt loop. Increasing the value may result FET switching loses, reducing may reduce undesirable noise.
SYSTEM_SHUTDOWN_MODE – the desired control method for the stop control. The supported values are defined in api.h, SHUTDOWN enumeration.

SYSTEM_BRAKE_TIME_VALUE / SYSTEM_BRAKE_TIME_UNIT – specifies the number of PWM output cycles until full stop.

SYSTEM_BRAKE_PERCENTAGE_VALUE / SYSTEM_BRAKE_PERCENTAGE_UNIT – currently unused legacy parameter. Shall be kept zero.

SYSTEM_RESTART_MODE – restart option in case of electromagnetic field stall. Select among switch off or attempt restart. Defined in **api.h**, **RESTART** enumeration.

SYSTEM_STALL_DETECT_VALUE / SYSTEM_STALL_DETECT_UNIT – fixed border value used for stall detection together with the field raise/drop percentage.

SYSTEM_OVERTEMPERATURE_VALUE / SYSTEM_OVERTEMPERATURE_UNIT – the minimal temperature for protection engagement.

SYSTEM_SPEED_CONTROL_MODE – applicable for external speed control only, specifies the input mode among ADC, PWM, “0 to max” or “–max to + max”.

SYSTEM_SW_OVERVOLTAGE_VALUE / SYSTEM_SW_OVERVOLTAGE_UNIT – only used with software overvoltage detection, specifying the threshold

SYSTEM_SW_UNDERVOLTAGE_VALUE / SYSTEM_SW_UNDERVOLTAGE_UNIT – define the threshold for the software enforced under voltage protection

SYSTEM_SW_OVERCURRENT_VALUE / SYSTEM_SW_OVERCURRENT_UNIT – only used with software overcurrent protection, specifying the threshold.

SYSTEM_SPEED_REDUCTION_VALUE / SYSTEM_SPEED_REDUCTION_UNIT – specify the reduction in terms of percentage form the set target speed.

5.7. Board Configuration via MCU Motor Studio

The board configuration is static and marked read-only so that the no changes are permitted via the UART Command Interface. Sometimes it is necessary to adjust and experiment with these. The pre-processor definitions **USE_RW_BOARD_SETTINGS** can be used to temporarily allow changes via the command interface/MCU Motor Studio. It is highly advisable to disable it again, once the proper settings are found and fixed, simply keep it defined with a prefix **no**, e.g. **noUSE_RW_BOARD_SETTINGS**, as illustrated below:

5.8. Quick reference for Clicker 4 Board

The below table gives a quick reference to software changes and applicable hardware settings to be considered while checking a functionality.

Sl.no	Functionality	Software	Hardware
1	Build setting - Configuring the set of features that shall be supported by the particular build of the firmware	File : user_config.h User can define the features to be supported. #define USE_INTERNAL_MOTOR_PARAMS #define USE_SERIAL_COMMUNICATION #define USE_DSO #define USE_LOAD_STATISTICS #define USE_ENCODER #define USE_STALL_DETECT #define USE_TURN_CONTROL #define USE_MOTION_CONTROL #define USE_TORQUE_CONTROL #define USE_SW_OVER_UNDER_VOLTAGE_DETECTION #define USE_MOTOR_DISCONNECT_DETECTION #define USE_LOAD_DEPENDANT_SPEED_REDUCTION #define USE_SW_OVERCURRENT_DETECTION	NA
2	Motor Channel and Motor selection	File : user_config.h Select one of the BLDC motors to be connected to channel 0. e.g. #define MOTOR_CHANNEL_0 "motor_define_ACT_42BLF01_3sh_opa.h"	Connect the U,V,W phase wires as shown below CN1 : U, V, W
3	Motor Parameter Configuration	File: motor_define_ACT_42BLF01_x.h #define MOTOR_POLE_PAIRS 4 #define MOTOR_DIRECTION 2	Connect target motor and specify the number of poles, Torque factor, motor resistance , inductance and applicable current limits as per datasheet.

