NXP Semiconductors

User's Guide

Document identifier: MCSXTE2BK142SWUG

Rev. 1, April, 2020

MCSXTE2BK142 Software User Guide

Contents

Chapter 1
Introduction

This user guide describes in detail the steps to install the necessary components to build the customer application. The following table shows the abbreviation used in the document.

Table 1. Acronyms and abbreviations

Abbreviation	Description
HW	Hardware
SW	Software
SDK	Software Development Kit
BSP	Board Support Package
LLD	Low-Level Driver
API	Application Interface
POR	Power-On Reset
BLDC	Brushless Direct Current Motor
PMSM	Permanent-Magnet Synchronous Motor
FOC	Field-Oriented Control
RTM	Ready to Manufacture

You need to download and install the following tools explained in the following sections.

1.1 S32 Design Studio IDE introduction

The S32 Design Studio IDE is a complimentary Integrated Development Environment (IDE) for automotive and ultra-reliable Power Architecture® (e200 core) and Arm® based microcontrollers. Based on open-source software including Eclipse IDE, GNU Compiler Collection (GCC), and GNU Debugger (GDB), the S32 Design Studio IDE is a straightforward development tool with no code-size limitations that enables editing, compiling, and debugging of designs.

NXP software, along with the S32 Design Studio, provides a comprehensive enablement environment that reduces development time.

According to different chip architecture (PPC/ARM), S32DS IDE has different version. For this project, S32 DS IDE for Arm is used.

Download the latest version from NXP website (Download the S32DS for Arm from NXP website (SW project created in V2018.R1)).

1.2 FreeMASTER introduction

FreeMASTER is a user-friendly real-time debug monitor and data visualization tool that you can use for application development and information management. Supports non-intrusive monitoring of variables on a running system. You can display multiple variables changing over time on an oscilloscope-like display or view the data in text form. FreeMASTER supports additional capabilities and targets with an on-target driver for transmitting data from the target to the host computer. Ideal for automotive, industrial or consumer applications.

Features:

- Real Time Monitor
- Control Panel
- Demonstration Platform
- Easy Project Deployment

In order to debug the motor conveniently, we customized a motor debugging GUI tool based on FreeMASTER, named MCAT (Motor Control Application Tuning Tool). To download, click FreeMASTER.

Chapter 2
System Features

The software package of MCSXTE2BK142 board enables user to evaluate the S32K142 based motor control performance with out-of-box and build their own motor control product prototype as a general motor control hardware platform.

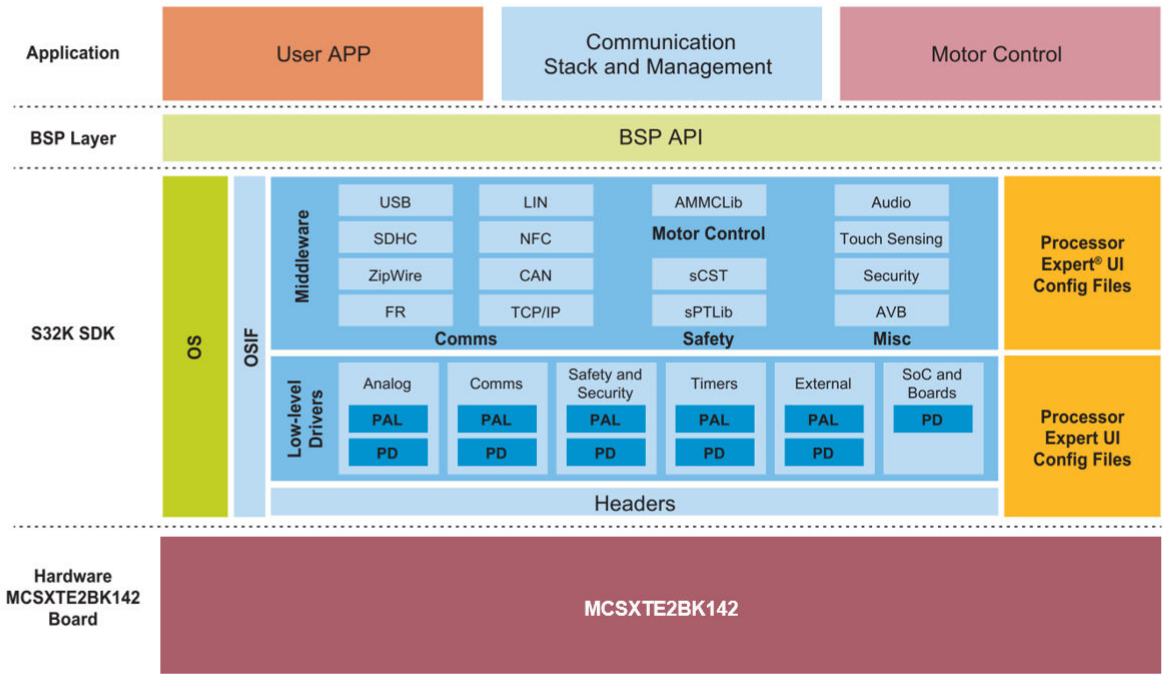
The software package has the following features:

- Out-of-box motor control and tuning via FreeMASTER MCAT.
- Support sensorless FOC PMSM/BLDC motor control.
- Support hall sensor based FOC PMSM/BLDC motor control.
- Support rich motor control diagnosis and protection: OV, UV, OC, OT and so on.
- Implement advanced motor control algorithm -- FW (Field Weakening) and stall detection.
- Support dual-shunt and tri-shunt phase current sample.
- Provide S32DS IDE project and makefile project to support multi-toolchain—GHS, IAR and GCC and multi-debugger—Lauterbach, U-Multilink and J-LINK debugger.

Chapter 3
Software Architecture Overview

The motor control software package is developed on NXP S32K1xx SDK RTM3.0.0 and AMMCLIB 1.1.15. In order to accelerate user application SW development, it offers a BSP layer to provide API of all onboard modules. In application layer, apart from the motor control, user can add their own application SW and communication management stack.

Figure 1. The board software architecture



3.1 S32K1xx SDK introduction

The S32K1xx Software Development Kit (S32K1xx SDK) is an extensive suite of peripheral abstraction layers, peripheral drivers, RTOS, stacks, and middleware designed to simplify and accelerate application development on NXP S32K microcontrollers.

All software included in this release have RTM quality level in terms of features, testing and quality documentation as per NXP software release criteria. RTM releases contain all planned features implemented and tested. RTM releases are candidates that can be used in production.

This SDK can be used standalone or it can be used with S32 Design Studio IDE (see Supported hardware and compatible software).

The addition of Processor Expert technology for software and board configuration provides unmatched ease of use and flexibility. Included in the S32 SDK is full source code under a permissive open-source license for all hardware abstraction and peripheral driver software. See the Release Notes for details. The S32 SDK consists of the following runtime software components written in C:

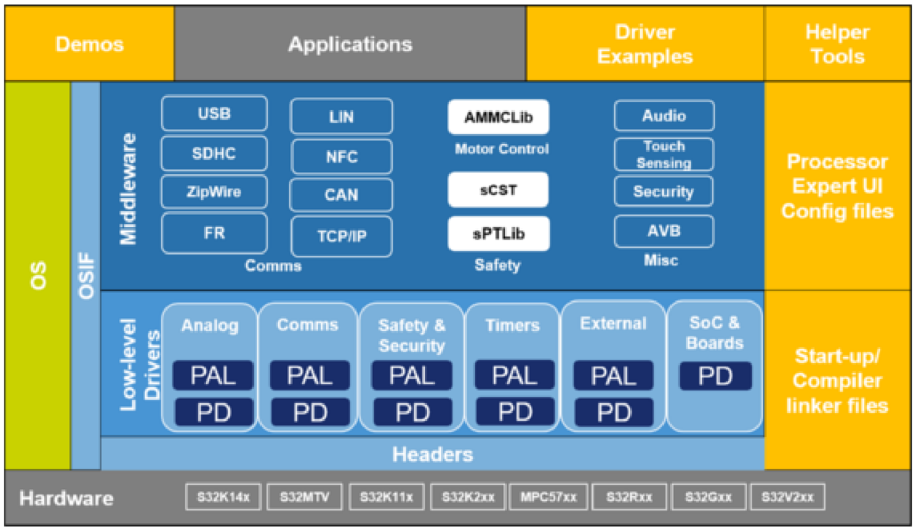


Figure 2. S32 SDK architecture

3.1.1 S32K1xx SDK development guidelines

Guidelines on SDK programming model:

1. Driver state structures should be declared as global or static variables as they are used in the whole time when the driver is used.
2. Driver state structures content should not be used or modified by the application code.
3. Peripheral drivers, PALs and Middleware code are not handling clock and pins initialization. Configuration of the clock and pins driver has to be done by the application. To make sure these are properly initialized before other modules are used, please call the corresponding initialization.

```
/* Initialize and configure clocks */
CLOCK_SYS_Init(g_clockManConfigsArr, CLOCK_MANAGER_CONFIG_CNT,
g_clockManCallbacksArr, CLOCK_MANAGER_CALLBACK_CNT);
CLOCK_SYS_UpdateConfiguration(0U, CLOCK_MANAGER_POLICY_AGREEMENT);
/* Initialize pins */
PINS_DRV_Init(NUM_OF_CONFIGURED_PINS, g_pin_mux_InitConfigArr);
```

NOTE

The configuration structure names used in this example are the default names generated by Processor Expert components for clock and pins. Applications not using processor expert might have different names for these structures.

4. The recommended approach at development time is to add DEV_ERROR_DETECT symbol to the compiler defines. This will enable DEV_ASSERT mechanism which can catch application code errors in the early development stage.
5. Special care should be taken to have a backup option when debug pins are routed to other functionalities.

3.1.2 S32K1xx SDK error detection and reporting

S32 SDK drivers use a mechanism to validate data coming from upper software layers (application code) by performing a number of checks on input parameters' range or other invariants that can be statically checked (not dependent on runtime conditions). A failed validation is indicative of a software bug in application code, therefore it is important to use this mechanism during development.

The validation is performed by using DEV_ASSERT macro. A default implementation of this macro is provided in this file. However, application developers can provide their own implementation in a custom file. This requires defining the CUSTOM_DEVASSERT symbol with the specific file name in the project configuration (Example: -DCUSTOM_DEVASSERT ="custom_devassert.h")

The default implementation accommodates two behaviors, based on DEV_ERROR_DETECT symbol:

- When DEV_ERROR_DETECT symbol is defined in the project configuration (Example: DDEV_ERROR_DETECT), the validation performed by the DEV_ASSERT macro is enabled, and a failed validation triggers a software breakpoint and further execution is prevented (application spins in an infinite loop) This configuration is recommended for development environments, as it prevents further execution and allows investigating potential problems from the point of error detection.
- When DEV_ERROR_DETECT symbol is not defined, the DEV_ASSERT macro is implemented as no-op, therefore disabling all validations. This configuration can be used to eliminate the overhead of development-time checks.

It is the application developer's responsibility to decide the error detection strategy for production code. One can opt to disable development-time checking altogether (by not defining DEV_ERROR_DETECT symbol), or can opt to keep the checks in place and implement a recovery mechanism in case of a failed validation, by defining CUSTOM_DEVASSERT to point to the file containing the custom implementation.

3.2 NXP AMMCLib introduction

The AMMCLib (Automotive Math and Motor Control Library) Set for NXP S32K14x devices consists of several sub-libraries, functionally connected as depicted in AMMCLIB components structure.

The Automotive Math and Motor Control Library Set for NXP S32K14x devices sublibraries are as follows:

- Mathematical Function Library (MLIB): comprising basic mathematical operations such as addition, multiplication, etc.
- General Function Library (GFLIB): comprising basic trigonometric and general math functions such as sine, cosine, tan, hysteresis, limit, etc.
- General Digital Filters Library (GDFLIB): comprising digital IIR and FIR filters designed to be used in a motor control application.
- General Motor Control Library (GMCLIB): comprising standard algorithms used for motor control such as Clarke/Park transformations, Space Vector Modulation, etc.
- Advanced Motor Control Function Library (AMMCLib): comprising advanced algorithms used for motor control purposes.

The Automotive Math and Motor Control Library Set for NXP S32K14x devices is developed to support these major implementations:

- Fixed-point 32-bit fractional
- Fixed-point 16-bit fractional
- Single precision floating point

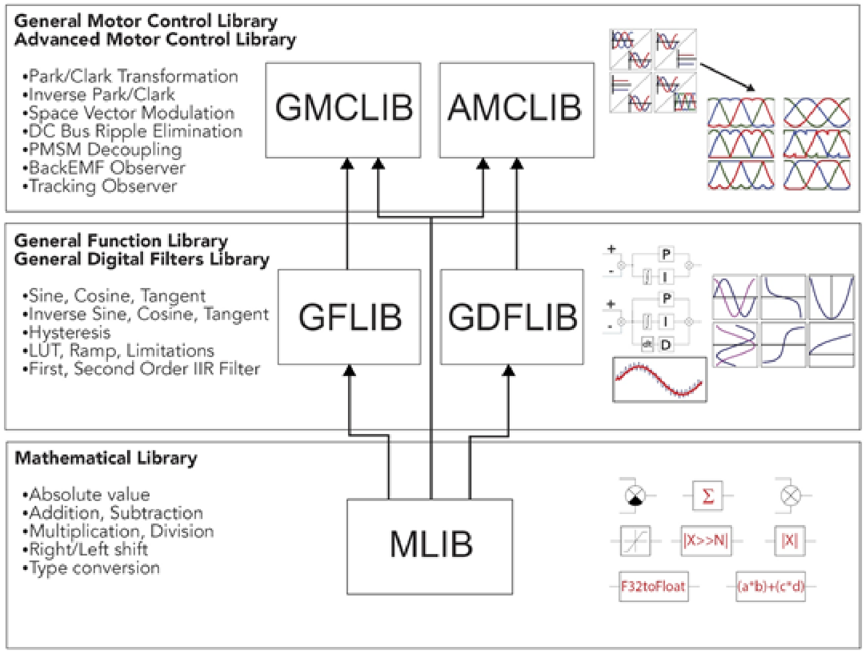
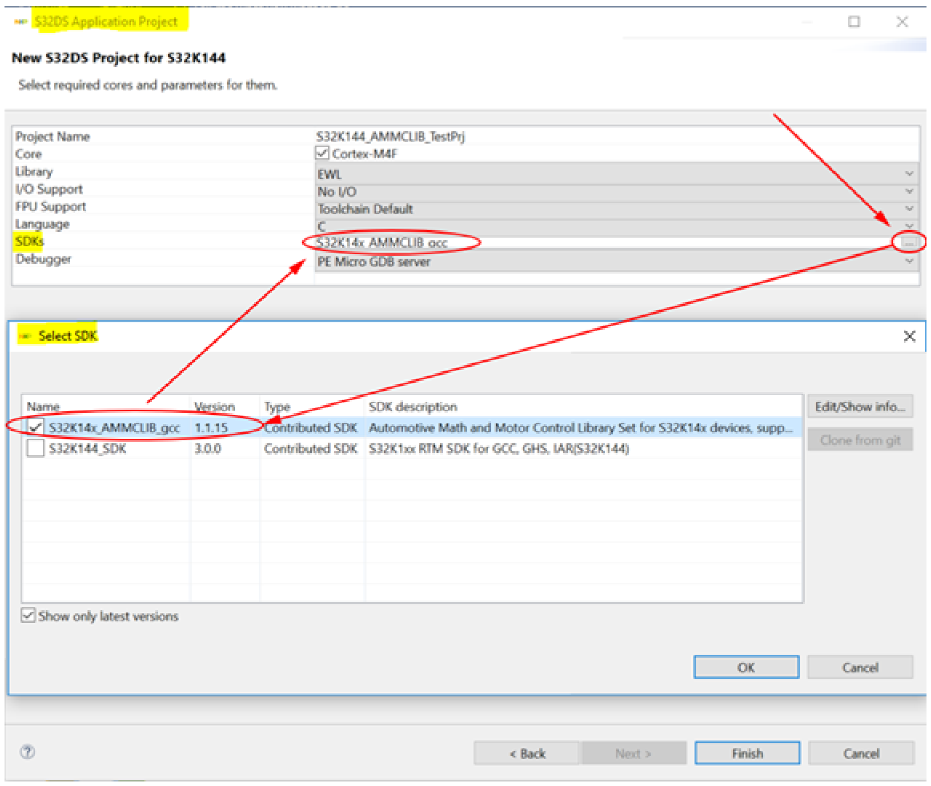


Figure 3. AMMCLIB components structure

More details about S32K14x AMMCLIB please refer S32K14XMCLUG.

3.2.1 Using AMMCLIB as a SDK in S32DS IDE

As introduced in S32K1xx SDK introduction, AMMCLIB is a very important middleware of S32K1xx SDK. In order to integrate it in application project, user must add the library header files and search path as well as toolchain compiled library binary file and configure file into to project. S32DS provides a convenient method to facilitate the operation, the AMMCLIB can be added into the application project when creating a new application project and select it as a SDK.


S32DS Application Project

New S32DS Project for S32K144

Select required cores and parameters for them.

Project Name	S32K144_AMMCLIB_TestPrj
Core	✓ Cortex-M4F
Library	EWL
I/O Support	No I/O
FPU Support	Toolchain Default
Language	C
SDKs	S32K14x_AMMCLIB_gcc
Debugger	PE Micro GDB server

Select SDK

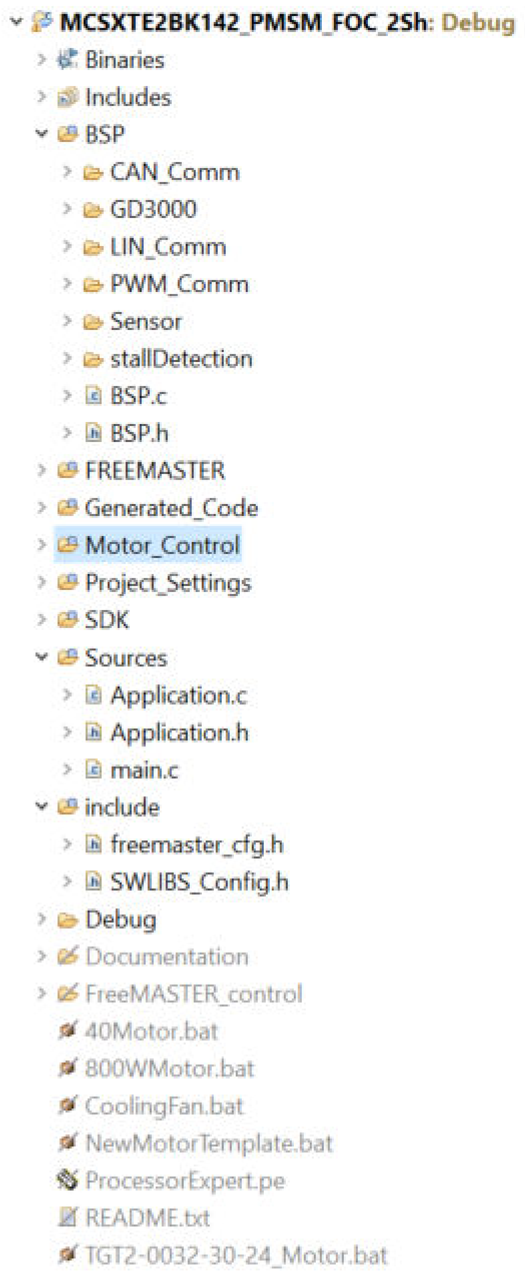
Name	Version	Type	SDK description
✓ S32K14x_AMMCLIB_gcc	1.1.15	Contributed SDK	Automotive Math and Motor Control Library Set for S32K14x devices, supp...
S32K144_SDK	3.0.0	Contributed SDK	S32K1xx RTM SDK for GCC, GHS, IAR(S32K144)

✓ Show only latest versions

Figure 4. Adding AMMCLIB as a SDK for application project

3.3 Project files tree and function briefing

The following figure shows software project files tree.



```
MCSXTE2BK142_PMSM_FOC_2Sh: Debug
  Binaries
  Includes
  BSP
    CAN_Comm
    GD3000
    LIN_Comm
    PWM_Comm
    Sensor
    stallDetection
    BSP.c
    BSP.h
  FREEMASTER
  Generated_Code
  Motor_Control
  Project_Settings
  SDK
  Sources
    Application.c
    Application.h
    main.c
  include
    freemaster_cfg.h
    SWLIBS_Config.h
  Debug
  Documentation
  FreeMASTER_control
  40Motor.bat
  800WMotor.bat
  CoolingFan.bat
  NewMotorTemplate.bat
  ProcessorExpert.pe
  README.txt
  TGT2-0032-30-24_Motor.bat
```

Figure 5. Project files tree

CAN, LIN and PWM communication, GD3000 pre-driver, and Hall sensor driver code as well as motor stall detection implementation code are located in the **BSP** folder. FreeMASTER driver source codes are under **FreeMASTER**, Motor control related FOC, ADC, FTM, PDB, TRIGMUX codes are placed in **Motor_Control**, **SDK** and **Generated_Code** folder are used to store S32K1xx SDK source code and Processor Expert generated SDK configuration files. Application is expected to be added in **main.c** and **Application.c** under **Sources** folder, **Freemaster_cfg.h** and **SWLINBS_Config.h** in **include** folder are the configuration files for FreeMASTER and AMMCLIB.

Other useful files include:

- ProcessorExpert.pe: project SDK configuration file for Processor Expert.
- 40Motor.bat: to change motor parameters for LINIX 45zwn24-40 motor from S32K144 motor control kit.
- 800WMotor.bat: the batch file to change motor parameters for 800W test motor.

- CoolingFan.bat: the batch file to change motor parameters for cooling fan test motor.
- TGT2-0032-30-24_Motor.bat: the batch file to change motor parameters for TGT2-0032-30-24t motor from MPC5744P PMSM motor control kit.
- NewMotorTemplate.bat: the template batch file to change motor parameters for a new motor.

Multiple Motor Parameter Management has a detailed introduction about how to use the above batch files.

Chapter 4
Motor control implementation

4.1 Fundamental principle of PMSM FOC

High-performance motor control is characterized by smooth rotation over the entire speed range of the motor, full torque control at zero speed, and fast acceleration/ deceleration. To achieve such control, Field Oriented Control is used for PM synchronous motors.

The FOC concept is based on an efficient torque control requirement, which is essential for achieving a high control dynamic. Analogous to standard DC machines, AC machines develop maximal torque when the armature current vector is perpendicular to the flux linkage vector. Thus, if only the fundamental harmonic of stator magnetomotive force is considered, the torque T_e developed by an AC machine, in vector notation, is given by the following equation:

$$T_e = \frac{3}{2} \cdot pp \cdot \overrightarrow{\psi_s} \times \overrightarrow{i_s}$$

Figure 6. Equation 1

pp: The number of motor pole-pairs

I_s: Stator current vector

ψ_s: Represents vector of the stator flux

3/2: Indicates a non-power invariant transformation form

In instances of DC machines, the requirement to have the rotor flux vector perpendicular to the stator current vector is satisfied by the mechanical commutator. Because there is no such mechanical commutator in AC Permanent Magnet Synchronous Machines (PMSM), the functionality of the commutator has to be substituted electrically by enhanced current control. This reveal that stator current vector should be oriented in such a way that component necessary for magnetizing of the machine (flux component) shall be isolated from the torque producing component.

This can be accomplished by decomposing the current vector into two components projected in the reference frame, often called the dq frame that rotates synchronously with the rotor. It has become a standard to position the dq reference frame such that the d-axis is aligned with the position of the rotor flux vector, so that the current in the d-axis will alter the amplitude of the rotor flux linkage vector. The reference frame position must be updated so that the d-axis should be always aligned with the rotor flux axis.

Because the rotor flux axis is locked to the rotor position, when using PMSM machines, a mechanical position transducer or position observer can be utilized to measure the rotor position and the position of the rotor flux axis. When the reference frame phase is set such that the d-axis is aligned with the rotor flux axis, the current in the q-axis represents solely the torque producing current component.

What further resulted from setting the reference frame speed to be synchronous with the rotor flux axis speed is that both d and q axis current components are DC values. This implies utilization of simple current controllers to control the demanded torque and magnetizing flux of the machine, thus simplifying the control structure design.

To perform vector control, it is necessary to perform the following four steps:

- Measure the motor quantities (DC link voltage and currents, rotor position/speed).
- Transform measured currents into the two-phase orthogonal system (α, β) using a Clarke transformation. After that transform the currents in, coordinates into the d, q reference frame using a Park transformation.
- The stator current torque (isq) and flux (isd) producing components are separately controlled in d, q rotating frame.

- The output of the control is stator voltage space vector and it is transformed by an inverse Park transformation back from the d, q reference frame into the two-phase orthogonal system fixed with the stator. The output three-phase voltage is generated using a space vector modulation.

Clarke/Park transformations discussed above are part of the Automotive Math and Motor Control Library set (see section References).

The following two figures show the basic structure of the vector control algorithm for the PM synchronous motor.