<table>
<tr><td>4</td><td>Sensor less control</td><td>#define MOTOR_ANGULAR_ACC_MAX_VALUE 1000
#define MOTOR_ANGULAR_ACC_MAX_UNIT RAD_BY_S2
#define MOTOR_TORQUE_FACTOR_VALUE 35
#define MOTOR_TORQUE_FACTOR_UNIT MNM_BY_A
#define MOTOR_RESISTANCE_VALUE 900
#define MOTOR_RESISTANCE_UNIT MOHM
#define MOTOR_INDUCTANCE_VALUE 270
#define MOTOR_INDUCTANCE_UNIT UH
#define MOTOR_SPEED_LIMIT_VALUE 4000
#define MOTOR_SPEED_LIMIT_UNIT RPM
#define MOTOR_SPEED_CHANGE_VALUE 2000
#define MOTOR_SPEED_CHANGE_UNIT RPM
#define MOTOR_POSITION_DELAY_VALUE 200
#define MOTOR_POSITION_DELAY_UNIT 0 /* [ms] */
#define MOTOR_IQ_START_VALUE 1500
#define MOTOR_IQ_START_UNIT MAMPERE
#define MOTOR_ID_START_VALUE 0
#define MOTOR_ID_START_UNIT MAMPERE
#define MOTOR_IQ_LIM_VALUE 1900
#define MOTOR_IQ_LIM_UNIT MAMPERE
#define MOTOR_ID_LIM_VALUE 0
#define MOTOR_ID_LIM_UNIT MAMPERE</td><td>Connect the U,V,W phase wires as shown below
CN1 : U, V, W</td></tr>
<tr><td>5</td><td>Hall sensor connection</td><td rowspan="2">File: motor_define_ACT_42BLF01_x.h
Set the Hall sensor type, usage and count settings
#define MOTOR_ENCODER_TYPE 0
#define MOTOR_ENCODER_USAGE 0
#define MOTOR_ENCODER_COUNT 6
#define MOTOR_ENCODER_MIN_COUNT 0
#define MOTOR_ENCODER_START_COUNT 15000
#define MOTOR_ENCODER_MAX_COUNT 30000
#define MOTOR_ENCODER_LM_APPROACH_RPM 10
#define MOTOR_ENCODER_RECEIVER 3
#define MOTOR_ENCODER_RECEIVER_MODE 1</td><td>Connect the U,V,W phase wires as shown below
CN1 : U, V, W
Connect Hall sensor wires as shown below
TB2 ~ 4 (A:Hu, A bar:NC, B:Hv, B bar:NC, Z:Hw, Z bar:NC)
TB5 (5V, GND)</td></tr>
<tr><td>6</td><td>Incremental Encoder connection</td><td>Connect the U,V,W phase wires as shown below
CN1 : U, V, W
Connect incremental encoder as shown below
TB2 ~ 4 (A:A, A bar:A bar, B:B, B bar:B bar, Z:Z, Z bar:Z bar)
TB5 (5V, GND)</td></tr>
<tr><td>7</td><td>PI configuration</td><td>File: motor_define_ACT_42BLF01_x.h
Motor Tuning parameters
#define CONTROL_ID_KI_VALUE 100
#define CONTROL_ID_KI_UNIT 0
#define CONTROL_ID_KP_VALUE 75
#define CONTROL_ID_KP_UNIT 0
#define CONTROL_IQ_KI_VALUE 100
#define CONTROL_IQ_KI_UNIT 0
#define CONTROL_IQ_KP_VALUE 75
#define CONTROL_IQ_KP_UNIT 0
#define CONTROL_POSITION_KI_VALUE 0
#define CONTROL_POSITION_KI_UNIT 0
#define CONTROL_POSITION_KP_VALUE 1000
#define CONTROL_POSITION_KP_UNIT 0
#define CONTROL_SPEED_KI_VALUE 100
#define CONTROL_SPEED_KI_UNIT 0
#define CONTROL_SPEED_KP_VALUE 75
#define CONTROL_SPEED_KP_UNIT 0</td><td>NA</td></tr>
<tr><td>8</td><td>System Configuration</td><td>File: motor_define_ACT_42BLF01_x.h
#define SYSTEM_PWM_FREQUENCY_VALUE 16000
#define SYSTEM_PWM_FREQUENCY_UNIT 0
#define SYSTEM_SHUTDOWN_MODE 2
#define SYSTEM_BRAKE_TIME_VALUE 0
#define SYSTEM_BRAKE_TIME_UNIT 0
#define SYSTEM_BRAKE_PERCENTAGE_VALUE 0
#define SYSTEM_BRAKE_PERCENTAGE_UNIT 0
#define SYSTEM_RESTART_MODE 1
#define SYSTEM_STALL_DETECT_VALUE 10
#define SYSTEM_STALL_DETECT_UNIT 0
#define SYSTEM_OVERTEMPERATURE_VALUE 60
#define SYSTEM_OVERTEMPERATURE_UNIT 0
#define SYSTEM_SPEED_CONTROL_MODE 0
#define SYSTEM_SW_OVERVOLTAGE_VALUE 26
#define SYSTEM_SW_OVERVOLTAGE_UNIT 0
#define SYSTEM_SW_UNDERVOLTAGE_VALUE 15
#define SYSTEM_SW_UNDERVOLTAGE_UNIT 0
#define SYSTEM_SW_OVERCURRENT_VALUE 7
#define SYSTEM_SW_OVERCURRENT_UNIT AMPERE
#define SYSTEM_SPEED_REDUCTION_VALUE 100
#define SYSTEM_SPEED_REDUCTION_UNIT 0</td><td>NA</td></tr>
<tr><td>9</td><td>1-shunt and 3 shunt setting</td><td>File name : clicker4_m4kn_pwr.h
Set the value of “BOARD_MEASUREMENT_TYPE” to CURRENT_SHUNT_1 or CURRENT_SHUNT_3</td><td><table><tr><td>Feedback</td><td>J1</td><td>J2</td><td>J3</td><td>J4</td></tr><tr><td>3-shunt</td><td>Open</td><td>Open</td><td>Open</td><td>Short</td></tr><tr><td>1-shunt</td><td>Short</td><td>Short</td><td>Short</td><td>Open</td></tr></table></td></tr>
<tr><td>10</td><td>Power Board</td><td>File name : clicker4_m4kn_pwr.h
#define BOARD_NAME_PWR</td><td>NA</td></tr>
</table>

	Configuration	"CLICKER4-INV-SHIELD" #define BOARD_DEAD_TIME 800 #define BOARD_BOOTSTRAP_DELAY 100	
11	Current and Voltage sensitivity	File name : clicker4_m4kn_pwr.h #define BOARD_SENSITIVITY_CURRENT_MEASURE_VALUE 164 #define BOARD_SENSITIVITY_VOLTAGE_MEASURE_VALUE 77	NA
12	Motor Name	File: motor_define_ACT_42BLF01_x.h #define MOTORID "ACT 42BLF01" (Upto 32 characters)	NA
13	Standalone Demo control using Slider2 Clicker	File : standalone_config.h #define DEMO_CLICKER4_SLIDER2	Connect the “Slider 2 click” to MicroBus-4.

Table 5.1 Quick reference for Clicker 4 board

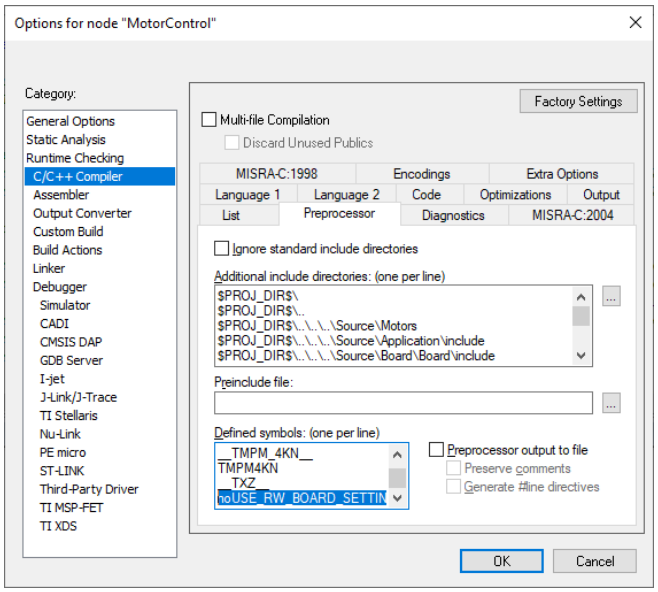

Figure 5.15 Read-Only Board Settings override

The MOSFET dead time, typically given in ns, is a very important parameter that shall be configured very carefully. It is recommended to start with very conservative values, at least 30% longer than the ones given in the datasheet. It is also advisory to ensure that the data sheet used is the most recent and/or the exact one for the particular hardware revision of the MOSFET devices used on the output/power board. It has to be ensured that the power supply current limiter is engaged and active for the time or tuning all parameters, dead time in particular.