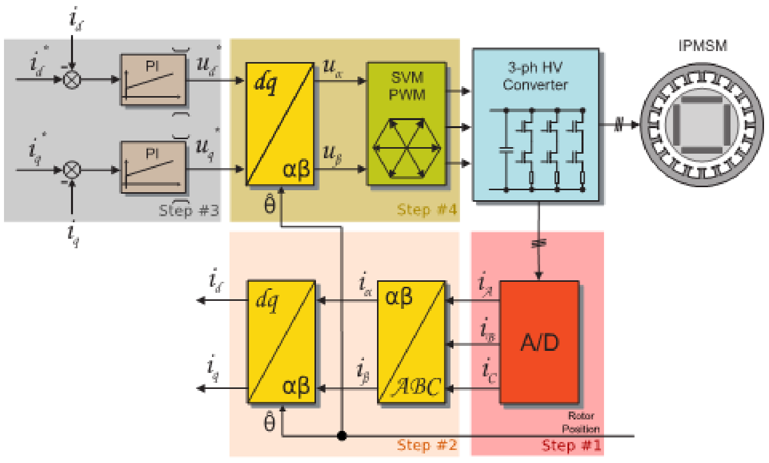


Figure 7. Field oriented control structure

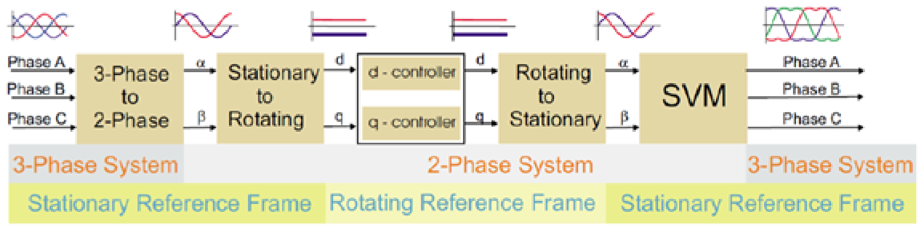


Figure 8. Field oriented control transformations

4.2 Sensorless control

To be able to decompose currents into torque and flux producing components (isd, isq), position of the motor-magnetizing flux has to be known. This requires knowledge of accurate rotor position as being strictly fixed with magnetic flux. This demo system deals with the sensorless FOC control where the position and velocity are obtained by either a position/velocity estimator or incremental Encoder sensor. The estimate method is using back-EMF observer, but back-EMF observer as well as incremental Encoder sensor provide only relative position. To get absolute position, initial position must be known.so a sensorless control system include multiply technical, like alignment algorithm, initial position detection (IPD), back-EMF observer.

Alignment algorithm is the first stage of control system, the alignment algorithm applies DC voltage to d-axis resulting full DC voltage applied to phase A and negative half of the DC voltage applied to phase B, C for a certain period. This will cause the rotor to move to "align" position, where stator and rotor fluxes are aligned. The rotor position in which the rotor stabilizes after applying DC voltage is set as zero position. Motor is ready to produce full startup torque once the rotor is properly aligned.

In the second stage, the field-oriented control is in open-loop mode (Application in sensorless mode must start with open loop), in order to move the motor up to a speed value where the observer provides sufficiently accurate speed and position estimations. As soon as the observer provides appropriate estimates, the rotor speed and position calculation are based on the estimation of a BEMF in the stationary reference frame using a Luenberger type of observer.

When the PMSM reaches a minimum operating speed, a minimum measurable level of BEMF is generated by the rotor's permanent magnets. The BEMF observer then transitions into the closed-loop mode. The feedback loops are then controlled by the estimated angle and estimated speed signals from the BEMF observer.

BEMF observer is as a part of the NXP's Automotive Math and Motor Control library. Following figure shows the BEMF structure.

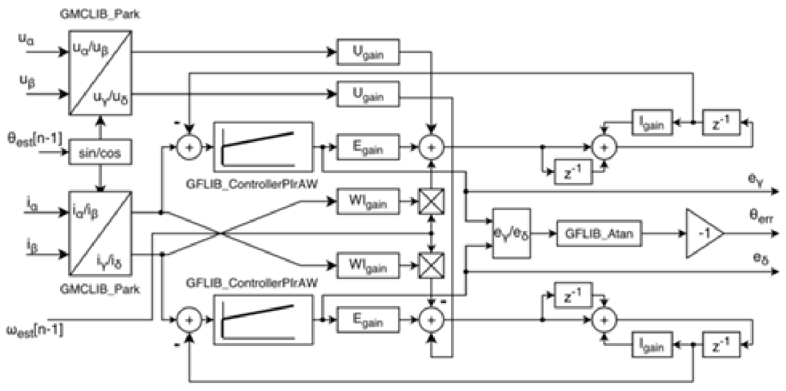


Figure 9. BEMF observer structure

4.3 Sensored control

This refence board provide hall and encoder two sensor control modes, by default it is disabled.

4.3.1 Enable sensored control

1. Connect encoder to refence board encoder Port

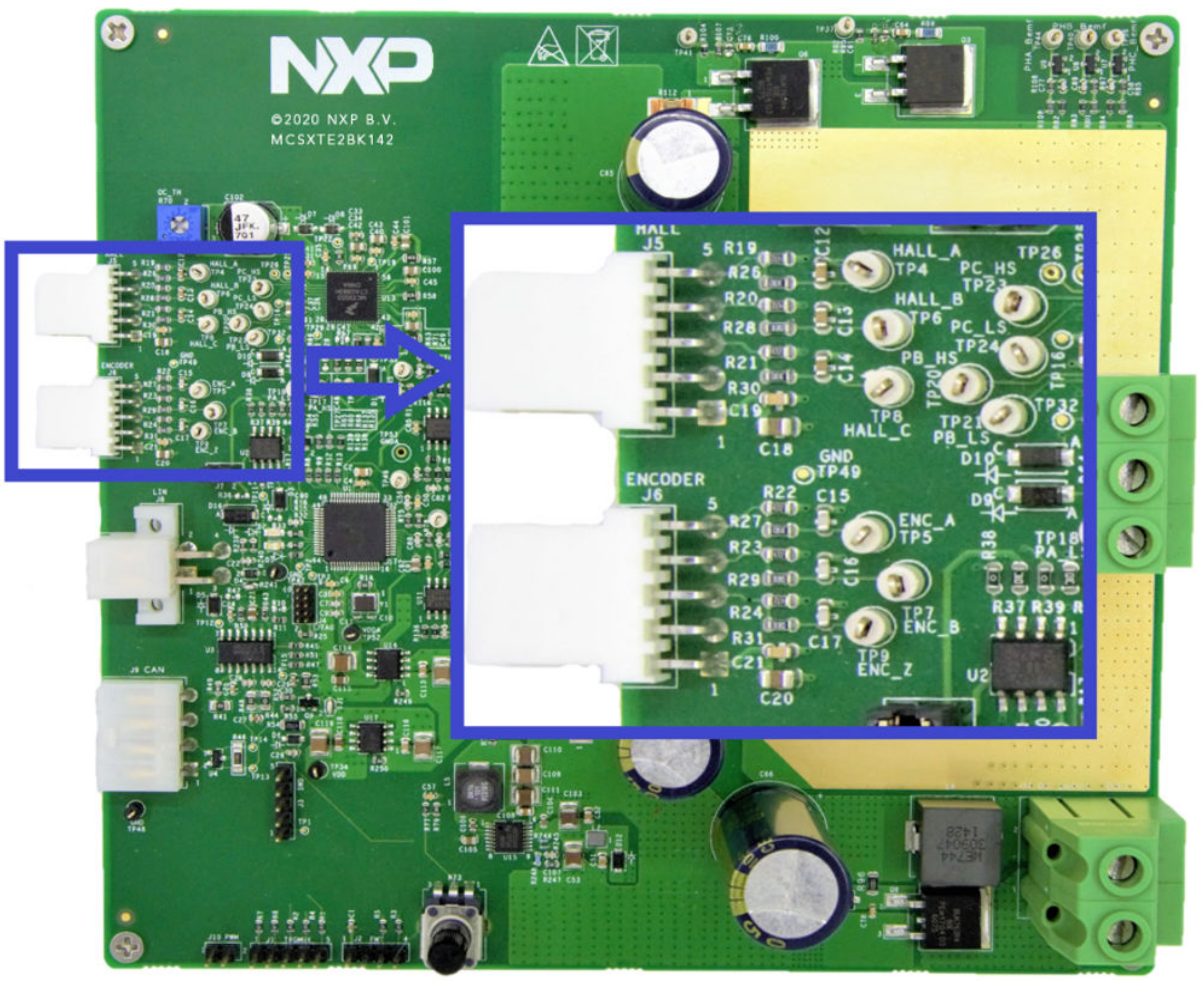


Figure 10. Sensor interface

2. Enable hall or encoder in the project: Set the macro definition “**SENSORED_OPTION**” to “**ENCODER**” or “**HALL**” (.../ Motor_Control/ Motor_Control.h) to enable hall sensor feature. This macro definition is set “SENSORLESS” by default.

   Then rebuild the project and download the S19 file to refence board MCU.

3. Select hall or encoder in FreeMASTER: Click Sensor Option select list and select hall or encoder.

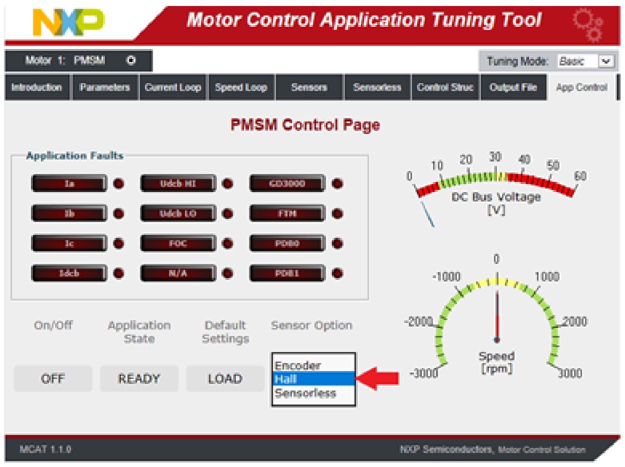


Figure 11. FreeMASTER sensored

4.3.2 Hall sensor support description

All source about hall sensor support are in source file **Hall.c** and **Hall.h**, hall sensor resource occupancy as below:

Table 2. MCU peripherals allocation for hall sensor

Resource	Name	Description
I/O port	PTD16 PTD15 PTE9	Input signal of Hall sensor
FTM	FTM1 MC mode	Calculate pulse width to get rotor speed
Interrupt	PTD16 I/O interrupt PTD15 I/O interrupt PTE9 I/O interrupt FTM1 MC overload interrupt	Get hall signal

For three phase motor control sensor-ed applications the use of Hall sensors, generally three sensors placed 120 degrees apart around the rotor, are deployed to detect position and speed. Each of the three sensors provides a pulse that applied to an input capture pin, can then be analyzed and both speed and position can be deduced.

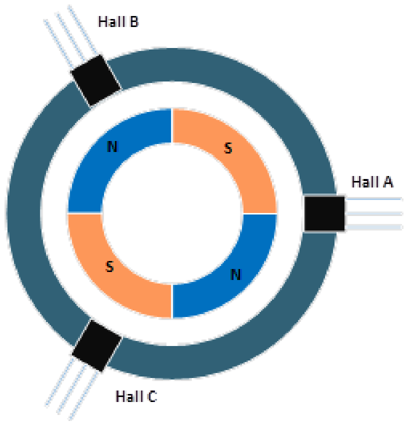


Figure 12. Hall installation diagram

If magnet south face close to the hall sensor, it output low level, and if north face close to it, output high level, when the motor run, 3 hall sensors signal as below:

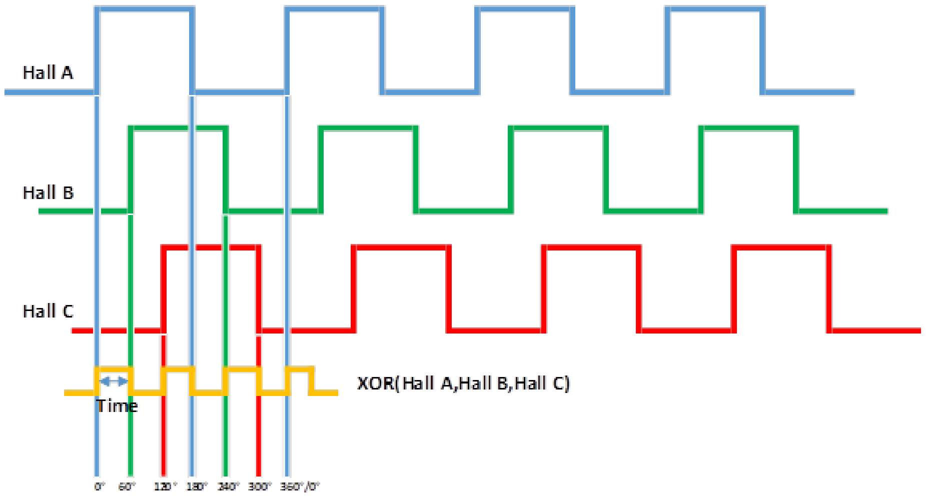


Figure 13. Hall sensor signal

To simplify the calculations required by the CPU on each hall sensor's input, if all 3 inputs are "exclusively XOR" into one timer channel and the free running counter is refreshed on every edge then this can simplify the speed calculation. S32K1xx provide hall sensor hardware XOR feature (refer to S32 RM to get more information).

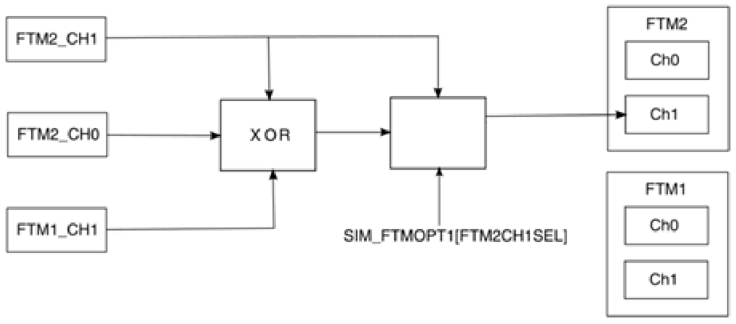


Figure 14. S32K1xx hall support

But in this reference design project, we process the hall sensor signal by software.

Following are the software processing steps:

1. Detect hall pins interrupt (Rising and falling edges), when it happened, read hall pins state.
2. Calculate the current position and speed of the rotor based on the status, the relationship between the hall sensor output state and the rotor position is as follows.

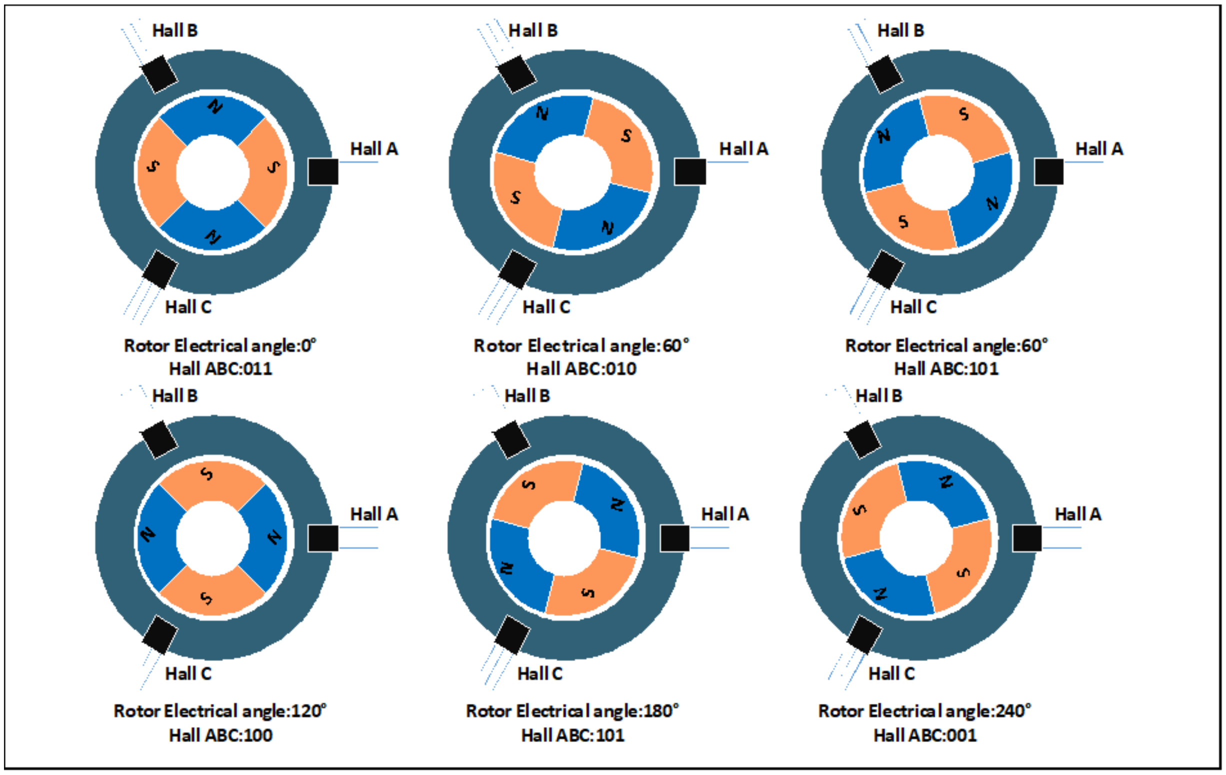

Hall B
Hall A
Hall C
Rotor Electrical angle:0°
Hall ABC:011
Rotor Electrical angle:60°
Hall ABC:010
Rotor Electrical angle:60°
Hall ABC:101
Rotor Electrical angle:120°
Hall ABC:100
Rotor Electrical angle:180°
Hall ABC:101
Rotor Electrical angle:240°
Hall ABC:001
N
S

Figure 15. Hall output and rotor position

With only the hall sensor used to calculate the rotor position, it can just achieve a resolution of 60 degrees. so, we need speed integration to get a more accurate rotor position.

$\text{Position}_{(\text{now})} = \text{Position}_{(\text{last hall position})} + \Sigma\ \text{Speed}$

However, the integral introduces an error, so it is necessary to make an error correction after obtaining a precise position by the every 60°.

Calculate the speed of the rotor by calculating the time of each interruption, each interruption indicates that the rotor is running 60 degrees.

Speed = 60°/ Time

In each sector, the accurate rotor position is calculated by speed integral, and then the integral error is eliminated every time the Hall signal comes.

4.3.3 Encoder support description

The FTM module offers a Quadrature decoder mode to decode the quadrature signals generated by rotary sensors used in motor control domain. This mode is used to process encoder signals and determine rotor position and speed. There are three output signals generated by incremental encoder as shown in the below figure. Phase A and Phase B signals consist of a series of pulses which are phase-shifted by 90° (therefore the term “quadrature” is used). The third signal (called “Index”) provides the absolute position information. In the motion control, it is used to check the pulse-counting consistency.

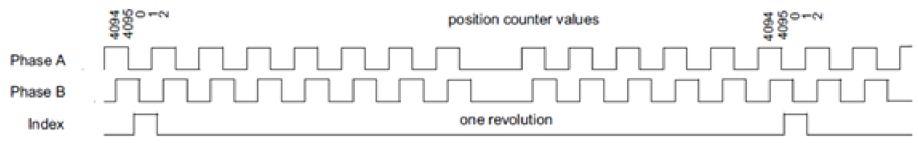
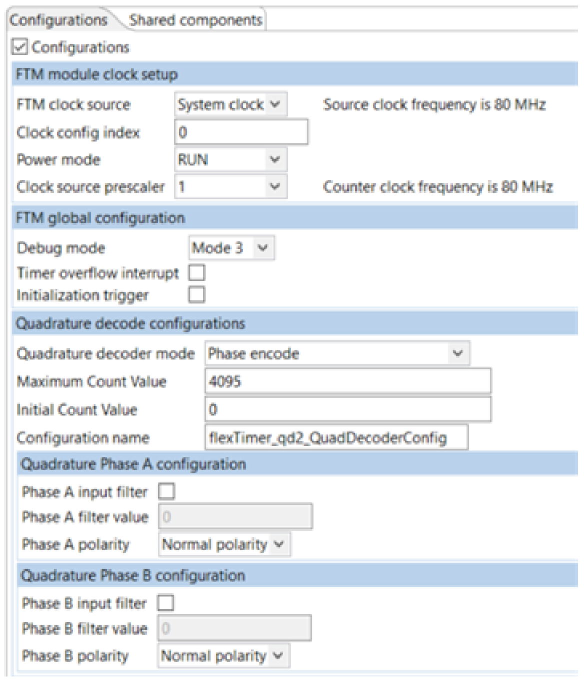


Figure 16. Output signals of the 1024 pulses encoder

To process the Phase A and Phase B signals from the encoder sensor, Quadrature decoder mode with Phase encode mode have to be enabled in Processor Expert, S32K14x FTM2 configuration in processor expert. In addition, Maximum Count Value has to be set according to the number of the encoder edges. In Quadrature decoder mode, the Phase A and Phase B signals indicate the counting direction as well as the counting rate. If the Phase B signal lags the Phase A signal, the FTM2 counter increases after every detected rising/falling edge of both signals. If the Phase B signal leads the Phase A signal, the FTM2 counter decreases after every detected rising/falling edge of both signals and the QUADIR bit in the FTM_QDCTRL register indicates the counting direction.



☑ Configurations

FTM module clock setup

FTM clock source	System clock	Source clock frequency is 80 MHz
Clock config index	0	
Power mode	RUN	
Clock source prescaler	1	Counter clock frequency is 80 MHz

FTM global configuration

Debug mode	Mode 3
Timer overflow interrupt	☐
Initialization trigger	☐

Quadrature decode configurations

Quadrature decoder mode	Phase encode
Maximum Count Value	4095
Initial Count Value	0
Configuration name	flexTimer_qd2_QuadDecoderConfig

Quadrature Phase A configuration

Phase A input filter	☐
Phase A filter value	0
Phase A polarity	Normal polarity

Quadrature Phase B configuration

Phase B input filter	☐
Phase B filter value	0
Phase B polarity	Normal polarity

Figure 17. S32K14x FTM2 configuration in processor expert

Configuration structures of the Quadrature decoder mode generated by Processor Expert are shown in S32K14x FTM2 configuration in processor expert.

All code that calculates the rotor angle and speed through the encoder is in the file **Pospe_sensor.c** and **Pospe_sensor.h**.

The API: tBool **POSPE_GetPospeElEnc**(encoderPospe_t *ptr) is used to calculate rotor speed and position. Refer to the code for more details.

4.4 Field Weakening

4.4.1 Principle of field weakening

Field weakening is an advanced control approach that extends standard FOC to allow electric motor operation beyond a base speed. The idea of field weakening control comes from the excitation control of the separately excited DC motor. In the application of the separately excited DC motor, when the terminal voltage of the motor reaches its maximum value, we can reduce the excitation current to keep voltage balance of the motor, so that the motor can operate at a higher speed with constant power. In permanent magnet synchronous motor (PMSM), when the motor speed reaches the rated speed, the synthetic flux of the motor is reduced by increasing the direct axis demagnetization current, and then the speed-up range of the motor is extended to continue to increase the speed.

Once the motor rotor rotates, the permanent magnet and the stator winding coil will move relatively, which will generate current and back electromotive force (EMF). The back electromotive force (EMF) is proportional to the rotor speed and counteracts the motor supply voltage. With the speed increase, the difference between the induced back-EMF and the supply voltage decreases, the phase current flow is limited, hence the currents id and iq cannot be controlled sufficiently. Further increase of speed would eventually result in back-EMF voltage equal to the limited stator voltage, which means a complete loss of current control. If a given speed is to be reached, the terminal voltage must be increased to match the increased stator back-EMF. A sufficient voltage is available from the inverter in the operation up to the base speed. Beyond the base speed, motor voltages ud and uq are limited and cannot be increased because of the ceiling voltage given by inverter. The only way to retain the current control even beyond the base speed is to lower the generated back-EMF by weakening the flux that links the stator winding.

Base speed defines the rotor speed at which the back-EMF reaches maximal value and motor still produces the maximal torque. Base speed splits the whole speed motor operation into two regions: constant torque and constant power, see the following figure.

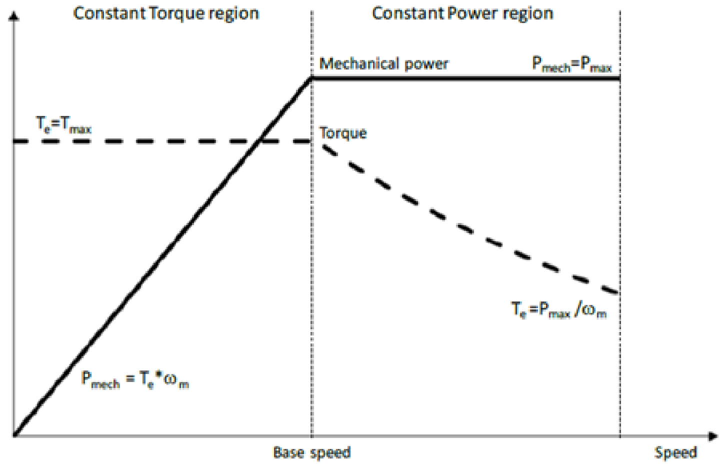


Figure 18. Constant torque/power operating regions

To know more about field weakening, refer to AN12235.