5.9. First Run & Adjustment with MCU Motor Studio

MCU Motor Studio has proven its efficiency for the initial system or motor configuration as most of the parameters may be adjusted on the fly or calculated using its helper functions.

5.9.1. Rules

The following rules are advisable:

Static change rule – changes shall be done with stopped/disconnected motor as far as possible

One change rule – do not change more than one parameter at a time, for better judgement of the effect

Safety first rule – ensure current limiter is engaged to avoid any injuries

Known parameter rule – start with all known parameters, ensure to use the latest datasheet available

for the motor & boards.

Unknown parameter rule – some parameters are not so important, the number of poles for example. Others may be measured, like the resistance and inductance of the motor. In all other cases we have some “best practice” values which shall be used as a starting point and tuned one by one.

5.9.2. Unknown Parameters

The following “best practice“ values may be used during the initial set-up and only if the corresponding parameter is not known:

Polepair – set to 1, if wrong the angular speed will be different then the set one. The “pole pair calculator” shall be used later on to find the proper value.

Direction – set to **MOTOR_CW_CCW** or **2**, as most motors are capable of both.

Encoder – set the type to none, all encoder related parameters will not matter in that case.

Maximal Angular Acceleration – start with value in the range of 100 ~ 300 rad/s2, reduce if needed to get the system up and running, then fine tune together with the PI regulation parameters.

Torque Factor – in the default speed control the value is unused, thus any start value shall be fine.

Resistance / Inductance – shall be measured.

Speed Change – take the **Speed Limit+1** to avoid change in FOC mode. The motor shall be driven in Forced mode at first.

Position Delay – very much system dependent, typically 200 ~ 500 ms shall be sufficient

Id Start/ Id Limit – keep “0”

Iq Start/ Iq Limit – keep same and start with pretty low value, very much motor dependent, typical initial value is 200 mA.

PWM Frequency – start with 16kHz, reduce if needed.

Shutdown mode – “no signals“

Id Ki / Iq Ki – start with “40”

Id Kp/ Iq Kp – start with “20”

Position Ki – set to “0”

Position Kp – set to “15000”

Speed Ki – set to “10”

Speed Kp – set to “20“

Keep all convenience or software protection features disabled. The Parameter tab shall/may look like, resistance and inductance still need to be entered:

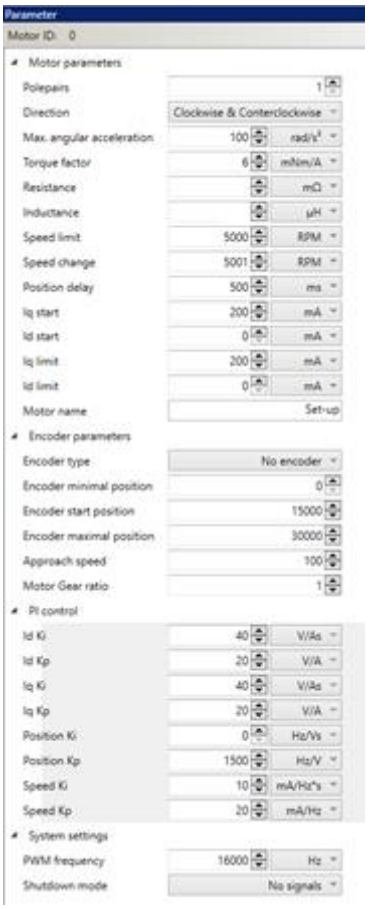

Figure 5.16 “Best practice” start-up values

5.9.3. "Rotate" in Forced Mode

Select the Motor ID, confirm all relevant parameters are properly set and the unknown values are configured as described in Chapter 5.9.2. Start the MCU Motor Studio statistics with 500ms update interval. Set the desired speed to 60 RPM in CW direction and start the motor.

Increase the integral and proportional gain of **Id** and **Iq**. Eventually increase the **Iq start** & **Iq limit** values by 50 to 100mA. Continue until the motor starts revolving. Stop/start the motor a few times, ensuring it is successful every time.