Field Weakening (FW) is as a part of the NXP’s Automotive Math and Motor Control library.

This algorithm is protected by US Patent No. US 2011/0050152 A1.

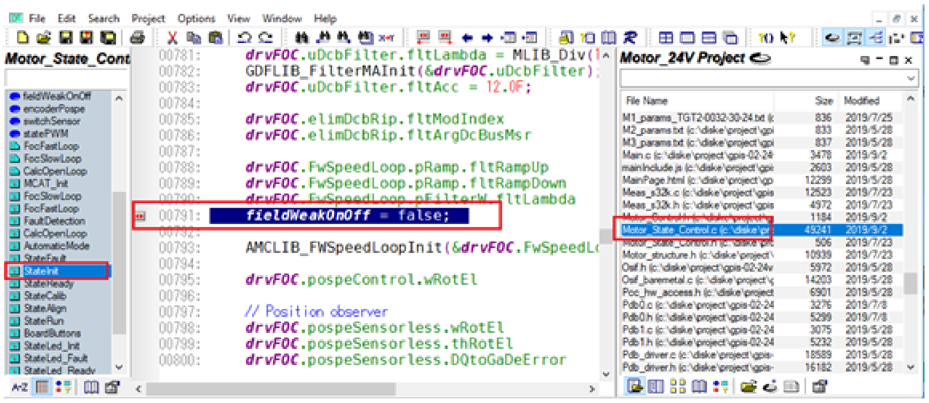
4.4.2 How to enable field weakening function

In our demo project, this function is disabled in default. You have two ways to enable it. If you want to enable it all the time, you can modify the code directly; if you just want to enable it temporarily, you can enable it in the tuning tool (MCAT), this method only keeps in current system cycle, which means you need re-enable it after system reset.

Method 1:

Search the variable in function **StateInit()** of the project code: **fieldWeakOnOff**.

In this function, we set it to false in default, change the initial value to true, then recompiling project.

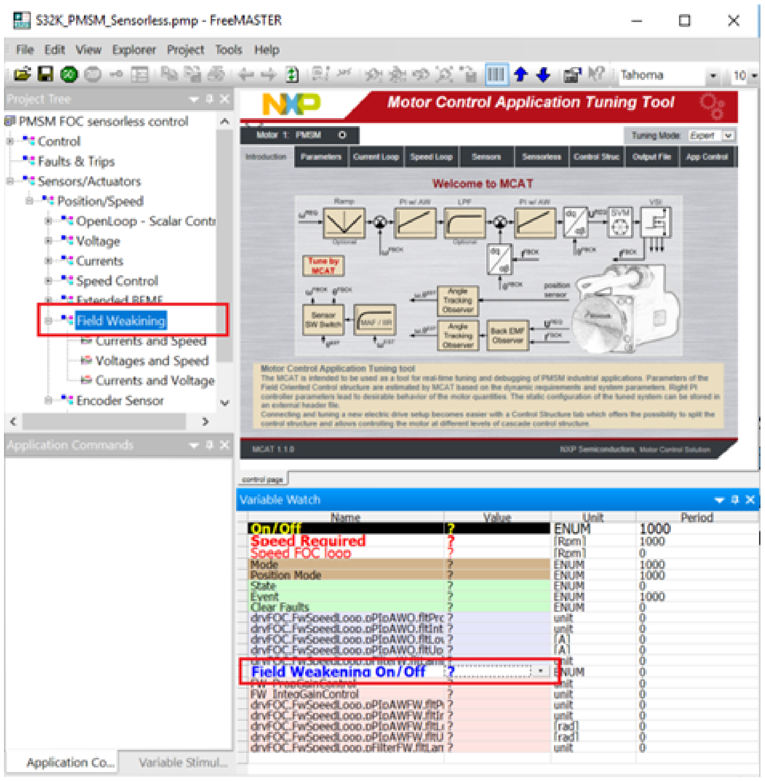


```
drvFOC.uDcbFilter.fltLambda = MLIB_Div(1
GDFLIB_FilterMAInit(&drvFOC.uDcbFilter);
drvFOC.uDcbFilter.fltAcc = 12.0F;

drvFOC.elimDcbRip.fltModIndex
drvFOC.elimDcbRip.fltArgDcBusMsr

drvFOC.FwSpeedLoop.pRamp.fltRampUp
drvFOC.FwSpeedLoop.pRamp.fltRampDown
drvFOC.FwSpeedLoop.pFilterW.fltLambda
fieldWeakOnOff = false;

AMCLIB_FWSpeedLoopInit(&drvFOC.FwSpeedL

drvFOC.pospeControl.wRotEl

// Position observer
drvFOC.pospeSensorless.wRotEl
drvFOC.pospeSensorless.thRotEl
drvFOC.pospeSensorless.DQtoGaDeError
```

Figure 19. Field weakening variable switch

Method 2:

Go into "**FreeMASTER_control**" folder which under project's root path. Double click "**S32K_PMSM_Sensorless.pmp**" file to open FreeMASTER/MCAT. And then running your system (please reference the previous chapter), when the speed reaches the nominal speed, then open the window which showed in following picture. Set the switch value of "Field Weakening On/Off" as true. Then set your target speed which higher than nominal speed. Commonly, we use this method to test whether your motor suit for doing field weakening function. In fact, you also can set the switch value as true firstly before you start motor.



Name	Value	Unit	Period
On/Off	?	ENUM	1000
Speed Required	?	[Rpm]	1000
Speed FOC loop	?	[Rpm]	0
Mode	?	ENUM	1000
Position Mode	?	ENUM	1000
State	?	ENUM	0
Event	?	ENUM	1000
Clear Faults	?	ENUM	0
drvFOC.FwSpeedLoop.pPIpAWQ.fltPrc	?	unit	0
drvFOC.FwSpeedLoop.pPIpAWQ.fltInt	?	unit	0
drvFOC.FwSpeedLoop.pPIpAWQ.fltLo	?	[A]	0
drvFOC.FwSpeedLoop.pPIpAWQ.fltUp	?	[A]	0
drvFOC.FwSpeedLoop.pFilterW.fltLam	?	unit	0
Field Weakening On/Off	?	ENUM	0
FW_PropGainControl	?	unit	0
FW_IntegGainControl	?	unit	0
drvFOC.FwSpeedLoop.pPIpAWFW.fltPr	?	unit	0
drvFOC.FwSpeedLoop.pPIpAWFW.fltIr	?	unit	0
drvFOC.FwSpeedLoop.pPIpAWFW.fltL	?	[rad]	0
drvFOC.FwSpeedLoop.pPIpAWFW.fltU	?	[rad]	0
drvFOC.FwSpeedLoop.pFilterFW.fltLan	?	unit	0

Figure 20. Field weakening GUI switch

4.4.3 Field weakening function related API

In Motor Control library, about the main function, we have two API related with field weakening function. One is **AMCLIB_FW**, and another is **AMCLIB_FWSpeedLoop**. But AMCLIB_FW is intended to be used in combination with **AMCLIB_SpeedLoop**. The code can be simplified further by utilizing **AMCLIB_FWSpeedLoop** which combines the speed controller with the field weakening controller in one library function. So we are not intend to introduce the first API (In our demo project, we used the second API directly). Besides the main function, we have an initial function (**AMCLIB_FWSpeedLoopInit**), and before you call the main function, you need call the initial function first. We also supply a function (**AMCLIB_FWSpeedLoopSetState**) to obtain the state value.

In the library, each API supply three different types to match different environment requirement.

- **AMCLIB_Xxx_F32**: 32-bit fixed-point
- **AMCLIB_Xxx_F16**: 16-bit fixed-point
- **AMCLIB_Xxx_FLT**: hardware single-point float

In our project, S32K142 uses ARM Cortex-M4F CPU core with hardware single-point FPU, the third type is applied to get the maximum performance. About the difference of these types, please refer the library document [4]).

4.4.3.1 AMCLIB_FWSpeedLoopInit

This function clears the AMCLIB_FWSpeedLoop state variables.

Declaration:

void AMCLIB_FWSpeedLoopInit_FLT(AMCLIB_FW_SPEED_LOOP_T_FLT *const pCtrl);

Arguments:

Table 3. Arguments of AMCLIB_FWSpeedLoopInit_FLT

Type	Name	Direction	Description
AMCLIB_FW_SPEED_LOOP_T_FLT *const	pCtrl	input, output	Pointer to the structure with AMCLIB_FWSpeedLoop state.

4.4.3.2 AMCLIB_FWSpeedLoop

This function implements the speed PI controller and the field weakening controller in the outer control loop highlighted in the following figure. It combines the functionalities of AMCLIB_FW and AMCLIB_SpeedLoop in a more integrated form to simplify the application code and improve execution speed.

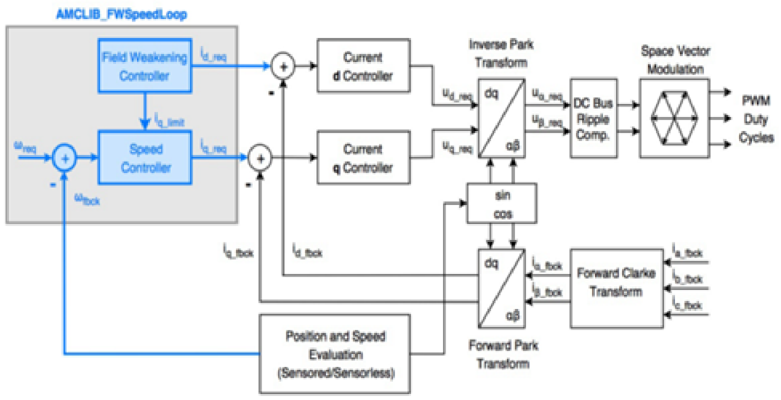


Figure 21. Principle field oriented control scheme with AMCLIB_FWSpeedLoop

Declaration:

void AMCLIB_FWSpeedLoop_FLT(tFloat fltVelocityReq,

tFloat fltVelocityFbck,

SWLIBS_2Syst_FLT *const pIDQReq, AMCLIB_FW_SPEED_LOOP_T_FLT *pCtrl);

Arguments:

Table 4. Arguments of AMCLIB_FWSpeedLoop

Type	Name	Direction	Description
tFloat	fltVelocityReq	input	Required electrical angular velocity (setpoint).
tFloat	fltVelocityFbck	input	Actual electrical angular velocity from the feedback.
SWLIBS_2Syst_FLT *const	pIDQReq	input, output	Pointer to the structure with the required stator currents in the two-phase rotational orthogonal system (d-q).
AMCLIB_FW_SPEED_ LOOP_T_FLT *	pCtrl	input, output	Pointer to the structure with AMCLIB_FWSpeedLoop state.

4.4.3.3 AMCLIB_FWSpeedLoopSetState

This function initializes the AMCLIB_FWSpeedLoop state variables to achieve the required output values.

Declaration:

void AMCLIB_FWSpeedLoopSetState_FLT(tFloat fltFilterMAWOut,

tFloat fltFilterMAFWOut,

tFloat fltControllerPIpAWQOut,

tFloat fltControllerPIpAWFWOut,

tFloat fltRampOut,

AMCLIB_FW_SPEED_LOOP_T_FLT *pCtrl);

Arguments:

Table 5. Arguments of AMCLIB_FWSpeedLoopSetState

Type	Name	Direction	Description
tFloat	fltFilterMAWOut	input	Required output of the speed FilterMA.
tFloat	fltFilterMAFWOut	input	Required output of the field-weakening FilterMA.
tFloat	fltControllerPIpAWQOut	input	Required output of the speed ControllerPIpAW.
tFloat	fltControllerPIpAWFWOut	input	Required output of the field-weakening ControllerPIpAW.
tFloat	fltRampOut	input	Required output of the speed ramp.
AMCLIB_FW_ SPEED_LOOP_T_FLT *	pCtrl	input, output	Pointer to the structure with AMCLIB_ FWSpeedLoop state.

4.5 Stall detection implementation

In PMSM sensor-less application, motor will stall when the load become very large or rotor is stuck by something or the load change dramatically. Usually it will trigger overcurrent protection, but sometime the motor phase current is not very big when motor is in stall condition. Meanwhile, sensor-less algorithm may still work, it can generate speed and angel regularly. This "fake running" should be detected to avoid any harm to the system. The stall detection method needs to be adopted to achieve the task.

4.5.1 Stall detection principle

There are several methods can do stall detection, NXP are using BEMF consistency checking method to detect the stall case.

Usually, the BEMF of PMSM are linear with motor speed. Actually, BEMF of observer output should be consistent with motor K_e multiply motor speed and plus the offset. This is shown in the following equation.

$$E_q = K_e \cdot \omega + Ke_{offset}$$

Figure 22. Equation 2

E_q: BEMF output of observer

K_e: BEMF coefficiency of the motor

ω: Motor speed

Ke_{offset}: BEMF offset

So principle of the method is check the consistency of two BEMFs. If observer output Eq is not linear with motor speed, that means observer is not work correctly and indicate the motor is in stall.

The following diagram can show the principle also.

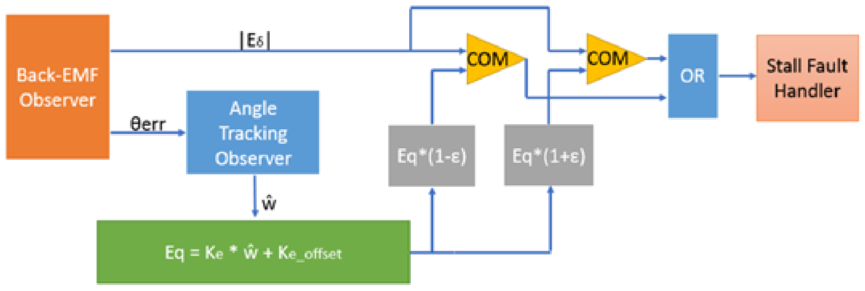


Figure 23. Stall detect principle

4.5.2 Implementation

1. The method is based on module design, so it's easy to implement in your own project.
   - Firstly, copy the **stallDetection**.c and .h file in your project;
   - Secondly, define the variable of structure, for example, stallDetection_T stallDetectionPsrams;
   - Thirdly, initial the stall detection function, **tallDetectionInit** (&stallDetectionPsrams);
   - Last, add the stall detection function in **StateRun** function.

```
if (TRUE == stallDetection(&stallDetectionPsrams))
{
    permFaults.motor.B.StallError = 1;
}
```

2. Stall detection parameters

The parameters should configure correctly to make sure the stall detection function can work correctly.

There are some macros to configure the function, list all the macros as below:

```
#define    STALLDETECTION_BLANKCNT         20000
#define    STALLDETECTION_CHKCNT           30
#define    STALLDETECTION_CHKERRCNT
 STALLDETECTION_CHKCNT-5
#define    STALLDETECTION_COEFFL            0.75F
#define    STALLDETECTION_COEFFH           2.0F-
STALLDETECTION_COEFFL
#define    STALLDETECTION_COEFFKE          0.01672F
#define    STALLDETECTION_COEFFKEOFT        -0.3
#define    BEMFOBSFILTER_LAMBDA            1.0F
#define    ROTELFILTER_LAMBDA              1.0F
```

"STALLDETECTION_BLANKCNT " is used to set a blank time slot, in this slot, system will not do BEMF checking. The method is based on BEMF checking, so it's not applicable in startup stage.

"STALLDETECTION_CHKCNT " and "STALLDETECTION_CHKERRCNT " mean the check time and the error time. The allowable error check time is the check time minus 5. in the default setting, check 30 time, if the error time bigger than 25, it will trigger the stall fault. Off course, the user can change the checking time and the error checking time according to your application.

"STALLDETECTION_COEFFL" and "STALLDETECTION_COEFFH" mean the threshold of allowable different range between observer Eq and calculated BEMF. In default setting, if the calculated BEMF is in range of 0.75*Eq and 1.25*Eq, it indicates the motor is not in stall status, but if the calculated BEMF is out of the range, meanwhile, the error checking time bigger than the setting, it will trigger the stall fault.

"STALLDETECTION_COEFFKE" and "STALLDETECTION_COEFFKEOFT " means the slope and the offset for the calculation equation. These parameters are very important and need manually offline calculation. Different type of motors may have different parameters.

For example, if get Eq = 3.2 @1000rpm and 6.7@2000rpm.

Then there are two equations 1000*a+b = 3.2 and 2000*a+b=6.7. After the calculation, a = 0.035 and b = -0.3.

For speed, it uses rad/s, not rpm, so the

$$\mathbf{Eq} = (\omega * 60/(2*\mathrm{PI}*\mathrm{PP})) * 0.0035 - 0.3.$$

Figure 24. Equation 3

PP is pole pairs and equal to 2.

Make it simple and get the result

$$\mathbf{Eq} = \omega * 0.1672 - 0.3.$$

Figure 25. Equation 4

"BEMFOBSFILTER_LAMBDA" means BEMF observer output MA filter coefficient. The range is from 0 to 1.0F. Bigger, filter less. So 1.0F mean no filter influence.

"ROTELFILTER_LAMBDA" means speed ω MA Filters.Bigger, filter less. So 1.0F mean no filter influence.

4.5.3 Stall detection summary and enable

The BEMF consistency checking method is based on NXP patent **US20170126153A1**. The developer can check the original patent for more information.

In the demo project, this function disabled in default, if you want to enable it, just change the macro: **STALL_DETECTION** value from **STD_OFF** to **STD_ON** in Motor_Control.h.

4.6 Current sampling method

4.6.1 Overview

There are three current sampling methods which using shunts in inverter legs as current sensors. dual-shunt, tri-shunt and single shunt. MCSXTE2BK142 can support dual-shunt and tri-shunt methods. If you want to use single shunt on this board, the 3 x 1000 uF capacitors on three power phases (C65, C68 and C70) must be removed. This section will focus on dual-shunt and tri-shunt, for singles shunt solution, refer to AN12235 and AN5327.

In the software package, three separate demo projects for different FOC methods are provided:

1. Single-shunt FOC: MCSXTE2BK142_PMSM_FOC_1Sh
2. Dual-shunt FOC: MCSXTE2BK142_PMSM_FOC_2Sh
3. Triple-shunt FOC: MCSXTE2BK142_PMSM_FOC_3Sh

4.6.2 Dual-shunt current sampling

1. Dual-shunt current sampling is the most popular method due to the best performance vs cost. Main stream microcontrollers have 2 ADC modules and it's perfect to get two current sampling at one shot. The third shunt is optional to make the circuit balance. The following figure shows the topology.

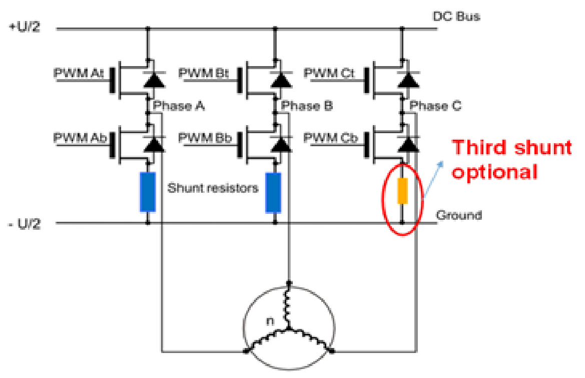


Figure 26. Shunt resistors topology

2. When all the bottom MOSFET are ON, the current free running in the motor, the voltage of shunts can indicate the motor phase currents.

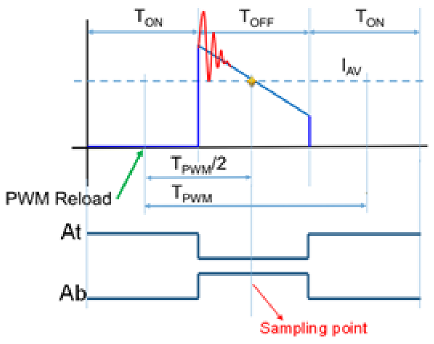


Figure 27. Sampling theory diagram

3. For dual-shunt sampling, the sampling point is no need change, the method is much easy to achieve. The phase currents obtained in the same time, so reconstruct phase current THD is low.

4. But if SVPWM waveform is shown as following phase II, there is a concept "minimal pulse width" which rely on hardware design. If the available duty is too short, the current sampling for Phase A would be bad quality. To avoid this case, the duty cycle limitation should be applied. 0.9 is the default value, for higher quality hardware, it can be set higher, for instance, 0.95 or more.

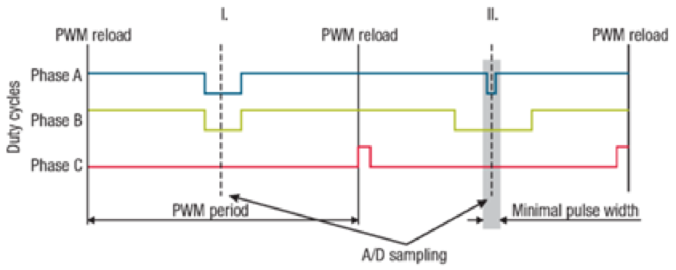


Figure 28. PWM diagram when sampling

4.6.3 Tri-shunt current sampling

Actually, S32K14x only have two ADC modules, so it cannot achieve real tri-shunt current sampling the same time. But, FOC only need dual-shunt currents, and then using $I_A+I_B+I_C=0$ to get the third phase current. The question is why need the tri-shunt?

As mentioned in the previous sections, there are limitation in dual-shunt current sampling. If one of the three phases duty cycle is very high, the bottom ON time is very short, the window for current sampling is also very short. The user needs to use vector limitation to limit the short duty cycle. That also means you cannot fully make use of DC bus voltage. Tri-shunt current sampling is intent to solve the issue.

The principle is that sampling strategy finds the most comfortable two phase to sample. The sampling is not fix to certain two phases, but dynamic selection choose two phases to sample according to the voltage sectors. In other words, the tri-shunt current sampling is just dual-shunt sampling in different periods.

1. Tri-shunt current sampling implementation: The following figure shows tri-shunt current sampling.

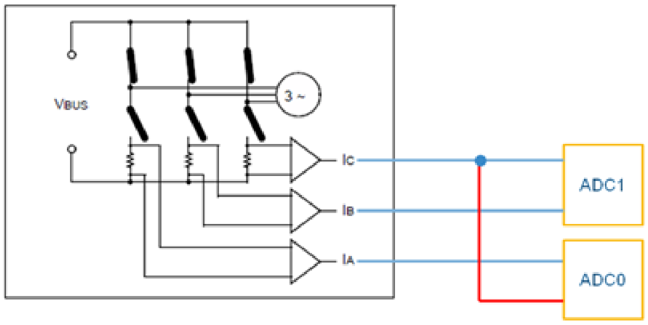


Figure 29. Tri-shunt current sampling diagram

I_A is sampled by ADC0 and I_B is sampled by ADC1. For I_C, it should be connected with ADC0 and ADC1. With the help of the ADC interleave function it can be easily achieved.

MCSXTE2BK142 uses PTB13 to sample I_C and mapped to ADC0 channel 8 and ADC1 channel 8. It is highlighted as bold green line in the following figure.

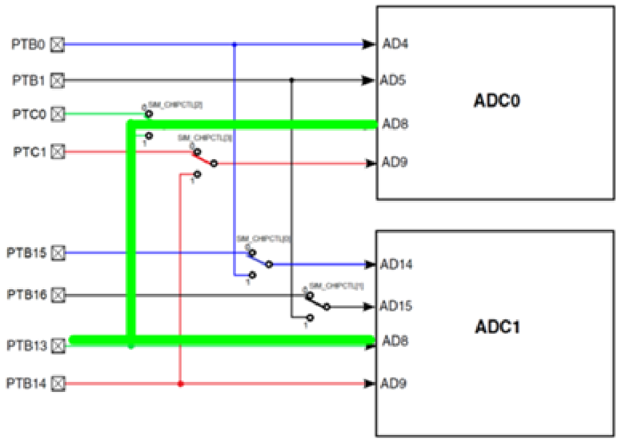


Figure 30. ADC interleave function

2. Software consideration: For tri-shunt current sampling, the different in software mostly focus on current sampling. Especially in meas_s32K.c file. Software will get the two phase currents result according to the voltage sectors. Meanwhile, software changes the sampling channels also according the sector and hardware configuration. MEAS_CurrChanAssign function contains further information.

   For "MEAS_CalibCurrentSense" function, the software needs to consider and sample three currents and get the average value as calibration result.

3. Tri-shunt pros and cons: The best benefit for tri-shunt is the higher DC bus voltage usage, that means motor can run at higher speed or higher power range. The drawback of tri-shunt obviously is the high cost and a little bit higher CPU load compare with dual-shunt.

4.6.4 Summary

There is no strict rule for selection of dual-shunt or tri-shunt current sampling. But usually, if the motor power range is high, it is strongly recommend to use tri-shunt method due to the higher DC bus voltage usage.

Chapter 5
BSP layer module and API introduction

To reduce user time on Low Level Driver (LLD) and focus on application development, in the board SW package a BSP layer is included to provide the initialize and operate API for all onboard module/interface and used MCU peripherals.

BSP.h, all used LED control, hall/encoder inputs, GD3000 IRQ inputs GPIO pin, and interrupts IRQ number are defined. All BSP module API header files are also included.

BSP.c, provides the following API for application:

- void **BSP_Init**(void): it calls the BSP layer module initialize API and SDK components API to complete initializations of the following module
- System clock
- CPU cache
- Power mode
- GPIO pin function mux
- Hall sensor and Encoder input and interrupt (per macro **SENSORED_OPTION** to select, which is disabled to use sensorless FOC by default)
- LPUART used by FreeMASTER communication
- ADC and PDB for phase and DC bus voltage/current sample for FOC
- FlexTimer PWM
- CAN/LIN/PWM communication
- MC33GD3000 three-phase gate driver

If user wants to use the BSP layer API, user just needs to include the **BSP.h** header file in the application.

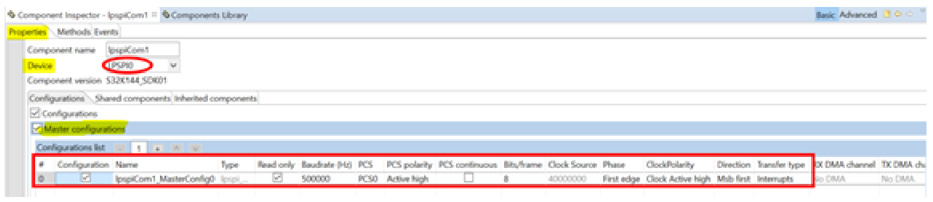
```
#include "BSP.h"
```

5.1 GD3000 gate driver LLD

The GD3000 gate driver LLD is based on S32K1xx SDK **lpspi** and **PinSettings** PD component for configuration and diagnostic communication and interrupt handle.

5.1.1 Processor expert configuration for GD3000 gate driver

MC33GD3000 gate pre-driver is connected to S32K142 via LPSPI0 as a slave device, the SDK **lpspi** PD component configuration is as below.

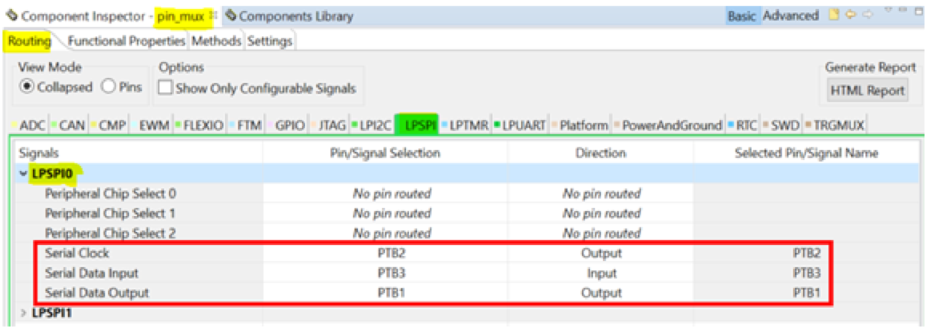


Component name: lpspiCom1
Device: LPSPI0
Component version: S32K144_SDK01
Configurations ✓
Master configurations ✓

#	Configuration	Name	Type	Read only	Baudrate (Hz)	PCS	PCS polarity	PCS continuous	Bits/frame	Clock Source	Phase	ClockPolarity	Direction	Transfer type	RX DMA channel	TX DMA ch
0	✓	lpspiCom1_MasterConfig0	lpspi_...	✓	500000	PCS0	Active high	☐	8	40000000	First edge	Clock Active high	Msb first	Interrupts	No DMA	No DMA

Figure 31. lpspi component configuration

The LPSPI0 pin mux configured as below in **PinSettings** PD component, its Chip Select (PTB0) is controlled by software, so it's not allocated in LPSPI0.



Component Inspector - pin_mux | Components Library | Basic | Advanced

Routing | Functional Properties | Methods | Settings

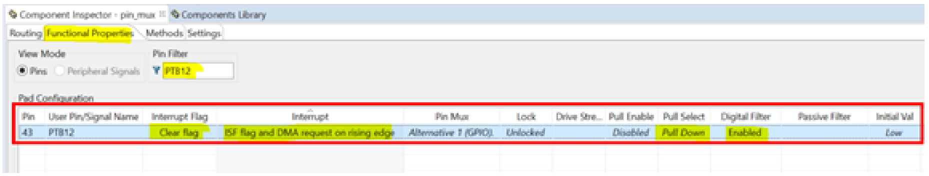
View Mode: [✓ Collapsed] Pins — Options: Show Only Configurable Signals — Generate Report: HTML Report

ADC | CAN | CMP | EWM | FLEXIO | FTM | GPIO | JTAG | LPI2C | LPSPI | LPTMR | LPUART | Platform | PowerAndGround | RTC | SWD | TRGMUX

Signals	Pin/Signal Selection	Direction	Selected Pin/Signal Name
LPSPI0			
Peripheral Chip Select 0	*No pin routed*	*No pin routed*	
Peripheral Chip Select 1	*No pin routed*	*No pin routed*	
Peripheral Chip Select 2	*No pin routed*	*No pin routed*	
Serial Clock	PTB2	Output	PTB2
Serial Data Input	PTB3	Input	PTB3
Serial Data Output	PTB1	Output	PTB1
LPSPI1			

Figure 32. PinSettings component configuration for LPSPI

Besides, the GD3000 gate driver LLD also uses the S32K142 GPIO IRQ rising edge interrupt to capture its event interrupts.



Component Inspector - pin_mux | Components Library

Routing | Functional Properties | Methods | Settings

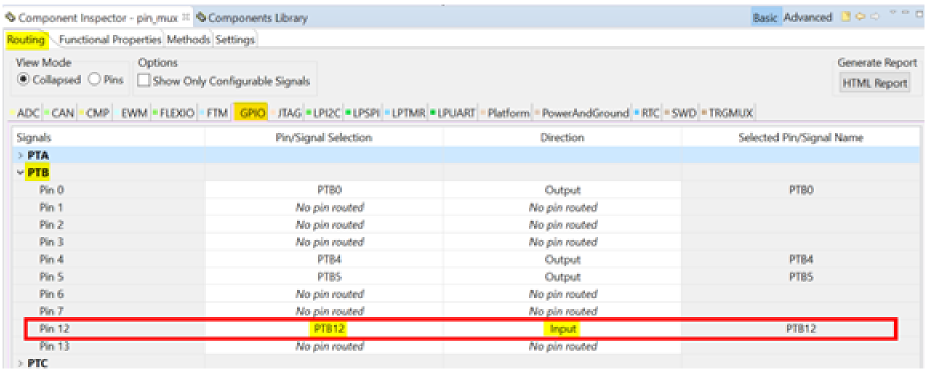
View Mode: [✓ Pins] Peripheral Signals — Pin Filter: PTB12

Pad Configuration

Pin	User Pin/Signal Name	Interrupt Flag	Interrupt	Pin Mux	Lock	Drive Stre...	Pull Enable	Pull Select	Digital Filter	Passive Filter	Initial Val
43	PTB12	Clear flag	ISF flag and DMA request on rising edge	*Alternative 1 (GPIO).*	*Unlocked*		*Disabled*	*Pull Down*	Enabled		*Low*

Figure 33. PinSettings component configuration for GD3000 IRQ interrupt

Configure PTB12 as GPIO input.



Component Inspector - pin_mux | Components Library | Basic | Advanced

Routing | Functional Properties | Methods | Settings

View Mode: [✓ Collapsed] Pins — Options: Show Only Configurable Signals — Generate Report: HTML Report

ADC | CAN | CMP | EWM | FLEXIO | FTM | GPIO | JTAG | LPI2C | LPSPI | LPTMR | LPUART | Platform | PowerAndGround | RTC | SWD | TRGMUX

Signals	Pin/Signal Selection	Direction	Selected Pin/Signal Name
PTA			
PTB			
Pin 0	PTB0	Output	PTB0
Pin 1	*No pin routed*	*No pin routed*	
Pin 2	*No pin routed*	*No pin routed*	
Pin 3	*No pin routed*	*No pin routed*	
Pin 4	PTB4	Output	PTB4
Pin 5	PTB5	Output	PTB5
Pin 6	*No pin routed*	*No pin routed*	
Pin 7	*No pin routed*	*No pin routed*	
Pin 12	PTB12	Input	PTB12
Pin 13	*No pin routed*	*No pin routed*	
PTC			

Figure 34. PinSettings component configuration for PTB12 as GPIO input

5.1.2 LLD API of GD3000 gate driver

The GD3000 gate driver LLD is based on NXP analog middleware with aml and tpp layer.

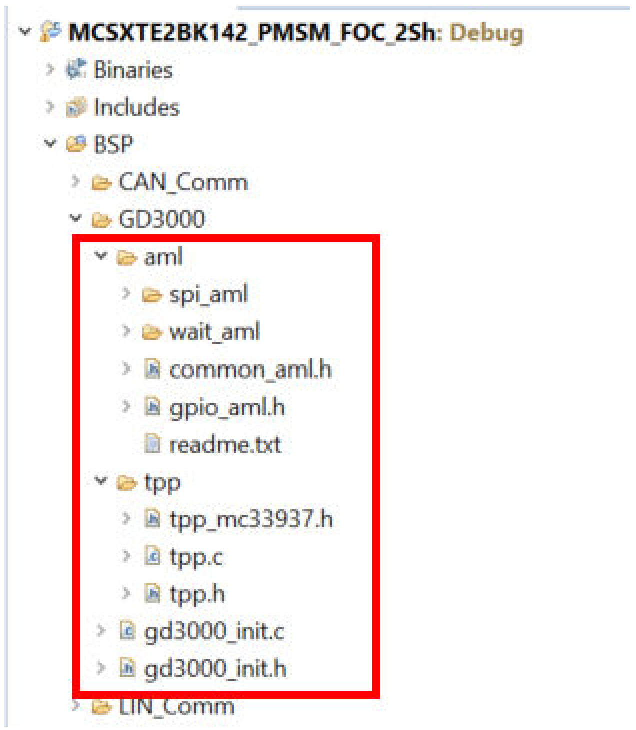
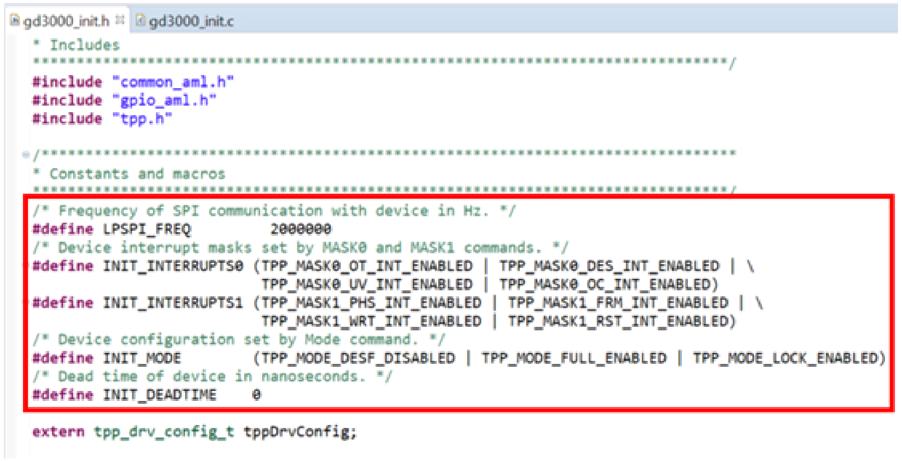


Figure 35. GD3000 LLD code structure

The main API for user to call in application are as below:

- void **GD3000_Init**(void): initialize MC33GD3000 MOSFET pre-driver. MC33GD3000 SW driver uses S32K14x LPSPI0 module as a communication interface to configure MC33GD3000 operation mode and to track MC33GD3000 Status0/ Status1 registers.

  User can change the GD3000 configuration by modifying the macros definition in **gd3000_init.h**.



```
gd3000_init.h    gd3000_init.c
 * Includes
 ******************************************************************************/
#include "common_aml.h"
#include "gpio_aml.h"
#include "tpp.h"

/*******************************************************************************
 * Constants and macros
 ******************************************************************************/
/* Frequency of SPI communication with device in Hz. */
#define LPSPI_FREQ         2000000
/* Device interrupt masks set by MASK0 and MASK1 commands. */
#define INIT_INTERRUPTS0 (TPP_MASK0_OT_INT_ENABLED | TPP_MASK0_DES_INT_ENABLED | \
                          TPP_MASK0_UV_INT_ENABLED | TPP_MASK0_OC_INT_ENABLED)
#define INIT_INTERRUPTS1 (TPP_MASK1_PHS_INT_ENABLED | TPP_MASK1_FRM_INT_ENABLED | \
                          TPP_MASK1_WRT_INT_ENABLED | TPP_MASK1_RST_INT_ENABLED)
/* Device configuration set by Mode command. */
#define INIT_MODE        (TPP_MODE_DESF_DISABLED | TPP_MODE_FULL_ENABLED | TPP_MODE_LOCK_ENABLED)
/* Dead time of device in nanoseconds. */
#define INIT_DEADTIME    0

extern tpp_drv_config_t tppDrvConfig;
```

Figure 36. macros to configure LPSI and GD3000 interrupt

The API's in TPP layer which can be used by application are:

- status_t **TPP_ConfigureGpio**(tpp_drv_config_t* const drvConfig): This function initializes GPIO (EN1, EN2, RST pins);
- status_t **TPP_ConfigureSpi**(tpp_drv_config_t* const drvConfig, spi_sdk_master_config_t* const spiSdkMasterConfig): This function configures SPI for usage with this driver.

- status_t **TPP_Init**(tpp_drv_config_t* const initConfig, tpp_device_mode_t mode) : This function initializes the device. Configures device mode and interrupts. Prepares low side and high side output stages.

> NOTE
> Before each call of **TPP_Init()** method **TPP_Deinit()** method must be called except first initialization.

- status_t **TPP_Deinit**(tpp_drv_config_t* const drvConfig): This function deinitializes the device.

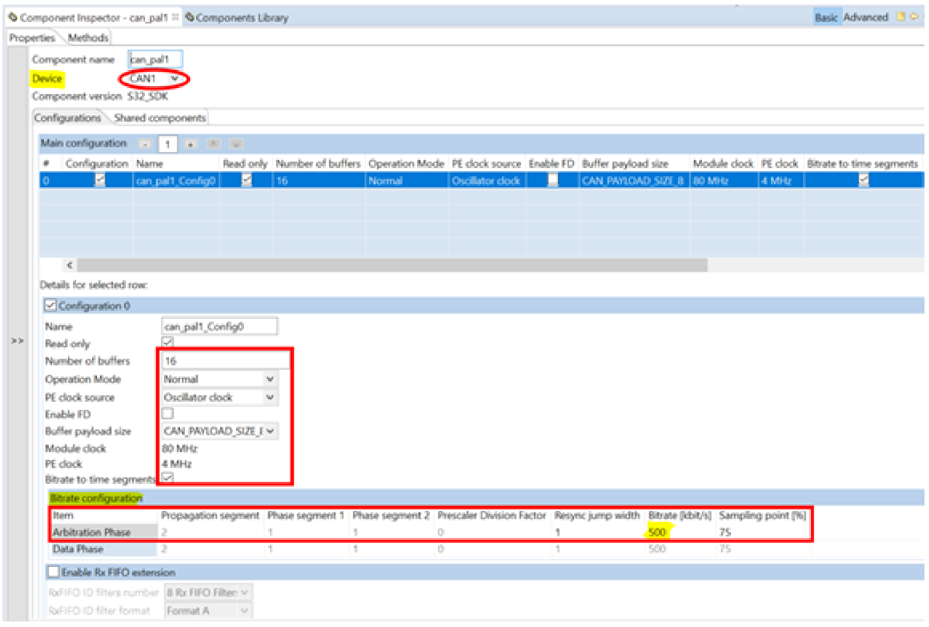
> NOTE
> The device must be in **ACTIVE** or **SLEEP** operational mode before calling this method.

- status_t **TPP_SetOperationalMode**(tpp_drv_config_t* const drvConfig, tpp_device_mode_t mode): This function sets the operational mode of the device and sets device operational mode. Internal method logic is responsible for consistency between transitions, but it is recommended to use only valid and logical transitions.
  - **SLEEP/INITIALIZATION-->ACTIVE:** Enables SPI communication, loads configuration stored in tpp_device_data_t structure.
  - **STANDBY --> ACTIVE:** Initializes and turns on output stages.
  - **FAULT_PROTECTION -> ACTIVE:** Clears all interrupt flags, initializes and turns on output stages.
  - **ACTIVE --> SLEEP:** Disables SPI communication, erases device internal configuration, but configuration is preserved in tpp_device_data_t structure.
  - **ACTIVE --> STANDBY:** Turns off output stages, preserves device internal configuration.
  - **ACTIVE --> FAULT_PROTECTION:** Can be used during interrupt processing, when output stages are automatically turned off.
- status_t **TPP_SendCommand**(tpp_drv_config_t* const drvConfig, tpp_spi_command_t cmd, uint8_t subcmd, uint8_t* rxData): This function sends command to device over SPI communication bus and receives content of device status register 0 (or another status register in case of NULL command). One of tpp_spi_command_t could be selected. The command is combined with its subcommand.
- status_t **TPP_GetStatusRegister**(tpp_drv_config_t* const drvConfig, tpp_status_register_t statusRegister, uint8_t* const rxData): This function reads selected status register of the device. It reads data from device using SPI. One status register of tpp_status_register_t could be selected. Data will be stored in buffer which is pointed by rxData.
- status_t **TPP_SetInterruptMasks**(tpp_drv_config_t* const drvConfig, uint8_t mask0, uint8_t mask1): This function sets device interrupts mask. If selected interrupt is disabled, it is still possible to check occurrence of that event by reading device status register manually. Note that it is not possible to change interrupt mask when lock mode is enabled.
- status_t **TPP_ClearInterrupts**(tpp_drv_config_t* const drvConfig, uint8_t mask0, uint8_t mask1): This function clears device interrupt flags. It is recommended to clear the flags which were actually set in device status register 0.
- status_t **TPP_SetModeRegister**(tpp_drv_config_t* const drvConfig, uint8_t modeMask): This function sets the device mode register. Note that it is not possible to change device mode settings (lock, full-on, desaturation) when lock mode is enabled.
- status_t **TPP_SetDeadtime**(tpp_drv_config_t* const drvConfig, uint16_t deadtime): This function sets deadtime value. Admissible range is from 0 to 15000 ns. If the value is set to 0, user is responsible for handling delay between low side and high side output stages toggling to prevent short. This method disables maskable interrupts for necessary time to prevent deadtime calibration interference and is available only when full-on mode is disabled. Note that it is not possible to change deadtime when lock mode is enabled. Minimal delay is one TPP predriver internal time base clock cycle duration (typically 58.82 ns for 17 MHz) * 16 = 940 ns.

5.2 CAN communication API

5.2.1 Processor expert configuration for CAN communication

The CAN communication API is based on S32K1xx SDK **can_pal** PAL component, it uses **FlexCAN1** as CAN 2.0A/B protocol with **500**Kbit/s bitrate.



Component Inspector - can_pal1 | Components Library | Basic | Advanced

Properties | Methods

Component name	can_pal1
Device	CAN1
Component version	S32_SDK

Configurations | Shared components

Main configuration: 1

#	Configuration	Name	Read only	Number of buffers	Operation Mode	PE clock source	Enable FD	Buffer payload size	Module clock	PE clock	Bitrate to time segments
0	✓	can_pal1_Config0	✓	16	Normal	Oscillator clock	☐	CAN_PAYLOAD_SIZE_8	80 MHz	4 MHz	✓

Details for selected row:

☑ Configuration 0

Name	can_pal1_Config0
Read only	✓
Number of buffers	16
Operation Mode	Normal
PE clock source	Oscillator clock
Enable FD	☐
Buffer payload size	CAN_PAYLOAD_SIZE_8
Module clock	80 MHz
PE clock	4 MHz
Bitrate to time segments	✓

Bitrate configuration

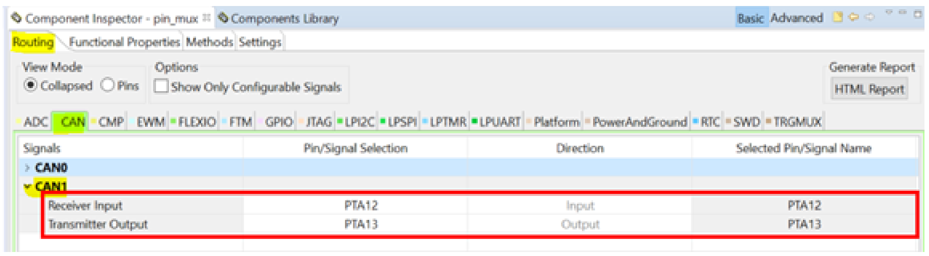
Item	Propagation segment	Phase segment 1	Phase segment 2	Prescaler Division Factor	Resync jump width	Bitrate [kbit/s]	Sampling point [%]
Arbitration Phase	2	1	1	0	1	500	75
Data Phase	2	1	1	0	1	500	75

☐ Enable Rx FIFO extension

RxFIFO ID filters number	8 Rx FIFO Filters
RxFIFO ID filter format	Format A

Figure 37. can_pal component configuration

The TXD and RXD signal route out via PTA13 and PTA12, so need to configure the **PinSettings** component as below.



Component Inspector - pin_mux | Components Library | Basic | Advanced

Routing | Functional Properties | Methods | Settings

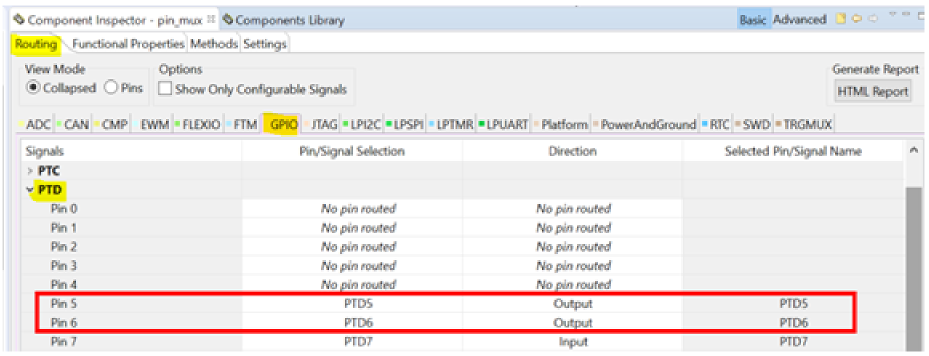
View Mode: (● Collapsed) ○ Pins — Options: ☐ Show Only Configurable Signals — Generate Report: HTML Report

ADC | CAN | CMP | EWM | FLEXIO | FTM | GPIO | JTAG | LPI2C | LPSPI | LPTMR | LPUART | Platform | PowerAndGround | RTC | SWD | TRGMUX

Signals	Pin/Signal Selection	Direction	Selected Pin/Signal Name
CAN0			
CAN1			
Receiver Input	PTA12	Input	PTA12
Transmitter Output	PTA13	Output	PTA13

Figure 38. PinSettings component configuration for FlexCAN1 TX/RX pin

In addition, PTD5 and PTD6 are configured as output to control TJA1043 enable (**EN**) and standby (**STB**).



Component Inspector - pin_mux | Components Library | Basic | Advanced

Routing | Functional Properties | Methods | Settings

View Mode: (● Collapsed) ○ Pins — Options: ☐ Show Only Configurable Signals — Generate Report: HTML Report

ADC | CAN | CMP | EWM | FLEXIO | FTM | GPIO | JTAG | LPI2C | LPSPI | LPTMR | LPUART | Platform | PowerAndGround | RTC | SWD | TRGMUX

Signals	Pin/Signal Selection	Direction	Selected Pin/Signal Name
PTC			
PTD			
Pin 0	*No pin routed*	*No pin routed*	
Pin 1	*No pin routed*	*No pin routed*	
Pin 2	*No pin routed*	*No pin routed*	
Pin 3	*No pin routed*	*No pin routed*	
Pin 4	*No pin routed*	*No pin routed*	
Pin 5	PTD5	Output	PTD5
Pin 6	PTD6	Output	PTD6
Pin 7	PTD7	Input	PTD7

Figure 39. PinSettings component configuration for TJA1043 STB control pin

5.2.2 API of CAN communication

The BSP layer provides the following CAN communication API for application:

- void **CAN_Comm_Init**(void): This function initializes the CAN communication: 1) enable the CAN transceiver-TJA1043; 2) initialize the CAN_PAL driver with dedicated TX/RX MB; 3) configure the CAN interrupt priority and install its interrupt ISR callback.
- void **CAN_Transmit**(uint32_t id, uint8_t* Data,uint8_t Length): This function sends the classical CAN message with an id and Data as input buffer with Length(Byte).