While motor stopped, set the target speed to the maximal supported. Start the motor. If it stalls, stop and again increase **Id Ki, Iq Ki, Id Kp, Iq Kp, Iq start** and **Iq limit**. Ensure stable operation.

Forced mode is now configured, FOC is to be done.

5.9.4. "Rotate" in FOC Mode

While the motor is stopped, set the Speed change parameter to 50% of the Speed limit. Set the target speed at 60 ~ 70% of the Speed limit. Start the motor and wait until the set speed is reached.

If the motor stalls, reduce the maximal angular acceleration and modify Position Ki and Position Kp. Speed Ki and Speed Kp may also be slightly changed in the one or the other direction.

Switch off Torque and Current view in the statistics graph. Modify Speed Ki/Kp and Position Ki/Kp so that the variability of the actual speed is minimized around the target speed. The actual speed has to be around the target speed and not constantly below or above it. If this happens the values on Position Ki/Kp and Speed Ki/Kp are not properly set.

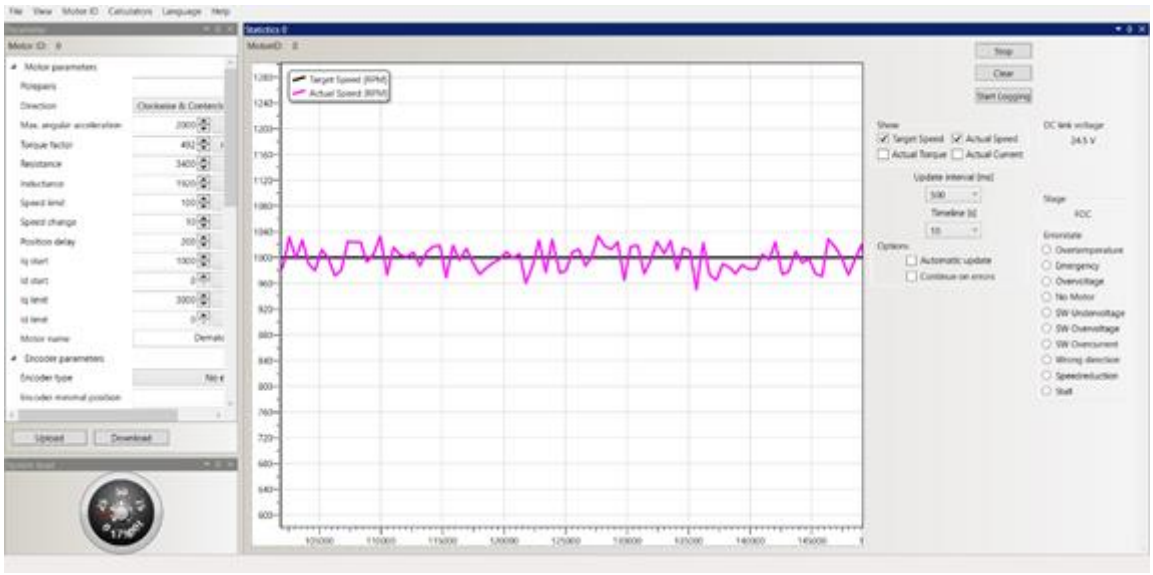

Figure 5.17 Actual speed fluctuations in FOC

Once stable operation is achieved the Speed change parameter may be further reduced and the above adjustment shall be carried out again. At a certain value no further optimization may be possible.

5.9.5. "Position Kp" Adjustment

There is a very simple solution for determining reasonable value for the Position Kp. The target speed shall be set right below the **Change speed** so that the motor is controlled in Forced mode, right before changing to FOC. Parameters **Omega** and **Omega$_{calc}$** shall be monitored by the DSO with a spread factor of 5 to 10.

Omega shall always be flat line (depending on the change frequency) representing the selected rotational speed (in this example in Hz).

Omega$_{calc}$ shall always be a curve, close to a sinusoidal one. It has to be ensured that the spread of the DSO capture is set to a low number, so that a few PWM cycles are recorded only.