For CAN message receive, user can check the CAN_RxComplete_Flag. New received data Rx_CAN_Msg_Buffer[**RX_MAX_BUF_NUM**] via Rx_CAN_Msg_Buffer_Index.

```
extern can_message_t Rx_CAN_Msg_Buffer[RX_MAX_BUF_NUM];
extern uint32_t CAN_RxComplete_Flag;
extern uint8_t Rx_CAN_Msg_Buffer_Index;
```

The new CAN message receive buffer is used for next receive in **CAN_ISR_Callback**

```
void CAN_ISR_Callback(uint32_t instance,can_event_t eventType,uint32_t ob-
jIdx,void *driverState)
{
    switch(eventType)
    {
        case CAN_EVENT_RX_COMPLETE:
            /* increase the CAN message buffer index */
            Rx_CAN_Msg_Buffer_Index++;

            if(Rx_CAN_Msg_Buffer_Index >= RX_MAX_BUF_NUM)
            {
                /* reset the CAN message buffer index */
                Rx_CAN_Msg_Buffer_Index = 0;
            }
            /* call the CAN_PAL receive API to receive new CAN message frame
with new buffer */
            CAN_Receive(&can_pal1_instance,RX_MAIL-
BOX,&Rx_CAN_Msg_Buffer[Rx_CAN_Msg_Buffer_Index]);

            /* set the flag to notice application code */
            CAN_RxComplete_Flag = 1;

            break;

        case CAN_EVENT_TX_COMPLETE:
            break;

        default:
            break;
    }
}
```

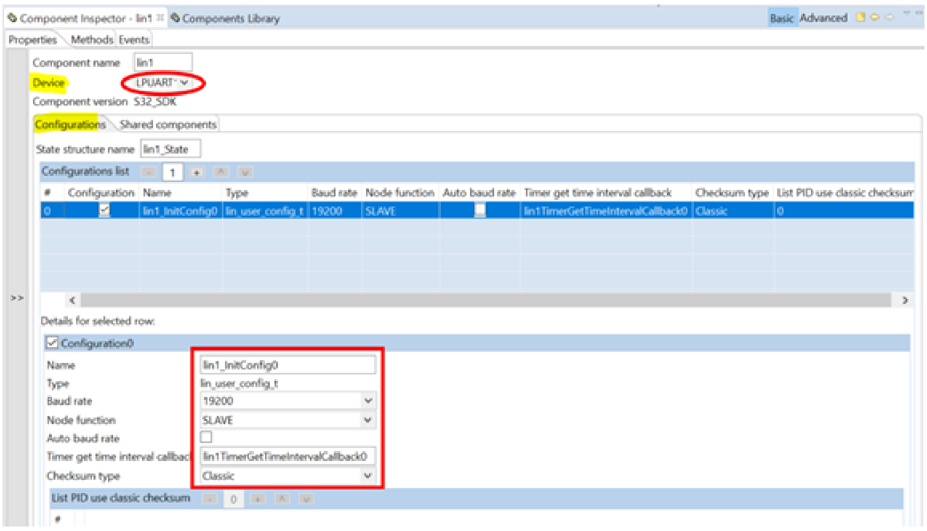
The following snippet shows an example (complete code please refer to void **CAN_Motor_Speed_Control** (void) function implementation in /**Sources/Application.c**).

```
uint8_t CAN_RxMsg_Index = 0; /*buffer index of the received CAN message */

/*handle the Speed control command from CAN bus*/
if(CAN_RxComplete_Flag)
{
    /*get the new received CAN message buffer index*/
    if(0 == Rx_CAN_Msg_Buffer_Index)
    {
        CAN_RxMsg_Index = RX_MAX_BUF_NUM - 1;
    }
    else
    {
        CAN_RxMsg_Index = Rx_CAN_Msg_Buffer_Index - 1;
    }
    /*keep the motor speed is 500 + input(0~4095)/2 = 500~2547 rpm */
    drvFOC.pospeControl.wRotElReq = (500 +
((Rx_CAN_Msg_Buffer[CAN_RxMsg_Index].data[0]<<8) +
    Rx_CAN_Msg_Buffer[CAN_RxMsg_Index].data[1])/2)*1000/fmScale.speed_n_m;

    CAN_RxComplete_Flag = 0;/* clean the flag */
}

/* send status feedback via CAN massage frame */
CAN_Transmit(TX_MSG_ID,Motor_Status_Feedback,8);
```

5.3 LIN communication API

5.3.1 Processor expert configuration for LIN communication

MCSXTE2BK142 uses LPUART1 to implement LIN communication, and it works a LIN slave node with baud rate of 19200 bit/s, so configure the S32K1xx SDK LIN component in Processor Expert as shown below.



Component name: lin1
Device: LPUART1
Component version: S32_SDK

State structure name: lin1_State

#	Configuration	Name	Type	Baud rate	Node function	Auto baud rate	Timer get time interval callback	Checksum type	List PID use classic checksum
0	✓	lin1_InitConfig0	lin_user_config_t	19200	SLAVE	☐	lin1TimerGetTimeIntervalCallback0	Classic	0

Details for selected row:

Configuration0

Name	lin1_InitConfig0
Type	lin_user_config_t
Baud rate	19200
Node function	SLAVE
Auto baud rate	☐
Timer get time interval callbac	lin1TimerGetTimeIntervalCallback0
Checksum type	Classic

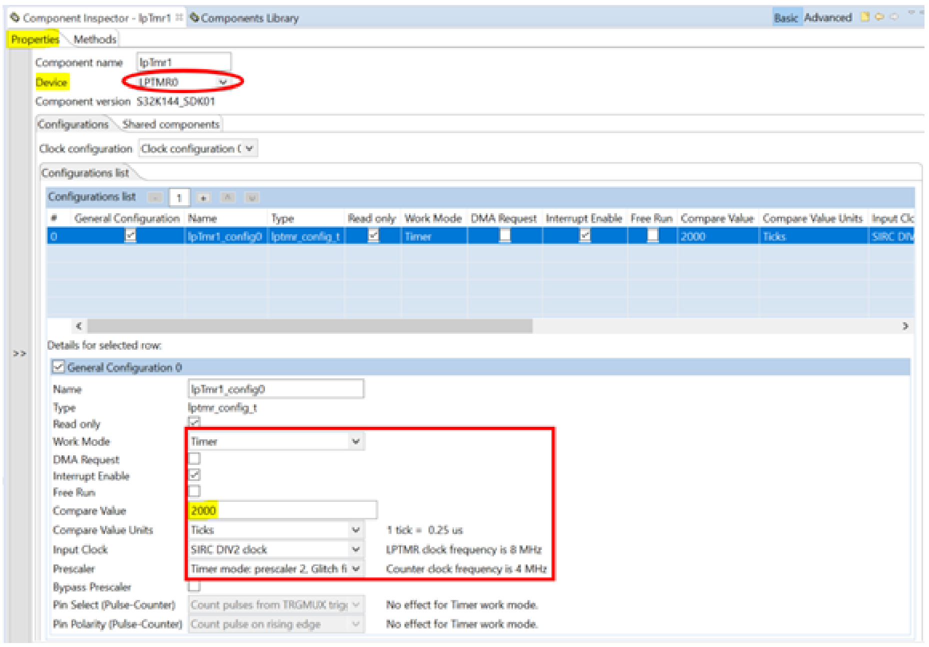
List PID use classic checksum 0

Figure 40. LIN component configuration

To get accurate time interval for LIN auto baud rate and wakeup feature, the LIN LLD requires a get timer time interval callback, it's implemented as below.

```
uint32_t lin1TimerGetTimeIntervalCallback0(uint32_t *ns)
{
   /*get current timer counter*/
    timerCounterValue[1] = (uint16_t)(LPTMR0->CNR );

    /* calculate to get the interval time*/
   *ns = ((uint32_t)(timerCounterValue[1] +
          timerOverflowInterruptCount*TIMER_COMPARE_VAL -
          timerCounterValue[0]))*1000 / TIMER_TICKS_1US;

    /*reset the timer overflow interrupt counter*/
    timerOverflowInterruptCount = 0U;

    /*buffer current time stamp*/
    timerCounterValue[0] = timerCounterValue[1];

    return 0U;
}
```

The timer interval uses the LPTMR (lptmr component) with 500 us overflow interrupt (compare value is 2000).



Component name lpTmr1
Device LPTMR0
Component version S32K144_SDK01
Clock configuration Clock configuration (

Configurations list 1

#	General Configuration	Name	Type	Read only	Work Mode	DMA Request	Interrupt Enable	Free Run	Compare Value	Compare Value Units	Input Clc
0	✓	lpTmr1_config0	lptmr_config_t	✓	Timer		✓		2000	Ticks	SIRC DIV

Details for selected row:

General Configuration 0

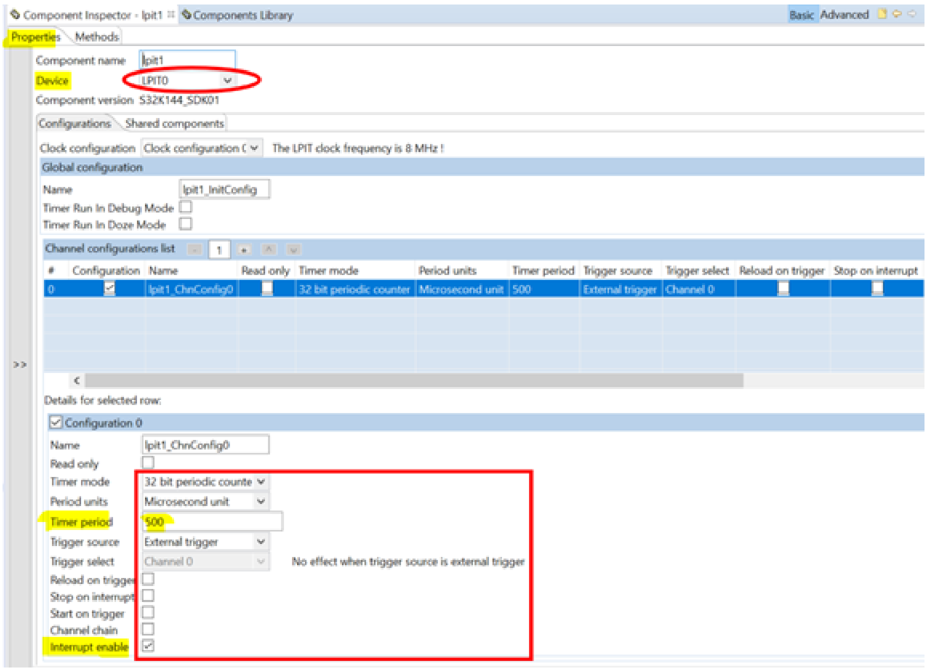
Name	lpTmr1_config0	
Type	lptmr_config_t	
Read only	✓	
Work Mode	Timer	
DMA Request		
Interrupt Enable	✓	
Free Run		
Compare Value	2000	
Compare Value Units	Ticks	1 tick = 0.25 us
Input Clock	SIRC DIV2 clock	LPTMR clock frequency is 8 MHz
Prescaler	Timer mode: prescaler 2, Glitch fi	Counter clock frequency is 4 MHz
Bypass Prescaler		
Pin Select (Pulse-Counter)	Count pulses from TRGMUX trig	No effect for Timer work mode.
Pin Polarity (Pulse-Counter)	Count pulse on rising edge	No effect for Timer work mode.

Figure 41. lptmr component configuration

Besides, it also uses a LPIT channel with 500 us interrupt period to call LIN driver API--LIN_DRV_TimeoutService() for LIN message frame transmit and receive timeout.

```
/**********************************************************************
*  Function:     void LPIT_ISR(void)
*  Description: the PIT timer channel interrupt ISR
*                clear the interrupt flag and call the LIN timeout service
***********************************************************************/
void LPIT_ISR(void)
{
    /* Clear LPIT channel flag */
    LPIT_DRV_ClearInterruptFlagTimerChannels(INST_LPIT1, (1 << LPIT_CHANNEL));

    /* call the LIN timeout service */
    LIN_DRV_TimeoutService(INST_LIN1);
}
```

The S32K1xx SDK **lpit** PD timer component is configured in Processor Expert and is shown below.



Component name: lpit1
Device: LPIT0
Component version: S32K144_SDK01

Clock configuration: Clock configuration (— The LPIT clock frequency is 8 MHz !

Global configuration
Name: lpit1_InitConfig
Timer Run In Debug Mode: ☐
Timer Run In Doze Mode: ☐

Channel configurations list: 1

#	Configuration	Name	Read only	Timer mode	Period units	Timer period	Trigger source	Trigger select	Reload on trigger	Stop on interrupt
0	✓	lpit1_ChnConfig0	☐	32 bit periodic counter	Microsecond unit	500	External trigger	Channel 0	☐	☐

Details for selected row:
✓ Configuration 0
Name: lpit1_ChnConfig0
Read only: ☐
Timer mode: 32 bit periodic counte
Period units: Microsecond unit
Timer period: 500
Trigger source: External trigger
Trigger select: Channel 0 — No effect when trigger source is external trigger
Reload on trigger: ☐
Stop on interrupt: ☐
Start on trigger: ☐
Channel chain: ☐
Interrupt enable: ✓

Figure 42. lpit component configuration

5.3.2 APIs of LIN communication

The main APIs of LIN communication are:

- void **LIN_Comm_Init**(void): The function completes LIN communication initialization. It initializes the TJA1027 LIN transceiver, initialize LPIT and LPTMR for LIN timeout handle, set used peripheral interrupt priority, and install the LIN interrupt event ISR callback to handle the LIN frame.
- void **LIN_DRV_ISR_CallBack**(uint32_t instance, void * linState): The LIN interrupt event ISR callback to handle the LIN frame, handle the data request and response to the LIN event, receive and transmit the LIN data according to the LIN frame PID.

User can add customized LIN event handle code according to the available event defined in **lin_event_id_t**.

```
typedef enum {
    LIN_NO_EVENT            = 0x00U,    /*!< No event yet */
    LIN_WAKEUP_SIGNAL        = 0x01U,    /*!< Received a wakeup signal */
    LIN_BAUDRATE_ADJUSTED     = 0x02U,    /*!< Indicate that baudrate was
adjusted to Master's baudrate */
    LIN_RECV_BREAK_FIELD_OK   = 0x03U,    /*!< Indicate that correct Break
Field was received */
    LIN_SYNC_OK             = 0x04U,    /*!< Sync byte is correct */
    LIN_SYNC_ERROR           = 0x05U,    /*!< Sync byte is incorrect */
    LIN_PID_OK              = 0x06U,    /*!< PID correct */
    LIN_PID_ERROR            = 0x07U,    /*!< PID incorrect */
    LIN_FRAME_ERROR          = 0x08U,    /*!< Framing Error */
    LIN_READBACK_ERROR       = 0x09U,    /*!< Readback data is incorrect */
    LIN_CHECKSUM_ERROR       = 0x0AU,    /*!< Checksum byte is incorrect */
    LIN_TX_COMPLETED         = 0x0BU,    /*!< Sending data completed */
    LIN_RX_COMPLETED         = 0x0CU,    /*!< Receiving data completed */
    LIN_RX_OVERRUN           = 0x0DU     /*!< RX overrun flag */
} lin_event_id_t;
```

Besides, user can also directly call the following APIs provided by S32K1xx SDK lin PD component to send and receive LIN message frame, check node status and switch to sleep mode or idle state according to application requirements.

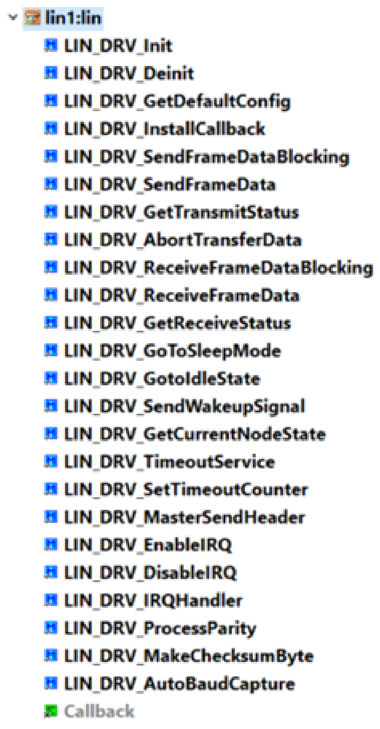

lin1:lin
- LIN_DRV_Init
- LIN_DRV_Deinit
- LIN_DRV_GetDefaultConfig
- LIN_DRV_InstallCallback
- LIN_DRV_SendFrameDataBlocking
- LIN_DRV_SendFrameData
- LIN_DRV_GetTransmitStatus
- LIN_DRV_AbortTransferData
- LIN_DRV_ReceiveFrameDataBlocking
- LIN_DRV_ReceiveFrameData
- LIN_DRV_GetReceiveStatus
- LIN_DRV_GoToSleepMode
- LIN_DRV_GotoIdleState
- LIN_DRV_SendWakeupSignal
- LIN_DRV_GetCurrentNodeState
- LIN_DRV_TimeoutService
- LIN_DRV_SetTimeoutCounter
- LIN_DRV_MasterSendHeader
- LIN_DRV_EnableIRQ
- LIN_DRV_DisableIRQ
- LIN_DRV_IRQHandler
- LIN_DRV_ProcessParity
- LIN_DRV_MakeChecksumByte
- LIN_DRV_AutoBaudCapture
- Callback

Figure 43. LIN driver API list

5.4 PWM communication API

In many old-fashioned automotive motor control systems, PWM command communication is still preferred over LIN or CAN. Even for in-lab application debugging, it is very convenient to use PWM command communication to control the motor placed in a test-bench or in an acoustic or thermal chamber, especially when the BDM debugger or the SCI/RS232 communication is not available.

PWM control is usually specified on the OEM level and it is not provided as a public standard. Nevertheless, adjustable software driver can be created to help the PWM command signal to be detected.

5.4.1 PWM signal specification for motor control

Since there is no public standard specification on PWM signal, let's target a general case with adjustable parameters. Since the PWM control signal is being replaced by LIN communication, the specification may be based on 5.5 V to 18 V voltage range with nominal 12 V level. PWM frequency varies from 10 Hz to 1 kHz (or more), but it really depends on the requirements of the application, compatibility, etc. Duty-cycle range of the PWM signal might be from 15% to 90% from stand-by to full speed. Hysteresis at low duty-cycle range (between 10 and 15%) prevents the system from periodic on/off switching at the lowest duty-cycle. Duty-cycles outside of this range may be identified as not valid and appropriate action may be taken (example full speed command to an engine cooling fan).

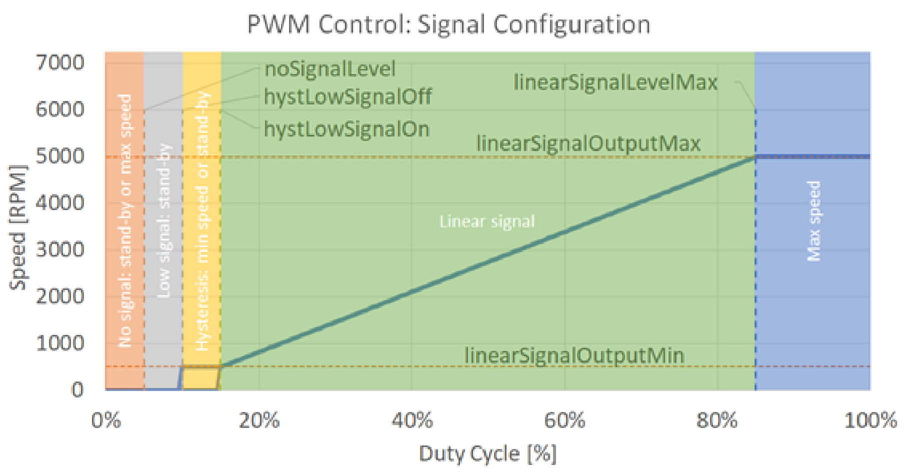


Figure 44. A typical PWM signal duty cycle vs motor speed control algorithm implementation

The signal definition should consider some specific values to help with the signal reading. The values are defined in the following table.

Table 6. Parameters of typical PWM signal detection for motor control

Label	Description
noSignalLevel	No signal detected below this duty cycle
noSignalOutput	Output on "no signal" detected
lowSignalOutput	Output on "low signal" detected
hystLowSignalOff	Low level of the hysteresis. Smaller duty-cycle means "low signal" state, higher duty-cycle enters the hysteresis range (min speed or stand-by)
hystLowSignalOn	High level of the hysteresis. Smaller duty-cycle means "hysteresis area: min speed or stand-by", higher duty-cycle means "run" in the linear signal range, with the speed given by the duty-cycle.
linearSignalOutputMin	Low level of the linear mode
linearSignalLevelMax	High level of the linear mode

Considering normal operation in stand-by mode, the duty-cycle is between **noSignalLevel** and **hystLowSignalOff**. In order to add some distortion-proof feature, the duty-cycle can go up to the **hystLowSignalOn** and the stand-by mode is still detected. In order to engage the "run" state, the duty-cycle has to exceed the **hystLowSignalOn** value and the linear area is reached. Corresponding output is calculated to meet the **linearSignalOutputMin** at the **hystLowSignalOn** value and **linearSignalOutputMax** at the **linearSignalLevelMax**. System keeps the output at **linearSignalOutputMax** for duty-cycles higher than **linearSignalLevelMax**. When slowing down, reaching the **hystLowSignalOn** sets the output to **linearSignalOutputMin** and keeps this output until the **hystLowSignalOff** is reached. This way, the hysteresis feature is provided in order to prevent motors from randomly switch on and off in case the PWM signal is noisy. Lowering the duty-cycle below the **hystLowSignalOff**, zero output is set, which represents the stand- by mode. Duty-cycles below the **noSignalLevel** are treated as not valid and **noSignalOutput** is engaged (which can be set to zero or to max, according to the application requirements).

For more details on PWM signal detection algorithm, refer to the following url.

5.4.2 Processor expert configuration for PWM communication

On MCSXTE2BK142, there is a hardware high voltage PWM input circuit for this function, the PWM_IN is connected to S32K142 FTM1 **Channel 4**(PTA10).

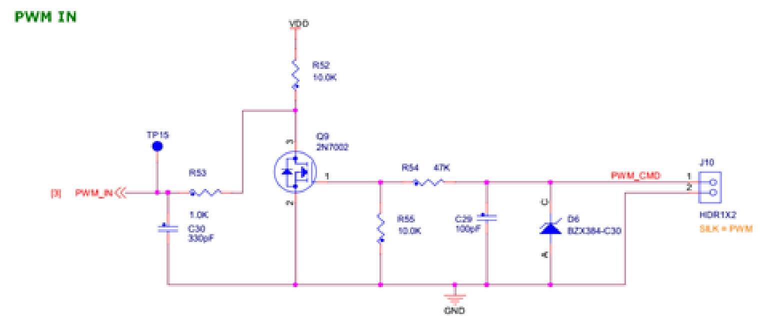
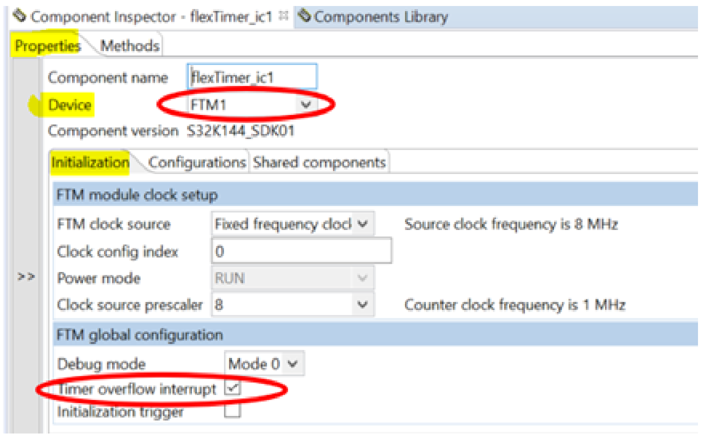


Figure 45. High-voltage PWM communication circuit

The **flexTimer_ic** component is configured as below:

Select **Device** as **FTM1** to configure it with **8** MHz **SIRC** as reference clock and overflow interrupt enabled.