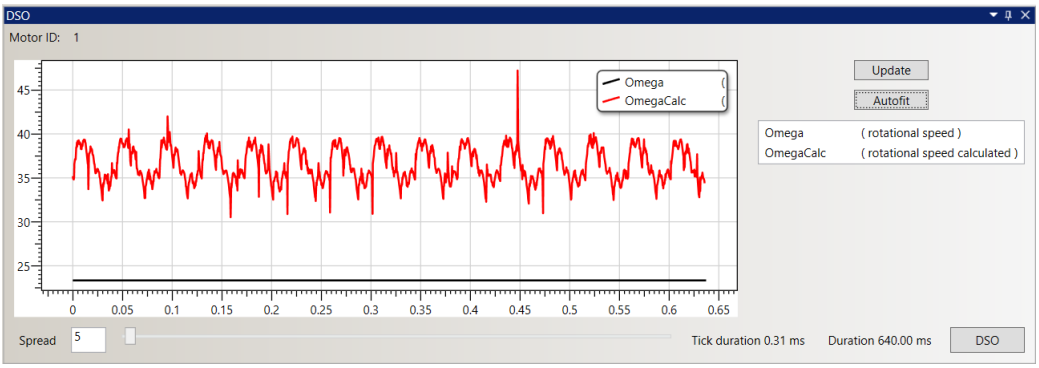

Figure 5.18 Position Kp adjustment using Omega / Omega$_{calc}$

Changing the Position Kp and updating the DSO capture will result a change in the offset between the two signals.

Position Kp shall be adjusted until the **Omega$_{calc}$** curve is about 10-20% above the Omega line.

The proper operation in FOC mode shall be confirmed by increasing the target speed. Further fine adjustment may be done in combination with then PI regulation parameters.

5.9.6. “Stall Detector Threshold” Adjustment

The voltage drop ratio in case of motor stall can be measured with the use of the DSO/HS-DSO and the **SYSTEM_STALL_DETECT_VALUE** may be tuned to achieve even more precise detection.

The target speed shall be set to the Change speed so that the motor is controlled in FOC mode.

The integral part of the toqrue axis voltage, parameter **Vqi**, shall be monitored by the DSO with a spread factor of at least 50, although the maximum of 256 is recommended. While the DSO is still collecting data a motor stall shall be provoked.

The outcome will be similar to the capture below:

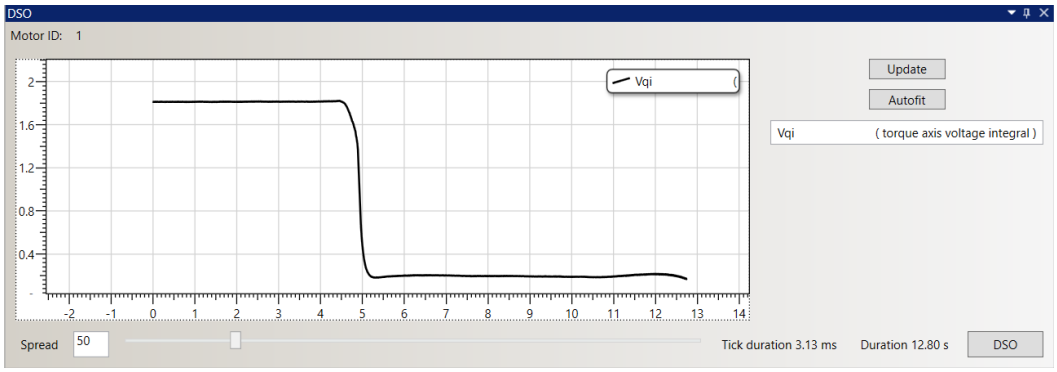

Figure 5.19 Stall detection adjustment using Vq

The difference is about 1.6V. A drop of half of it shall be taken as the actual threshold. This experimentally determined value of 0.8V has to be multiplied by 1000, resulting the new constant for the **SYSTEM_STALL_DETECT_VALUE** parameter:

```
#define CONTROL_SPEED_KI_UNIT          0      /* [mA/Hz*s] */
#define CONTROL_SPEED_KP_VALUE         50
#define CONTROL_SPEED_KP_UNIT          0      /* [mA/Hz] */
#define SYSTEM_PWM_FREQUENCY_VALUE     16000
#define SYSTEM_PWM_FREQUENCY_UNIT      0      /* [Hz] */
#define SYSTEM_SHUTDOWN_MODE           2
#define SYSTEM_BRAKE_TIME_VALUE        0
#define SYSTEM_BRAKE_TIME_UNIT         0      /* [ms] */
#define SYSTEM_BRAKE_PERCENTAGE_VALUE  0
#define SYSTEM_BRAKE_PERCENTAGE_UNIT   0      /* [%] */
#define SYSTEM_RESTART_MODE            1
#define SYSTEM_STALL_DETECT_VALUE      800
#define SYSTEM_STALL_DETECT_UNIT       0      /* [mV/s] */
#define SYSTEM_OVERTEMPERATURE_VALUE   60
#define SYSTEM_OVERTEMPERATURE_UNIT    0      /* [□] */
#define SYSTEM_SPEED_CONTROL_MODE      0
#define SYSTEM_SW_OVERVOLTAGE_VALUE    0
#define SYSTEM_SW_OVERVOLTAGE_UNIT     0      /* [V] */
#define SYSTEM_SW_UNDERVOLTAGE_VALUE   0
#define SYSTEM_SW_UNDERVOLTAGE_UNIT    0      /* [V] */
#define SYSTEM_SW_OVERCURRENT_VALUE    0
#define SYSTEM_SW_OVERCURRENT_UNIT     AMPERE
#define SYSTEM_SPEED_REDUCTION_VALUE   100
#define SYSTEM_SPEED_REDUCTION_UNIT    0      /* [%] */
```

Figure 5.20 Stall detection adjustment using Vqi

6. References

[1] TMPM4K Group(2) Datasheet, Revision 1.0, October 2018, Toshiba Electronic Devices & Storage Corporation
[2] Reference Manual Advanced Vector Engine Plus (A-VE+-B), Revision 3.0, May 2018, Toshiba Electronic Devices & Storage Corporation
[3] Reference Manual Advanced Programmable Motor Control Circuit (A-PMD-A), Revision 2.1, July 2020, Toshiba Electronic Devices & Storage Corporation
[4] Reference Manual Advanced Encoder Input Circuit(32-bit) (A-ENC32-A), Revision 1.1, October 2018, Toshiba Electronic Devices & Storage Corporation
[5] Reference Manual 12-bit Analog to Digital Converter (ADC-I), Revision 0.1, July 2020, Toshiba Electronic Devices & Storage Corporation.

7. Revision History

Table 7.1 Revision History

Revision	Date	Changes
1.0.0	2022/05/06	Baselined Version
1.1.0	2022/12/01	Updated for ver1.1 release Introduction section and DSO section updated.

Trademarks

- FreeRTOS™ is a trademark of Amazon Web Services, inc in the US and/or elsewhere. All rights reserved.
- Microsoft® and Windows® are either registered trademarks Microsoft Corporation in the United States and/or elsewhere. All rights reserved.
- Arm® , Cortex® ,Cortex®-M3, Cortex®-M4,Keil® and µVision® are registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere.
- Click boards™ is a trademark of MIKROELEKTRONIKA. All rights reserved.
- FTDI may be registered trademarks of “Future Technology Devices International Limited”. All rights reserved.
- IAR Systems® and IAR Embedded Workbench® are registered trademarks are owned by IAR Systems. All rights reserved
- SEGGER and J-Link are trademarks or registered trademarks of SEGGER Microcontroller GmbH & Co. KG. All rights reserved.

Other Company names, product names and service names mentioned herein may be trademarks of their respective companies.

RESTRICTIONS ON PRODUCT USE

Toshiba Corporation and its subsidiaries and affiliates are collectively referred to as “TOSHIBA”.
Hardware, software and systems described in this document are collectively referred to as “Product”.