Component Inspector - flexTimer_ic1 | Components Library

Properties | Methods

Component name: flexTimer_ic1
Device: FTM1
Component version: S32K144_SDK01

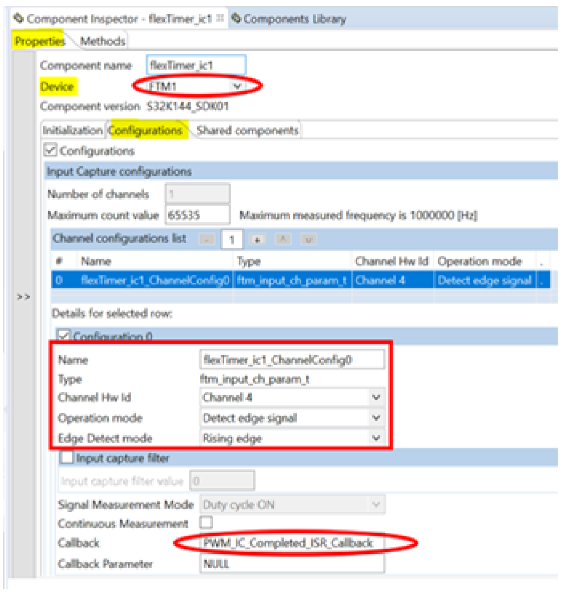
Initialization | Configurations | Shared components

FTM module clock setup
FTM clock source: Fixed frequency clock — Source clock frequency is 8 MHz
Clock config index: 0
Power mode: RUN
Clock source prescaler: 8 — Counter clock frequency is 1 MHz

FTM global configuration
Debug mode: Mode 0
Timer overflow interrupt: ✓
Initialization trigger: ☐

Figure 46. flexTimer_ic component initialization configuration

Also configure to use **channel 4** as the input capture channel with Detect edge signal and the input capture interrupt callback as “**PWM_IC_Completed_ISR_Callback**”.



Component Inspector - flexTimer_ic1 | Components Library

Properties | Methods

Component name: flexTimer_ic1
Device: FTM1
Component version: S32K144_SDK01

Initialization | Configurations | Shared components

[✓] Configurations

Input Capture configurations

Number of channels: 1
Maximum count value: 65535   Maximum measured frequency is 1000000 [Hz]

Channel configurations list: 1

#	Name	Type	Channel Hw Id	Operation mode
0	flexTimer_ic1_ChannelConfig0	ftm_input_ch_param_t	Channel 4	Detect edge signal

Details for selected row:

[✓] Configuration 0

Name	flexTimer_ic1_ChannelConfig0
Type	ftm_input_ch_param_t
Channel Hw Id	Channel 4
Operation mode	Detect edge signal
Edge Detect mode	Rising edge

[ ] Input capture filter

Input capture filter value	0
Signal Measurement Mode	Duty cycle ON
Continuous Measurement	[ ]
Callback	PWM_IC_Completed_ISR_Callback
Callback Parameter	NULL

Figure 47. flexTimer component input capture configuration

In BSP/PWM_Comm/PWM_Comm.c, implement the FlexTimer timer input capture interrupt callback: call the PWM control "Update" routine and switch the input capture mode between the rising edge and falling edge detection to measure PWM signal period and duty.

```
void PWM_IC_Completed_ISR_Callback(ic_event_t event, void *userData)
{
  uint16_t Timer_CaptureVal = 0;

  if(IC_EVENT_MEASUREMENT_COMPLETE==event)
  {
    Timer_CaptureVal = FTM_DRV_GetInputCaptureMeasurement(INST_FLEX-
TIMER_IC1,PWM_IC_FTM_CH);

    /* Read PWM input capture result and update status */
    PWMControlUpdate(Timer_CaptureVal, &pwmControlData);
    /* Read the PWM Control output */
    pwmControlData.outputValue = PWMControlGetOutputValue(&pwmControlData);

    /* Toggle edge detection */
    if(pwmControlData.flags.risingEdge)
     {
  FTM_DRV_SetOutputlevel(INST_FLEXTIMER_IC1,PWM_IC_FTM_CH,FTM_RISING_EDGE);
  FTM_DRV_StartNewSignalMeasurement(INST_FLEXTIMER_IC1,PWM_IC_FTM_CH);
    }
    if(pwmControlData.flags.fallingEdge)
    {
FTM_DRV_SetOutputlevel(INST_FLEXTIMER_IC1,PWM_IC_FTM_CH,FTM_FALLING_EDGE);
FTM_DRV_StartNewSignalMeasurement(INST_FLEXTIMER_IC1,PWM_IC_FTM_CH);
    }
  }
}
```

The FlexTimer **overflow** interrupt ISR is enabled and implemented to clear the measurement faults as shown below.

```
void PWM_IC_FTM_Overflow_ISR(void)
{
    /* Detect PWM input control period overflow (ultra low frequencies) */
     PWMControlTimerOverflow(&pwmControlData);

     /* get the PWM input measure result */
    pwmControlData.outputValue = PWMControlGetOutputValue(&pwmControlData);

     /* Clear interrupt flag */
     FTM_DRV_ClearStatusFlags(INST_FLEXTIMER_IC1, FTM_TIME_OVER_FLOW_FLAG);
}
```

5.4.3 API of PWM communication

To use the PWM communication API, user needs to include its header file.

```
#include PWM_comm.h
```

This BSP module provides the following API for user application.

- **unsigned int PWMControlUpdate**(**unsigned int** pin, **pwmControl_t** * data): Check the PWM input signal, calculate period and duty cycle. Call this function within an input capture interrupt of a timer;
- tFrac32 **PWMControlGetOutputValue**(**pwmControl_t** * data): Output value calculation.Call this function whenever a new updated value is needed
- **void PWMControlTimerOverflow**(**pwmControl_t** * data): Timer Overflow handling. Call this function within a timer overflow.
- **void PWM_Comm_Init**(**void**): Initialize the FTM(FlexTimer) timer channel with input capture mode with interrupt, configure the FTM timer overflow interrupt ISR callback and interrupt priority, initialize the **pwmControl_t** structure (**pwmControlData**) with the input PWM signal feature, start the first edge capture and signal measurement.

In application, use the following global variables to get the measure result and enable/disable the PWM signal measurement function.

```
extern pwmControl_t pwmControlData; /*PWM input control data structure */
extern tBool  pwmControlEnabled; /*the PWM control switch, OFF by default*/
```

An example (complete code please refer to void **PWM_Motor_Speed_Control** (void) function implementation in /**Sources**/ **Application.c**) in the demo project is shown below, take the PWM input signal duty cycle to control motor speed.

```
void PWM_Motor_Speed_Control(void)
{
    float Speed_required = 0;

    /* If PWM input control enabled, update the demanded speed */
    if(pwmControlEnabled)
    {
     /* If the linear or high signal is detected, switch the app on */
     if(pwmControlData.ControlInputClass >= PWM_LinearCtrlDetect)
     {
         cntrState.usrControl.switchAppOnOff = 1;

         Speed_required = (float)pwmControlData.outputValue/2147483648.0f;
         /* Store the demanded speed */
         drvFOC.pospeControl.wRotElReq = (Speed_required *
3000)*1000/fmScale.speed_n_m;
     }
   }
}
```

Chapter 6 Using the demo project

In the SW package, a demo project is provided for user to test the board with a real motor control application. In the following chapter, how to use the demo project will be introduced.

6.1 Motor parameters measurement

If you do not have the specified motor type and want to use your own motor to replace,it does not work properly due to parameters difference. So you need change the configuration before running your own motor. You must already know these parameters. Commonly, you can get these parameters from datasheet or you can measure it by yourself.

To measure motor parameters, besides motor you need to prepare some common equipment like multimeter, oscilloscope, RLC meter, Power Supply and so on.

For more information on measurement of motor parameters, see AN4680.

The following section provides you with method to measure some basic parameters.

6.1.1 Measure Ld and Lq

The method of testing L_d and L_q is similar.

Equipment needed: RLC meter

L_d Steps:

1. Align the rotor to phase A. Phase A is connected to the positive potential (+) and phase B and C are grounded (-).
2. Lock the rotor shaft.
3. Apply negative step voltage. Phase A is grounded (-) and phases B and C are connected to the positive potential (+). Usual level of the current is about 10% of the rated phase current.
4. Measure the step response of the current by a current probe.
5. Calculate inductance L_d.

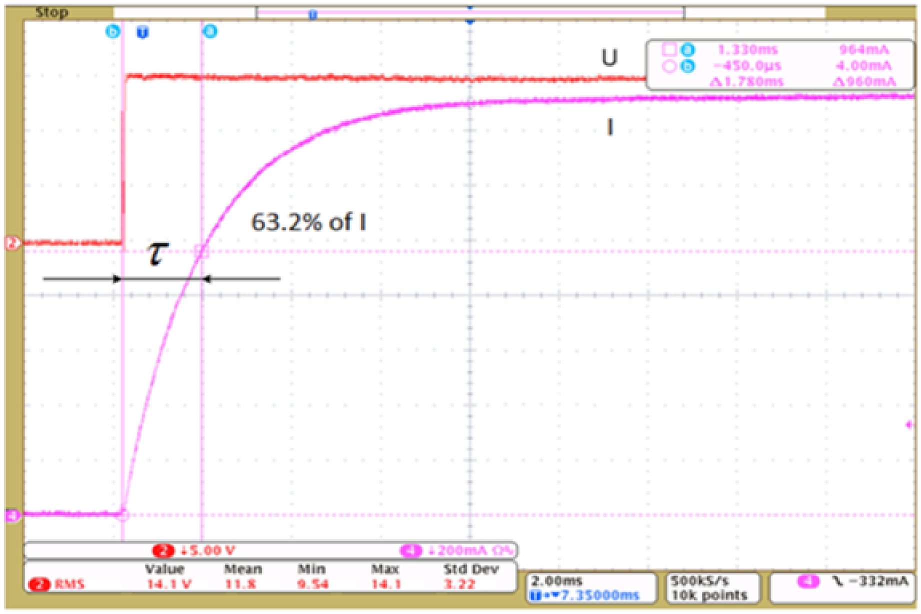


Figure 48. Current step response waveform

L_q Steps:

1. Align the rotor to the q-axis. Connect the phase B terminal to the positive potential (+) of the voltage source and phase C is grounded (-). Phase A terminal is floating.
2. Lock the rotor shaft firmly because current step response in q-axis creates torque.
3. Generate a current step response in this configuration: phase A is connected to the positive potential (+) of the voltage source and phases B and C are grounded.
4. Calculate inductance L_q in the same way as L_d.

6.1.2 Measure pole pairs

Equipment needed: oscilloscope

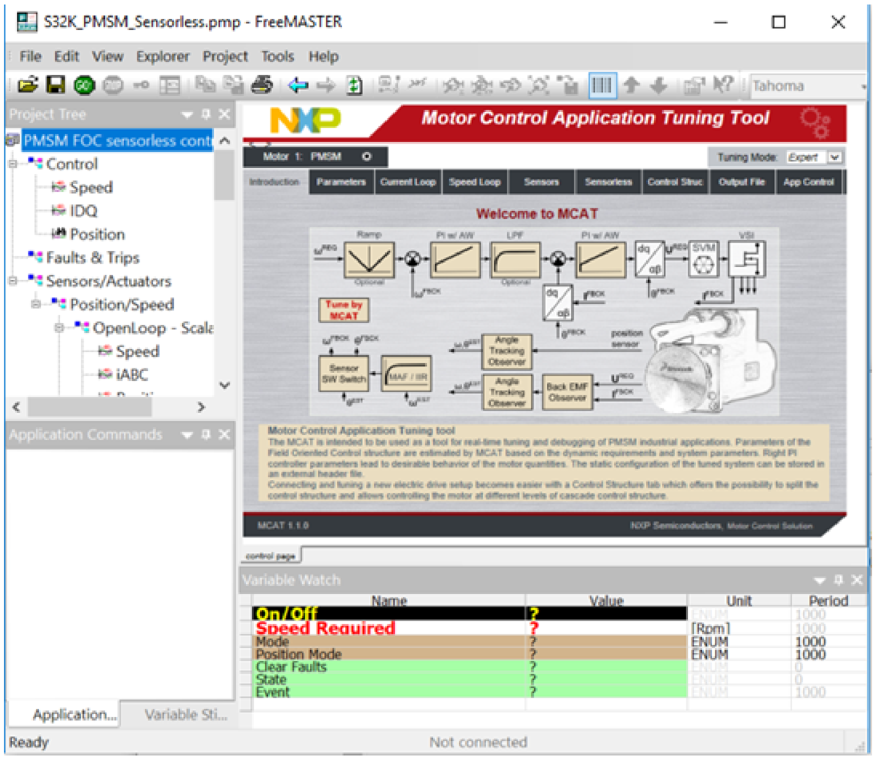
Steps:

1. Disconnect power supply.
2. Connect any two phases port with oscilloscope probes.
3. First make a mark on the motor, then turn the motor by hand one cycle, observe and record the number of waveforms displayed on the oscilloscope, that is, the polar logarithm.

6.2 FreeMASTER configuration

After finishing the measurement of motor's parameter, you need FreeMASTER to regenerate the configured profile with the new parameter.

Go into FreeMASTER_control folder under project's root path. Double click **S32K_PMSM_Sensorless.pmp** file, it will open FreeMASTER as shown below.

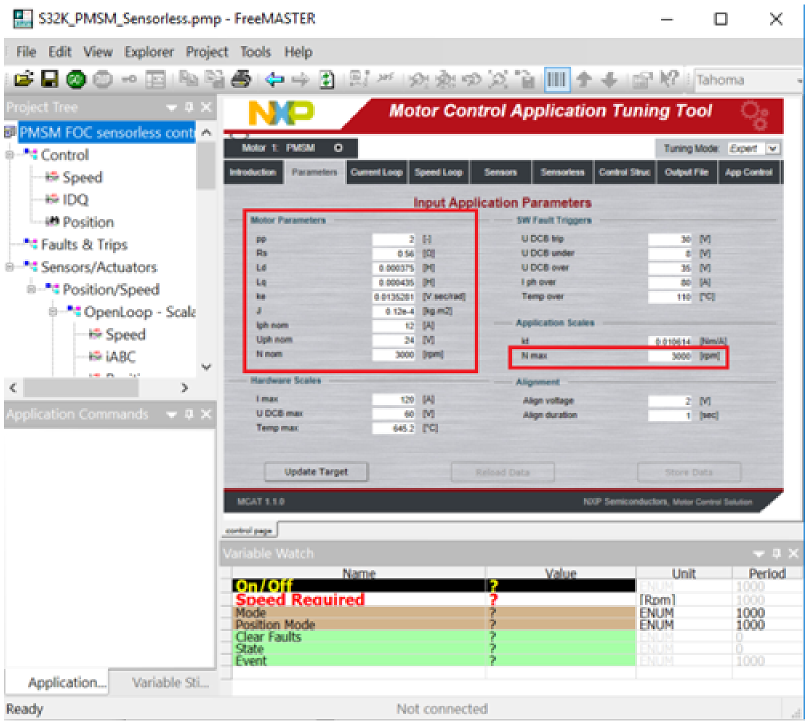


Motor Control Application Tuning tool
The MCAT is intended to be used as a tool for real-time tuning and debugging of PMSM industrial applications. Parameters of the Field Oriented Control structure are estimated by MCAT based on the dynamic requirements and system parameters. Right PI controller parameters lead to desirable behavior of the motor quantities. The static configuration of the tuned system can be stored in an external header file.
Connecting and tuning a new electric drive setup becomes easier with a Control Structure tab which offers the possibility to split the control structure and allows controlling the motor at different levels of cascade control structure.

Name	Value	Unit	Period
On/Off	?	ENUM	1000
Speed Required	?	[Rpm]	1000
Mode	?	ENUM	1000
Position Mode	?	ENUM	1000
Clear Faults	?	ENUM	0
State	?	ENUM	0
Event	?	ENUM	1000

Figure 49. FreeMASTER main window

Switch to “**Parameters**” sheet shown in the following picture and replace the parameters in red rectangular box using your new values.



Input Application Parameters

Motor Parameters

pp	2	[-]
Rs	0.56	[Ω]
Ld	0.000375	[H]
Lq	0.000435	[H]
ke	0.0135281	[V sec/rad]
J	0.12e-4	[kg.m2]
Iph nom	12	[A]
Uph nom	24	[V]
N nom	3000	[rpm]

SW Fault Triggers

U DCB trip	30	[V]
U DCB under	8	[V]
U DCB over	35	[V]
I ph over	80	[A]
Temp over	110	[°C]

Application Scales

kt	0.010614	[Nm/A]
N max	3000	[rpm]

Hardware Scales

I max	120	[A]
U DCB max	60	[V]
Temp max	645.2	[°C]

Alignment

Align voltage	2	[V]
Align duration	1	[sec]

Name	Value	Unit	Period
On/Off	?	ENUM	1000
Speed Required	?	[Rpm]	1000
Mode	?	ENUM	1000
Position Mode	?	ENUM	1000
Clear Faults	?	ENUM	0
State	?	ENUM	0
Event	?	ENUM	1000

Figure 50. Motor parameter configure window in FreeMASTER

Go to **Output File** sheet, click the **Generate Configuration File** button.

6.3 Multiple motor parameter management

After you regenerate your own motor's parameter configure file. It is suggested to back up using the following steps.

1. Copy and rename the following files (Here the "NewMotor" just as an example).

   .\Motor_Control\Motor_Config\ PMSM_appconfig.h →

   .\Motor_Control\Motor_Config\ **NewMotor.h**

   .\FreeMASTER_control\MCAT\param_files\M1_params.txt →

   .\FreeMASTER_control\MCAT\param_files\ **NewMotor.txt**
2. Go back to project's root folder, copy and rename "NewMotorTemplate.bat" file and open it, replace the **NewMotor.h and NewMotor.txt** in command list.

After you finish the above steps, when you want to use this motor, just double click the related ***.bat** file, there is no need to re-configure the parameter in FreeMASTER every time.

6.4 Demo project work flowchart introduction

The board SW package provides a demo project to enable user a out-of-box motor control experience.

Following is the demo project work flowchart.

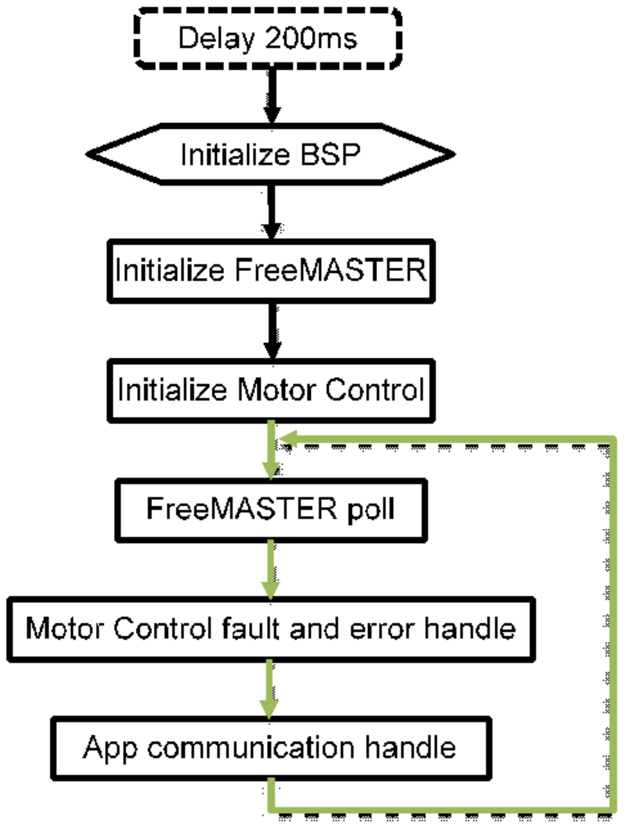


Figure 51. Demo project work flowchart

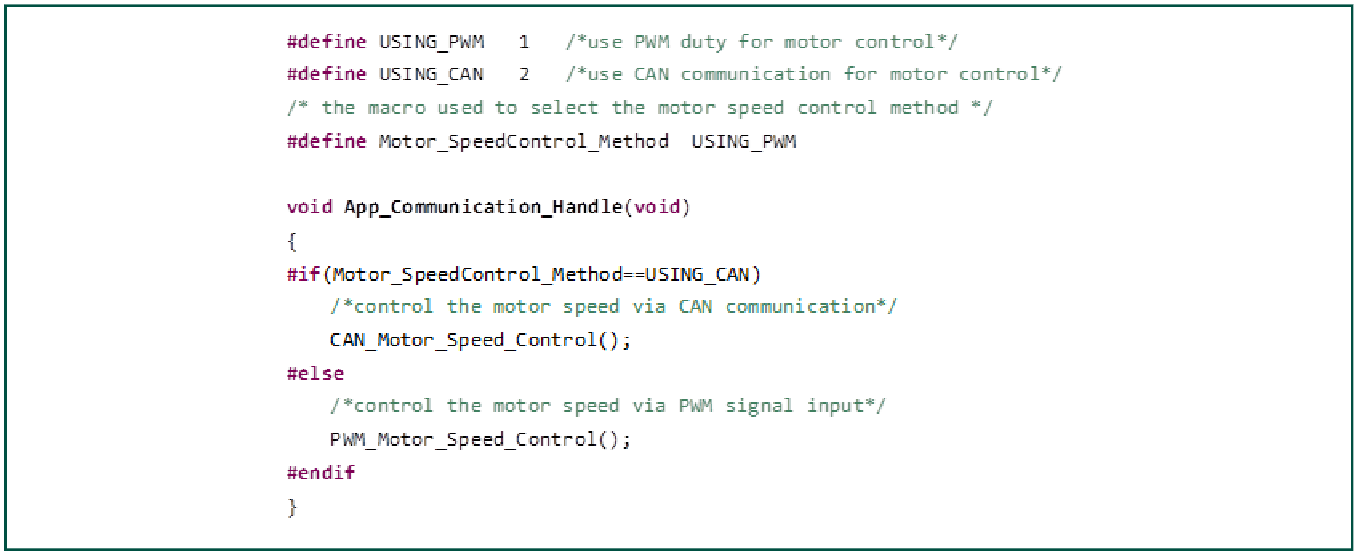
After board POR reset, memory ECC and App data copy initialization complete in startup process, adding 200 ms delay is optical to ensure MC33GD3000 to be initialized well as expected. Then it calls the **BSP_Init()** to initialize all onboard modules as required for motor control application, such as system clock, pin mux, ADC,PDB, and FlexTimer PWM as well as CAN/LIN/PWM communication and GD3000 gate pre-driver initialization. **FMSTR_Init()** is used to initialize the FreeMASTER data record protocol and LLD for communication with MCAT. After these preparations, **App_MotorControl_Init()** is called to initialize MCAT used variables and motor control state machine, configure the FOC control mode with ***speedControl***, start **FTM3** to generate PWM output to start the motor control.

Finally in the main loop, CPU polls and response MCAT (FreeMASTER) commands, handle the motor control faults and error as well as CAN/LIN/PWM communication via periodically calling API functions **FMSTR_Poll()**, **App_Motor_Fault_Error_Handle()** and **App_Communication_Handle()**. All the FOC motor control related algorithm and codes are called and executed in the **ADC1_IRQHandler()**, including phase current and DC bus voltage and current as well as temperature signals convert result read, FOC calculation, motor control state machine maintain, FlexTimer PWM duty cycle update and so on. PDB modules are configured to drive the ADC channel conversion according the FOC period automatically.

User can use one of the flowing methods to control the motor **RUN/STOP** and rotation speed:

1. MCAT FreeMASTER via **UART** communication(J2, 112500bit/s)" or SWD/JTAG debug port(**J3/J4**);
2. PWM signal input via **J10**(default configuration)
3. CAN (via **J9**, **500Kbit/s CAN 2.0 standard frame**) and LIN (via **J8**, **19200bit/s LIN frame with enhance checksum**) communication, CAN controls the speed and LIN switches motor **RUN/STOP**;

User can change between method 2) and method 3) by macro-- **Motor_SpeedControl_Method** in **Sources/Application.h**.



```
#define USING_PWM   1   /*use PWM duty for motor control*/
#define USING_CAN   2   /*use CAN communication for motor control*/
/* the macro used to select the motor speed control method */
#define Motor_SpeedControl_Method  USING_PWM

void App_Communication_Handle(void)
{
#if(Motor_SpeedControl_Method==USING_CAN)
    /*control the motor speed via CAN communication*/
    CAN_Motor_Speed_Control();
#else
    /*control the motor speed via PWM signal input*/
    PWM_Motor_Speed_Control();
#endif
}
```

6.5 Build and Debug the demo project

In the software package, three separate demo projects for different FOC methods are provided.

1. Single-shunt FOC: MCSXTE2BK142_PMSM_FOC_1Sh
2. Dual-shunt FOC: MCSXTE2BK142_PMSM_FOC_2Sh
3. Triple-shunt FOC: MCSXTE2BK142_PMSM_FOC_3Sh

Their usage are similar, but in this document MCSXTE2BK142_PMSM_FOC_2Sh is used as an example to demonstrate how to build and debug these demo projects.

6.5.1 Build and debug the project with S32DS for ARM v2018.R1 IDE

From menu File → Import…, open the **Import** window.

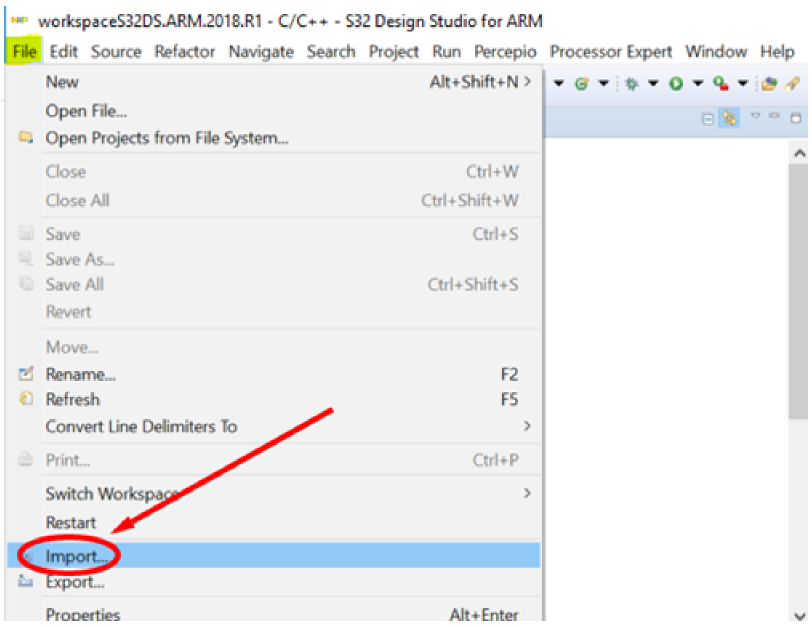


Figure 52. Menu of File to open Import window

Select General → Existing Projects into Workspace → Next.

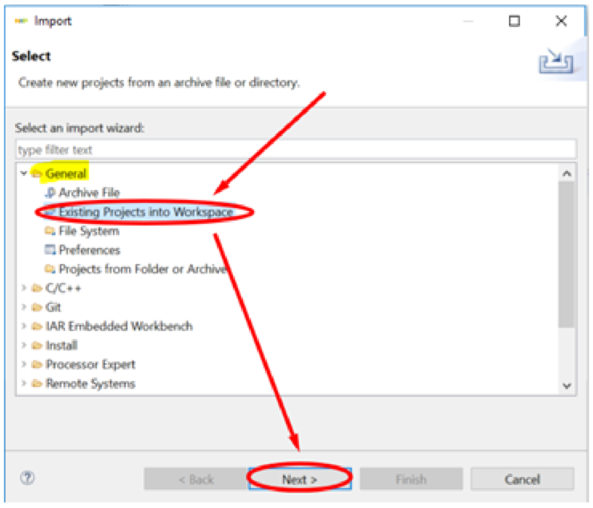
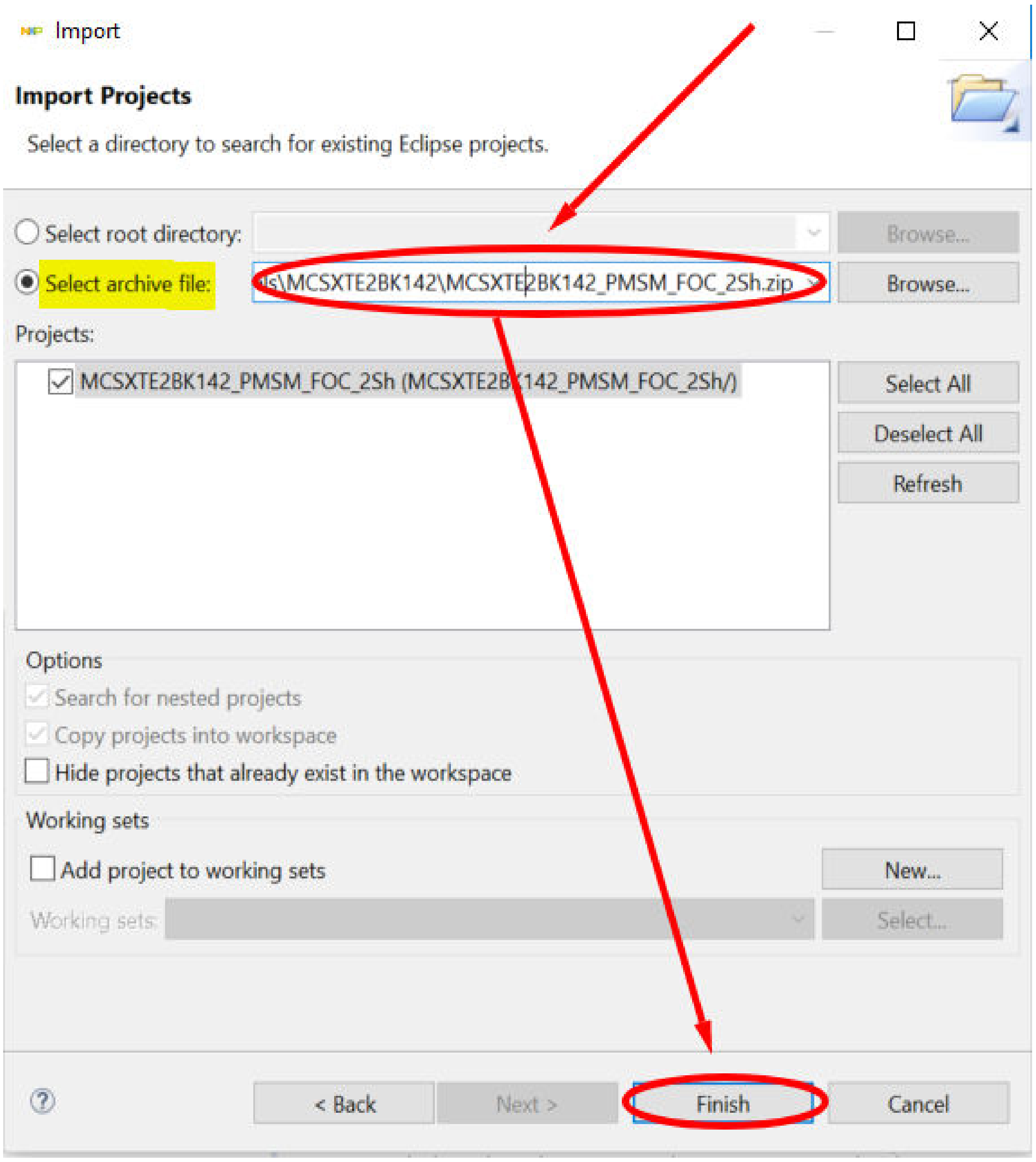


Figure 53. Select Existing Projects into Workspace

Select “Select archive file”, click on browse and choose the PMSM FOC project you want to evaluate, like MCSXTE2BK142_PMSM_FOC_2Sh.zip uncompressed from the **QSP** Finish. The demo project will be imported to current S32DS IDE workspace.



Import

Import Projects

Select a directory to search for existing Eclipse projects.

Select root directory:

Select archive file: s\MCSXTE2BK142\MCSXTE2BK142_PMSM_FOC_2Sh.zip

Projects:

☑ MCSXTE2BK142_PMSM_FOC_2Sh (MCSXTE2BK142_PMSM_FOC_2Sh/)

Options

☑ Search for nested projects

☑ Copy projects into workspace

☐ Hide projects that already exist in the workspace

Working sets

☐ Add project to working sets

Figure 54. Selecting the demo project

In the project explorer, select the project and right click “Clean Project” to clean it at first and then select “Build Project” to build the project.

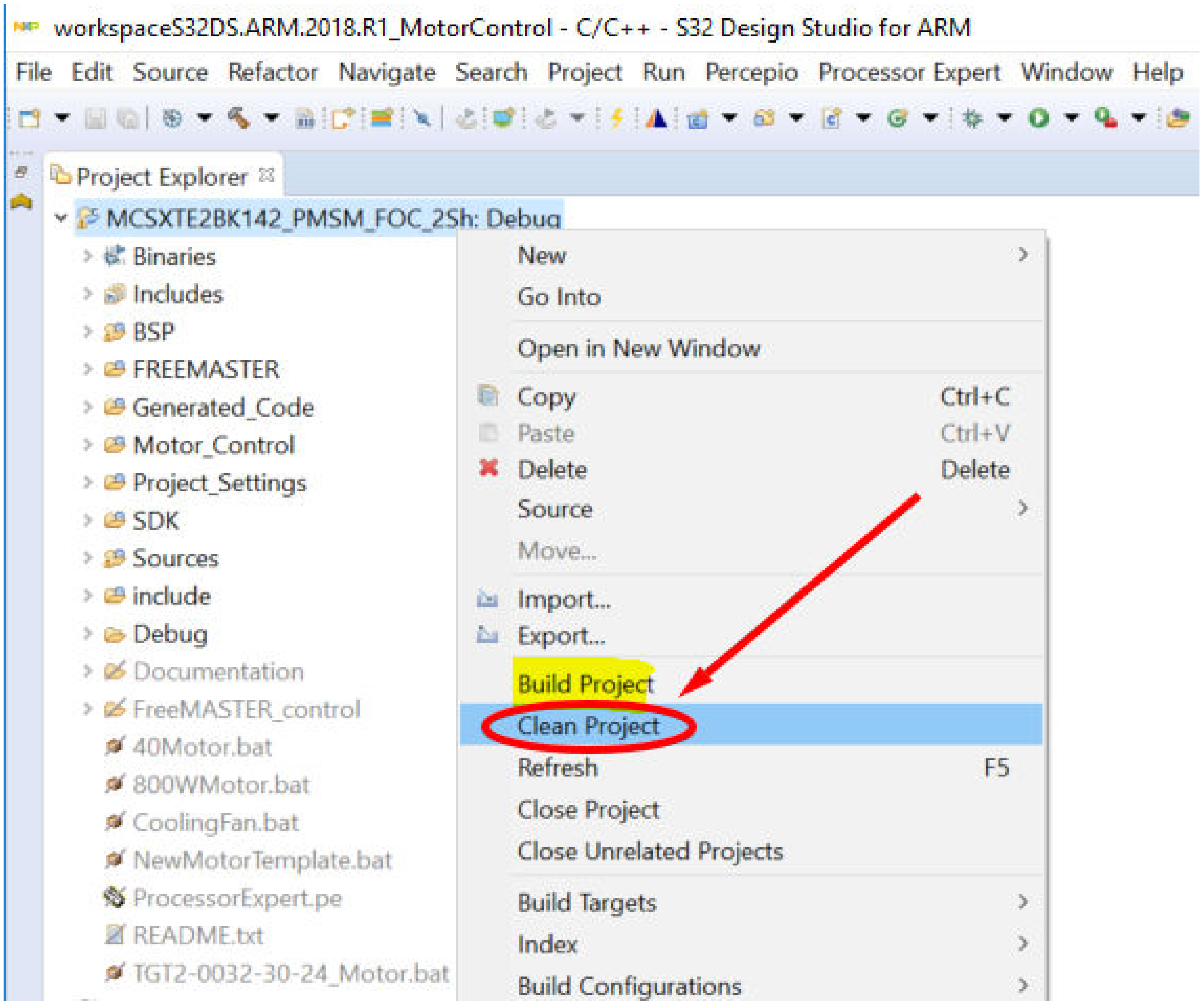
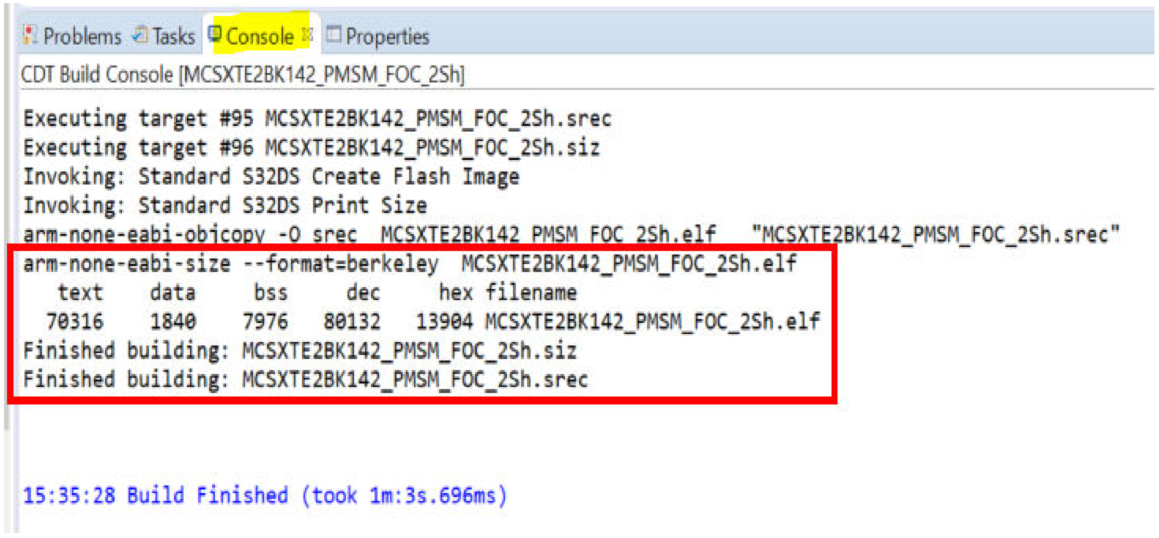


Figure 55. Clean and build project

After successfully project build, the Console will output and compile the result with size print and map. Srecord file is generated under **Debug** folder.

Figure 56. The compile result



CDT Build Console [MCSXTE2BK142_PMSM_FOC_2Sh]

```
Executing target #95 MCSXTE2BK142_PMSM_FOC_2Sh.srec
Executing target #96 MCSXTE2BK142_PMSM_FOC_2Sh.siz
Invoking: Standard S32DS Create Flash Image
Invoking: Standard S32DS Print Size
arm-none-eabi-objcopy -O srec  MCSXTE2BK142_PMSM_FOC_2Sh.elf   "MCSXTE2BK142_PMSM_FOC_2Sh.srec"
arm-none-eabi-size --format=berkeley  MCSXTE2BK142_PMSM_FOC_2Sh.elf
   text    data     bss     dec     hex filename
  70316    1840    7976   80132   13904 MCSXTE2BK142_PMSM_FOC_2Sh.elf
Finished building: MCSXTE2BK142_PMSM_FOC_2Sh.siz
Finished building: MCSXTE2BK142_PMSM_FOC_2Sh.srec



15:35:28 Build Finished (took 1m:3s.696ms)
```

The first time debug is launched, user needs to select the debug configuration. Select the project in project explorer, right click to select “Debug As” → “Debug Configurations…”.

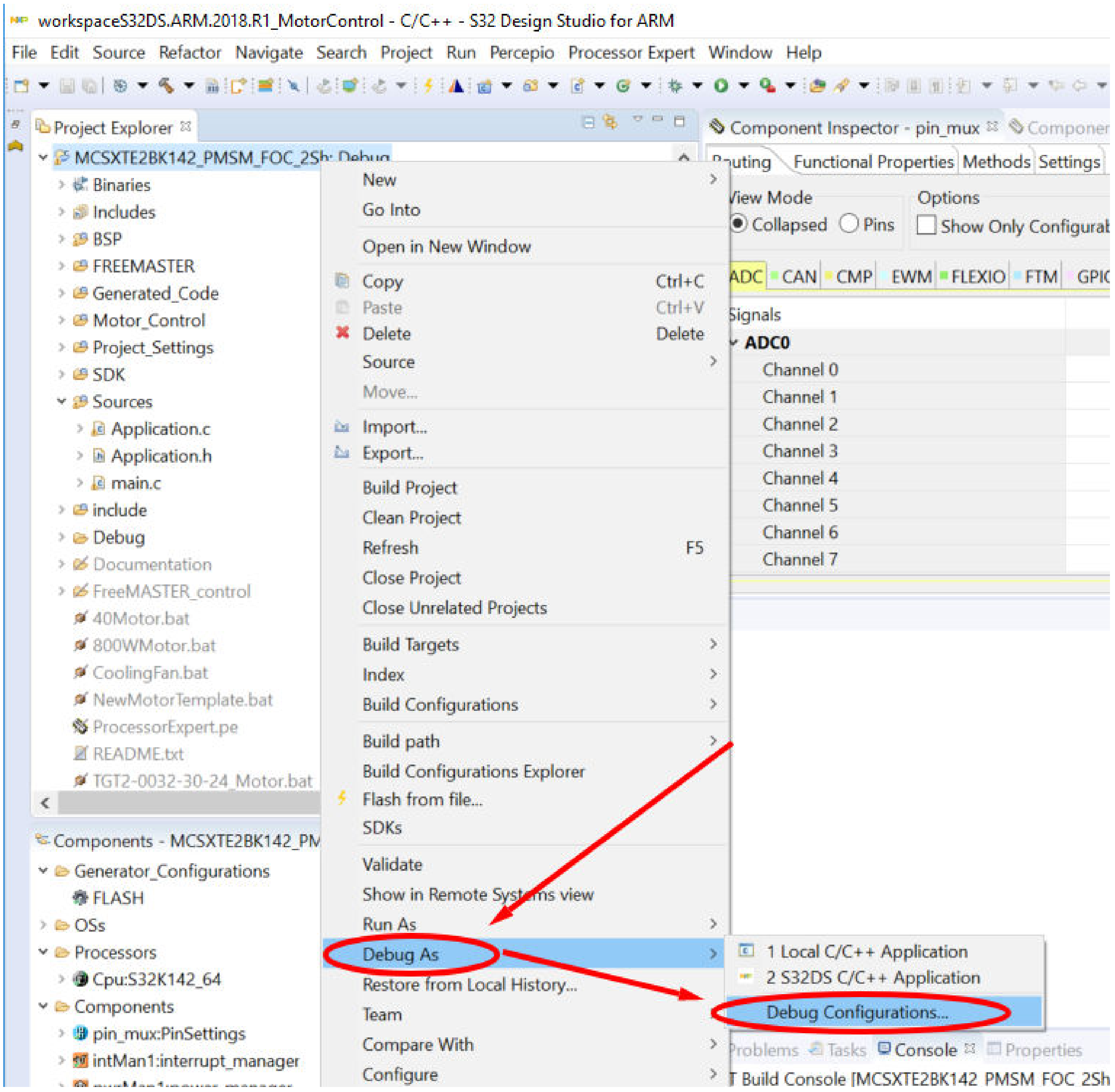


Figure 57. Menu to launch Debug configurations

In the debug configuration window, you can choose any of the following debug targets to launch the debug.

Using PEMicro debugger, such as **U-Multilink/FX** or **Cyclone**:

GDB PEMicro Interface Debugging

1. MCSXTE2BK142_PMSM_FOC_2Sh_Debug
2. MCSXTE2BK142_PMSM_FOC_2Sh_Debug_GCC_Makefile
3. MCSXTE2BK142_PMSM_FOC_2Sh_Debug_GHS_Makefile
4. MCSXTE2BK142_PMSM_FOC_2Sh_Debug_IAR_Makefile
5. MCSXTE2BK142_PMSM_FOC_2Sh_Debug_RAM

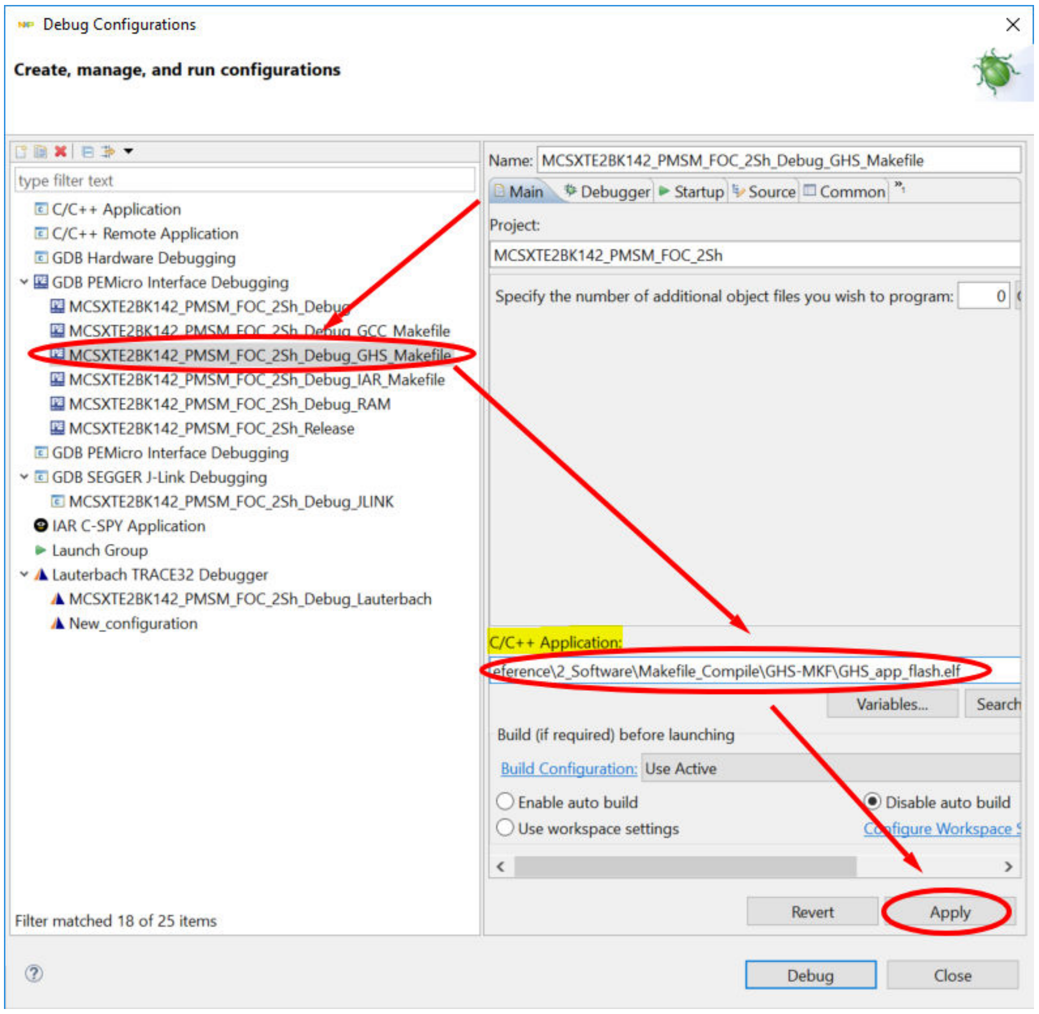
6. MCSXTE2BK142_PMSM_FOC_2Sh_Debug_Release

NOTE

For about debug target b), c) and d) should be launched after complete corresponding toolchain makefile (refer to Build and debug the project with Makefile) and generate the elf successfully.

NOTE

Besides, it’s necessary to check and ensure the debug target set with the actual elf file location as below screenshot.



Debug Configurations

Create, manage, and run configurations

- C/C++ Application
- C/C++ Remote Application
- GDB Hardware Debugging
- GDB PEMicro Interface Debugging
  - MCSXTE2BK142_PMSM_FOC_2Sh_Debug
  - MCSXTE2BK142_PMSM_FOC_2Sh_Debug_GCC_Makefile
  - MCSXTE2BK142_PMSM_FOC_2Sh_Debug_GHS_Makefile
  - MCSXTE2BK142_PMSM_FOC_2Sh_Debug_IAR_Makefile
  - MCSXTE2BK142_PMSM_FOC_2Sh_Debug_RAM
  - MCSXTE2BK142_PMSM_FOC_2Sh_Release
- GDB PEMicro Interface Debugging
- GDB SEGGER J-Link Debugging
  - MCSXTE2BK142_PMSM_FOC_2Sh_Debug_JLINK
- IAR C-SPY Application
- Launch Group
- Lauterbach TRACE32 Debugger
  - MCSXTE2BK142_PMSM_FOC_2Sh_Debug_Lauterbach
  - New_configuration

Filter matched 18 of 25 items

Name: MCSXTE2BK142_PMSM_FOC_2Sh_Debug_GHS_Makefile

Project: MCSXTE2BK142_PMSM_FOC_2Sh

Specify the number of additional object files you wish to program: 0

C/C++ Application: eference\2_Software\Makefile_Compile\GHS-MKF\GHS_app_flash.elf

Build (if required) before launching

Build Configuration: Use Active

Enable auto build [✓ Disable auto build]

Use workspace settings

Figure 58. Select the debug target based on used debugger

Using Segger debugger, such as J-LINK or J-LINK Pro/Trace:

GDB SEGGER J-Link Debugging

MCSXTE2BK142_PMSM_FOC_2Sh_Debug_JLINK

Using Lauterbach debugger, such as uTrace for ARM Cortex M:

Lauterbach TRACE32 Debugger

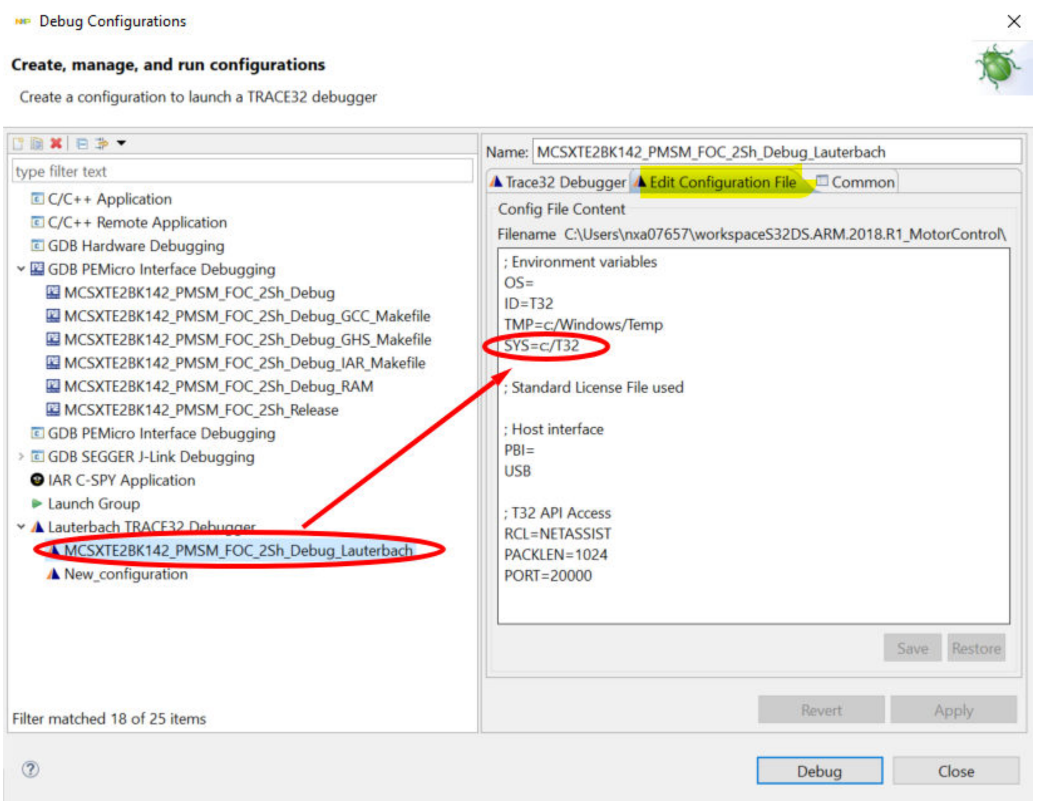
MCSXTE2BK142_PMSM_FOC_2Sh_Debug_Lauterbach.