- TOSHIBA reserves the right to make changes to the information in this document and related Product without notice.
- This document and any information herein may not be reproduced without prior written permission from TOSHIBA. Even with TOSHIBA's written permission, reproduction is permissible only if reproduction is without alteration/omission.
- Though TOSHIBA works continually to improve Product's quality and reliability, Product can malfunction or fail. Customers are responsible for complying with safety standards and for providing adequate designs and safeguards for their hardware, software and systems which minimize risk and avoid situations in which a malfunction or failure of Product could cause loss of human life, bodily injury or damage to property, including data loss or corruption. Before customers use the Product, create designs including the Product, or incorporate the Product into their own applications, customers must also refer to and comply with (a) the latest versions of all relevant TOSHIBA information, including without limitation, this document, the specifications, the data sheets and application notes for Product and the precautions and conditions set forth in the "TOSHIBA Semiconductor Reliability Handbook" and (b) the instructions for the application with which the Product will be used with or for. Customers are solely responsible for all aspects of their own product design or applications, including but not limited to (a) determining the appropriateness of the use of this Product in such design or applications; (b) evaluating and determining the applicability of any information contained in this document, or in charts, diagrams, programs, algorithms, sample application circuits, or any other referenced documents; and (c) validating all operating parameters for such designs and applications. **TOSHIBA ASSUMES NO LIABILITY FOR CUSTOMERS' PRODUCT DESIGN OR APPLICATIONS.**
- **PRODUCT IS NEITHER INTENDED NOR WARRANTED FOR USE IN EQUIPMENTS OR SYSTEMS THAT REQUIRE EXTRAORDINARILY HIGH LEVELS OF QUALITY AND/OR RELIABILITY, AND/OR A MALFUNCTION OR FAILURE OF WHICH MAY CAUSE LOSS OF HUMAN LIFE, BODILY INJURY, SERIOUS PROPERTY DAMAGE AND/OR SERIOUS PUBLIC IMPACT ("UNINTENDED USE").** Except for specific applications as expressly stated in this document, Unintended Use includes, without limitation, equipment used in nuclear facilities, equipment used in the aerospace industry, lifesaving and/or life supporting medical equipment, equipment used for automobiles, trains, ships and other transportation, traffic signaling equipment, equipment used to control combustions or explosions, safety devices, elevators and escalators, and devices related to power plant. **IF YOU USE PRODUCT FOR UNINTENDED USE, TOSHIBA ASSUMES NO LIABILITY FOR PRODUCT.** For details, please contact your TOSHIBA sales representative or contact us via our website.
- Do not disassemble, analyze, reverse-engineer, alter, modify, translate or copy Product, whether in whole or in part.
- Product shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable laws or regulations.
- The information contained herein is presented only as guidance for Product use. No responsibility is assumed by TOSHIBA for any infringement of patents or any other intellectual property rights of third parties that may result from the use of Product. No license to any intellectual property right is granted by this document, whether express or implied, by estoppel or otherwise.
- **ABSENT A WRITTEN SIGNED AGREEMENT, EXCEPT AS PROVIDED IN THE RELEVANT TERMS AND CONDITIONS OF SALE FOR PRODUCT, AND TO THE MAXIMUM EXTENT ALLOWABLE BY LAW, TOSHIBA (1) ASSUMES NO LIABILITY WHATSOEVER, INCLUDING WITHOUT LIMITATION, INDIRECT, CONSEQUENTIAL, SPECIAL, OR INCIDENTAL DAMAGES OR LOSS, INCLUDING WITHOUT LIMITATION, LOSS OF PROFITS, LOSS OF OPPORTUNITIES, BUSINESS INTERRUPTION AND LOSS OF DATA, AND (2) DISCLAIMS ANY AND ALL EXPRESS OR IMPLIED WARRANTIES AND CONDITIONS RELATED TO SALE, USE OF PRODUCT, OR INFORMATION, INCLUDING WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, ACCURACY OF INFORMATION, OR NONINFRINGEMENT.**
- Do not use or otherwise make available Product or related software or technology for any military purposes, including without limitation, for the design, development, use, stockpiling or manufacturing of nuclear, chemical, or biological weapons or missile technology products (mass destruction weapons). Product and related software and technology may be controlled under the applicable export laws and regulations including, without limitation, the Japanese Foreign Exchange and Foreign Trade Law and the U.S. Export Administration Regulations. Export and re-export of Product or related software or technology are strictly prohibited except in compliance with all applicable export laws and regulations.
- Please contact your TOSHIBA sales representative for details as to environmental matters such as the RoHS compatibility of Product. Please use Product in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. **TOSHIBA ASSUMES NO LIABILITY FOR DAMAGES OR LOSSES OCCURRING AS A RESULT OF NONCOMPLIANCE WITH APPLICABLE LAWS AND REGULATIONS.**

Toshiba Electronic Devices & Storage Corporation
https://toshiba.semicon-storage.com/