NOTE

The workspace path must not include any space, otherwise the Lauterbach TRACE32 cannot find the *.cmm command script file.

NOTE

User must install the Lauterbach eclipse plugin before using this debug target and should ensure the T32 software install path is correct configured.

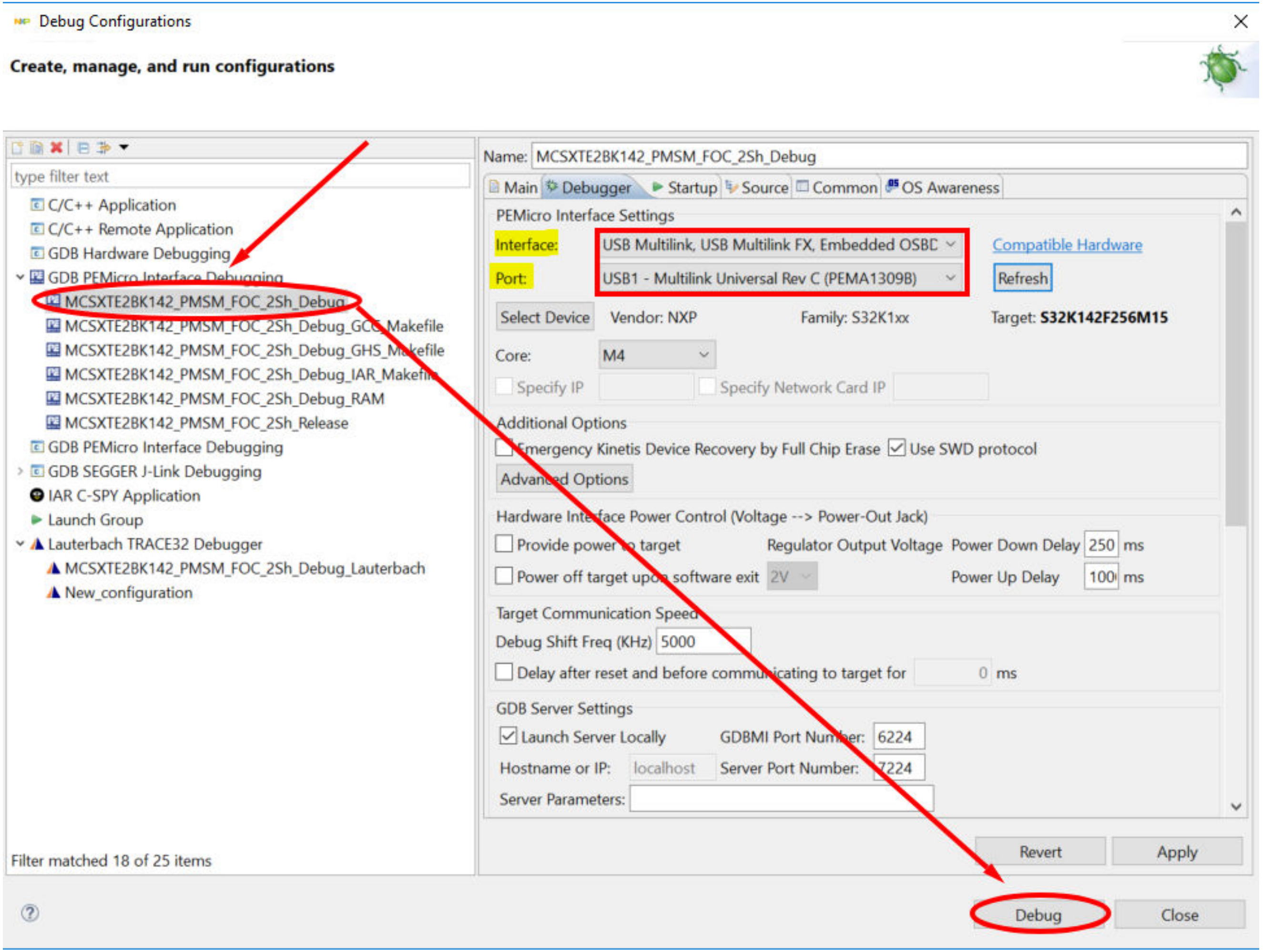

Debug Configurations
Create, manage, and run configurations
Create a configuration to launch a TRACE32 debugger
type filter text
C/C++ Application
C/C++ Remote Application
GDB Hardware Debugging
GDB PEMicro Interface Debugging
MCSXTE2BK142_PMSM_FOC_2Sh_Debug
MCSXTE2BK142_PMSM_FOC_2Sh_Debug_GCC_Makefile
MCSXTE2BK142_PMSM_FOC_2Sh_Debug_GHS_Makefile
MCSXTE2BK142_PMSM_FOC_2Sh_Debug_IAR_Makefile
MCSXTE2BK142_PMSM_FOC_2Sh_Debug_RAM
MCSXTE2BK142_PMSM_FOC_2Sh_Release
GDB PEMicro Interface Debugging
GDB SEGGER J-Link Debugging
IAR C-SPY Application
Launch Group
Lauterbach TRACE32 Debugger
MCSXTE2BK142_PMSM_FOC_2Sh_Debug_Lauterbach
New_configuration
Filter matched 18 of 25 items
Name: MCSXTE2BK142_PMSM_FOC_2Sh_Debug_Lauterbach
Trace32 Debugger
Edit Configuration File
Common
Config File Content
Save
Restore
Revert
Apply
Debug
Close

Filename C:\Users\nxa07657\workspaceS32DS.ARM.2018.R1_MotorControl\

```
; Environment variables
OS=
ID=T32
TMP=c:/Windows/Temp
SYS=c:/T32

; Standard License File used

; Host interface
PBI=
USB

; T32 API Access
RCL=NETASSIST
PACKLEN=1024
PORT=20000
```

Figure 59. Configure the T32 software path in Debug configurations

For example take the PEMicro U-Multilink FX debugger with debug target “MCSXTE2BK142_PMSM_FOC_2Sh_Debug”, select the debugger target and then click Debug.



Debug Configurations

Create, manage, and run configurations

type filter text

- C/C++ Application
- C/C++ Remote Application
- GDB Hardware Debugging
- GDB PEMicro Interface Debugging
  - MCSXTE2BK142_PMSM_FOC_2Sh_Debug
  - MCSXTE2BK142_PMSM_FOC_2Sh_Debug_GCC_Makefile
  - MCSXTE2BK142_PMSM_FOC_2Sh_Debug_GHS_Makefile
  - MCSXTE2BK142_PMSM_FOC_2Sh_Debug_IAR_Makefile
  - MCSXTE2BK142_PMSM_FOC_2Sh_Debug_RAM
  - MCSXTE2BK142_PMSM_FOC_2Sh_Release
- GDB PEMicro Interface Debugging
- GDB SEGGER J-Link Debugging
- IAR C-SPY Application
- Launch Group
- Lauterbach TRACE32 Debugger
  - MCSXTE2BK142_PMSM_FOC_2Sh_Debug_Lauterbach
  - New_configuration

Filter matched 18 of 25 items

Name: MCSXTE2BK142_PMSM_FOC_2Sh_Debug

Main | Debugger | Startup | Source | Common | OS Awareness

PEMicro Interface Settings

Interface: USB Multilink, USB Multilink FX, Embedded OSBD

Port: USB1 - Multilink Universal Rev C (PEMA1309B)

Compatible Hardware | Refresh

Select Device | Vendor: NXP | Family: S32K1xx | Target: **S32K142F256M15**

Core: M4

Specify IP | Specify Network Card IP

Additional Options

☐ Emergency Kinetis Device Recovery by Full Chip Erase ☑ Use SWD protocol

Advanced Options

Hardware Interface Power Control (Voltage --> Power-Out Jack)

☐ Provide power to target | Regulator Output Voltage | Power Down Delay 250 ms

☐ Power off target upon software exit | 2V | Power Up Delay 100 ms

Target Communication Speed

Debug Shift Freq (KHz) 5000

☐ Delay after reset and before communicating to target for 0 ms

GDB Server Settings

☑ Launch Server Locally | GDBMI Port Number: 6224

Hostname or IP: localhost | Server Port Number: 7224

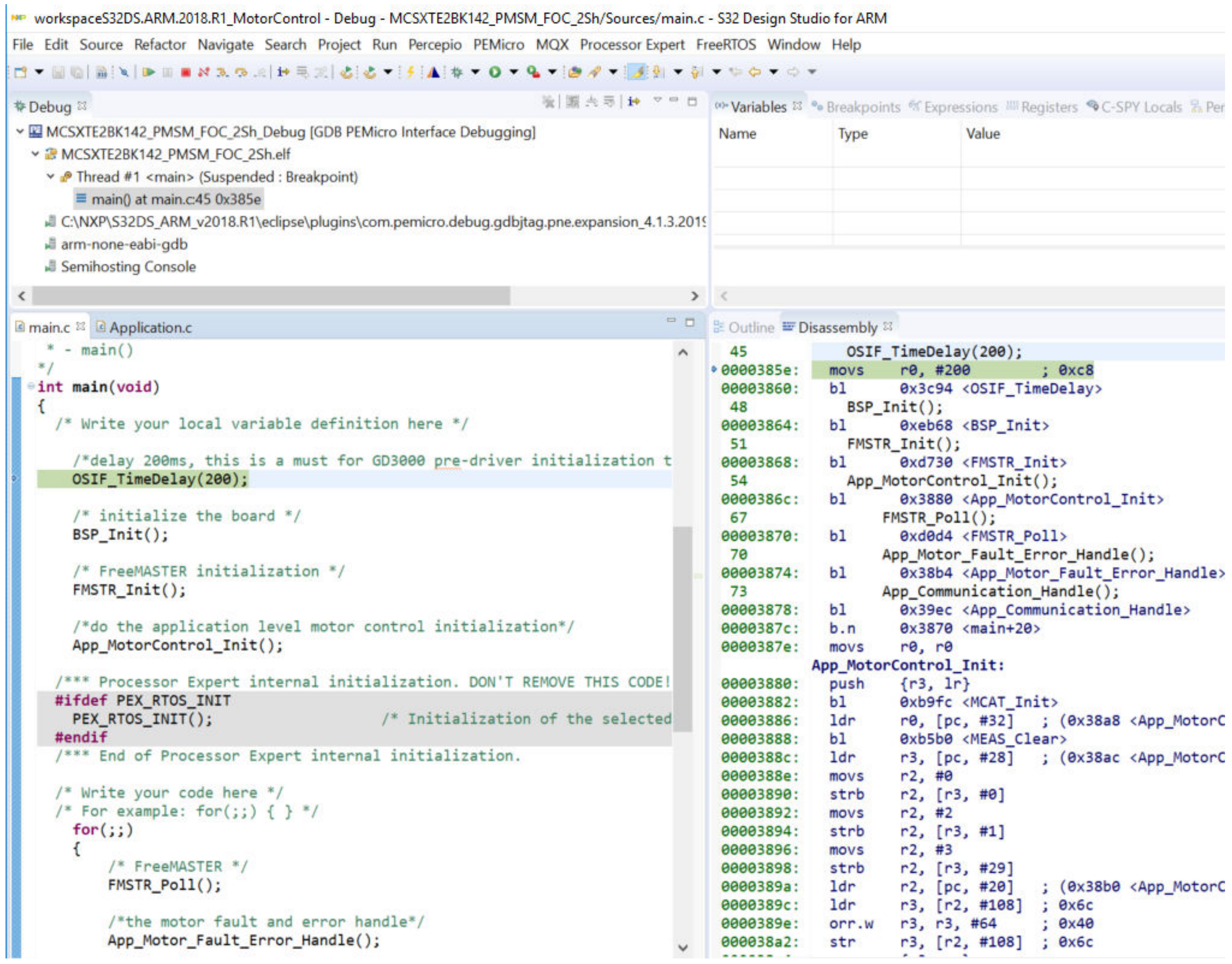
Server Parameters:

Revert | Apply

Debug | Close

Figure 60. Launch the GDB PEMicro debugger

It downloads the compile result(*.elf) and debug information, enter the debug and stop at the beginning of **main**() function.


workspaceS32DS.ARM.2018.R1_MotorControl - Debug - MCSXTE2BK142_PMSM_FOC_2Sh/Sources/main.c - S32 Design Studio for ARM
File Edit Source Refactor Navigate Search Project Run Percepio PEMicro MQX Processor Expert FreeRTOS Window Help
Debug
Variables
Breakpoints
Expressions
Registers
C-SPY Locals
Name
Type
Value
main.c
Application.c
Outline
Disassembly

```
MCSXTE2BK142_PMSM_FOC_2Sh_Debug [GDB PEMicro Interface Debugging]
  MCSXTE2BK142_PMSM_FOC_2Sh.elf
    Thread #1 <main> (Suspended : Breakpoint)
      main() at main.c:45 0x385e
  C:\NXP\S32DS_ARM_v2018.R1\eclipse\plugins\com.pemicro.debug.gdbjtag.pne.expansion_4.1.3.201
  arm-none-eabi-gdb
  Semihosting Console
```

```
 * - main()
 */
int main(void)
{
  /* Write your local variable definition here */

    /*delay 200ms, this is a must for GD3000 pre-driver initialization t
    OSIF_TimeDelay(200);

    /* initialize the board */
    BSP_Init();

    /* FreeMASTER initialization */
    FMSTR_Init();

    /*do the application level motor control initialization*/
    App_MotorControl_Init();

  /*** Processor Expert internal initialization. DON'T REMOVE THIS CODE!
  #ifdef PEX_RTOS_INIT
    PEX_RTOS_INIT();                        /* Initialization of the selected
  #endif
  /*** End of Processor Expert internal initialization.

  /* Write your code here */
  /* For example: for(;;) { } */
    for(;;)
    {
        /* FreeMASTER */
        FMSTR_Poll();

        /*the motor fault and error handle*/
        App_Motor_Fault_Error_Handle();
```

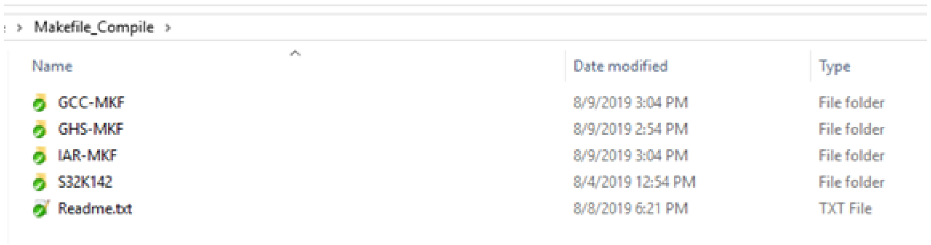
```
45           OSIF_TimeDelay(200);
0000385e:   movs    r0, #200          ; 0xc8
00003860:   bl      0x3c94 <OSIF_TimeDelay>
48           BSP_Init();
00003864:   bl      0xeb68 <BSP_Init>
51           FMSTR_Init();
00003868:   bl      0xd730 <FMSTR_Init>
54           App_MotorControl_Init();
0000386c:   bl      0x3880 <App_MotorControl_Init>
67               FMSTR_Poll();
00003870:   bl      0xd0d4 <FMSTR_Poll>
70               App_Motor_Fault_Error_Handle();
00003874:   bl      0x38b4 <App_Motor_Fault_Error_Handle>
73               App_Communication_Handle();
00003878:   bl      0x39ec <App_Communication_Handle>
0000387c:   b.n     0x3870 <main+20>
0000387e:   movs    r0, r0
          App_MotorControl_Init:
00003880:   push    {r3, lr}
00003882:   bl      0xb9fc <MCAT_Init>
00003886:   ldr     r0, [pc, #32]    ; (0x38a8 <App_MotorC
00003888:   bl      0xb5b0 <MEAS_Clear>
0000388c:   ldr     r3, [pc, #28]    ; (0x38ac <App_MotorC
0000388e:   movs    r2, #0
00003890:   strb    r2, [r3, #0]
00003892:   movs    r2, #2
00003894:   strb    r2, [r3, #1]
00003896:   movs    r2, #3
00003898:   strb    r2, [r3, #29]
0000389a:   ldr     r2, [pc, #20]    ; (0x38b0 <App_MotorC
0000389c:   ldr     r3, [r2, #108]   ; 0x6c
0000389e:   orr.w   r3, r3, #64      ; 0x40
000038a2:   str     r3, [r2, #108]   ; 0x6c
```

Figure 61. The default breakpoint in the main function entry

6.5.2 Build and debug the project with Makefile

The demo project also supports compliation and debug with makefile, the makefile is able to:

- Compile the demo project with toolchain: 1) **GCC**, 2) **GHS** and 3) **IAR**;
- Generate elf and S19 Flash program file;
- Download the compile result into the target MCU Flash with: 1) **J-LINK**, and 2) **Lauterbach uTrace for ARM Cortex M**;
- Launched Lauterbach debug GUI: **Trace32** for project debug.



› Makefile_Compile ›

Name	Date modified	Type
GCC-MKF	8/9/2019 3:04 PM	File folder
GHS-MKF	8/9/2019 2:54 PM	File folder
IAR-MKF	8/9/2019 3:04 PM	File folder
S32K142	8/4/2019 12:54 PM	File folder
Readme.txt	8/8/2019 6:21 PM	TXT File

Figure 62. Folder trees of makefile

The following steps are recommended to use the makefile:

Step1: Place the Makefile_Compile folder uncompressed from the QSP into the same folder of the demo project;

- GCC-MKF: GCC toolchain, GCC 6.3 from S32DS for ARM v2018.R1 is used by default;
- IAR-MKF: IAR toolchain, IAR Embedded_Workbench_8.0_ARM_8.11.2 is used by default;
- GHS-MKF: GHS toolchain, GHS comp_201854 is used by default;
- S32K142: contains the linker file and startup file for different toolchain;
- Readme.txt: a readme file;

Step2: Enter the toolchain folder you want to use, configure the toolchain to be used:

In each toolchain folder, there are the following files:

- Makefile: The project makefile of the source file input and compile/link options configuration;
- Makefile-arm: make target definition based on project makefile configuration;
- config.t32: Lauterbach debugger config file;
- project.cmm: The Lauterbach debug project setting file;
- JLINK_S32K142_Flash_Program_Command_Script.txt: The J-LINK Flash program command script file;
- Launch_Compile.bat: batch file to launch the IAR toolchain for compile and link;
- Launch_Debug.bat: batch file to launch the Lauterbach debugger for debug;
- Launch_Flash_Program.bat: batch file to launch the Flash program with J-LINK as the Flash programmer;
- Readme.txt: A usage introduction file;

To configure GCC as the compile toolchain:

1. The NXP AMMCLIB is correctly installed and the path variable (AMMCLIB_PATH, AMMCLIBDIR_INC) is configured in file "Makefile"
2. a make utility (e.g. the S32DS for ARM v2018.R1(default)) is available and its path is exactly added to system environment(path) in file "Launch_Compile.bat"
3. GCC toolchain is correctly installed (e.g. S32DS for ARM v2018.R1(default)) and its path is exactly added to system environment(path) in file "Launch_Compile.bat"
4. for GCC toolchain, the libgcc.a library path variable--"GCCLIB_PATH" must be configured as actual GCC toolchain installed in file "Makefile"
5. Select the target project to be compiled and debugged with the makefile.

   In line 147(for GCC-MKF)/144(for IAR-MKF)/145(for GHS-MKF) of Makefile, modify to select the target project:

   - single-shunt: BASEDIR:=../../MCSXTE2BK142_PMSM_FOC_1Sh/
   - dual-shunt: BASEDIR:=../../MCSXTE2BK142_PMSM_FOC_2Sh/
   - triple-shunt: BASEDIR:=../../MCSXTE2BK142_PMSM_FOC_3Sh/

To configure GHS as the compile toolchain:

1. the NXP AMMCLIB is correctly installed and the path variable (AMMCLIB_PATH, AMMCLIBDIR_INC) is configured in file "Makefile"
2. a make utility (e.g. the S32DS for ARM v2018.R1(default)) is available and its path is exactly added to system environment(path) in file "Launch_Compile.bat"
3. GHS toolchain is correctly installed (e.g. GHS comp_201854 (default)) and its path is exactly added to system environment(path) in file "Launch_Compile.bat"

To configure IAR as the compile toolchain:

1. the NXP AMMCLIB is correctly installed and the path variable (**AMMCLIB_PATH**, **AMMCLIBDIR_INC**) is configured in file "**Makefile**";
2. a make utility (e.g. the S32DS for ARM v2018.R1(default)) is available and its path is exactly added to system environment(**path**) in file "**Launch_Compile.bat**";
3. **IAR** toolchain is correctly installed (e.g. **IAR Embedded_Workbench_8.0_ARM_8.11.2** (default)) and its path is exactly added to system environment(**path**) in file "**Launch_Compile.bat**";

Step3: Double click the Launch_Compile.bat Windows command batch file to start compile and generate elf/S19 file:

Double click the **Launch_Compile.bat** to launch the project compile and link:

This bat file requires the following input files:

- Makefile
- Makefile-arm

If there are no any error, it generates/output:

- **obj_flash(folder)**: contains the object files(*.o) generated during compile
- **<GCC/GHS/IAR>_app_flash.elf** : the compile and link result--elf(executable) file
- **<GCC/GHS/IAR>_app_flash.map**: the compile and link result—map (memory map) symbol file
- **<GCC/GHS/IAR>_app_flash.s19**: the compile and link generate--S19(Srecord) Flash program file

Step4: **Double click the Launch_Flash_Program.bat Windows command batch file to start download the compile result to target MCU Flash;**

After **Step3** is complete the S19 Flash program file is successfully generated. User can launch the J-LINK to program and compile the result S19 file into the target MCU.

Before running the bat file, user must ensure:

1. Latest version (v6.46b or later) of Segger J-LINK software (it can be downloaded from www.segger.com) is correctly installed on your computer (of course the right license is available) and its install path is configured in file "**Launch_Flash_Program.bat**";
2. J-LINK debugger/programmer support S32K1xx hardware is correctly connected to the target board with 12V/24 power on;

Double click the **Launch_Flash_Program.bat**, the Trace 32 debugger GUI is launched.

This bat file requires the following input files:

- **JLINK_S32K142_Flash_Program_Command_Script.txt**: The J-LINK Flash program command script file
- **<GCC/GHS/IAR>_app_flash.s19**: the compile and link generate--S19(Srecord) Flash program file

Generate/output: None

NOTE

To run the board with motor control function, you need to re-power the board after S19 programmed successfully.

Step5: Double click the Launch_Debug.bat Windows command batch file to start Lauterbach debug GUI:

If user wants to debug the program before downloading it, user can launch the Lauterbach Trace 32 GUI to start the project debug after completing **Step3** and generate the S19 Flash program file successfully.

Before running the bat file, user must ensure:

1. The latest version (**v2019.2 or later**) of Trace 32 software (it can be downloaded from www.lauterbach.com) is correctly installed on your computer with proper license and its install path is configured in file "config.t32" and **Launch_Debug.bat**
2. The Lauterbach debugger support S32K1xx hardware is correctly connected to the target board with 12V/24 power on

Just double click the “**Launch_Debug.bat**”, you can launch the Trace 32 debugger GUI and the SW starts to debug.

This bat file requires the following input files:

- **config.t32**: Lauterbach debugger config file
- **project.cmm**: the Lauterbach debug project setting file
- **<GCC/GHS/IAR>_app_flash.elf**: the compile and link result--elf(executable) file

Generate/output: None

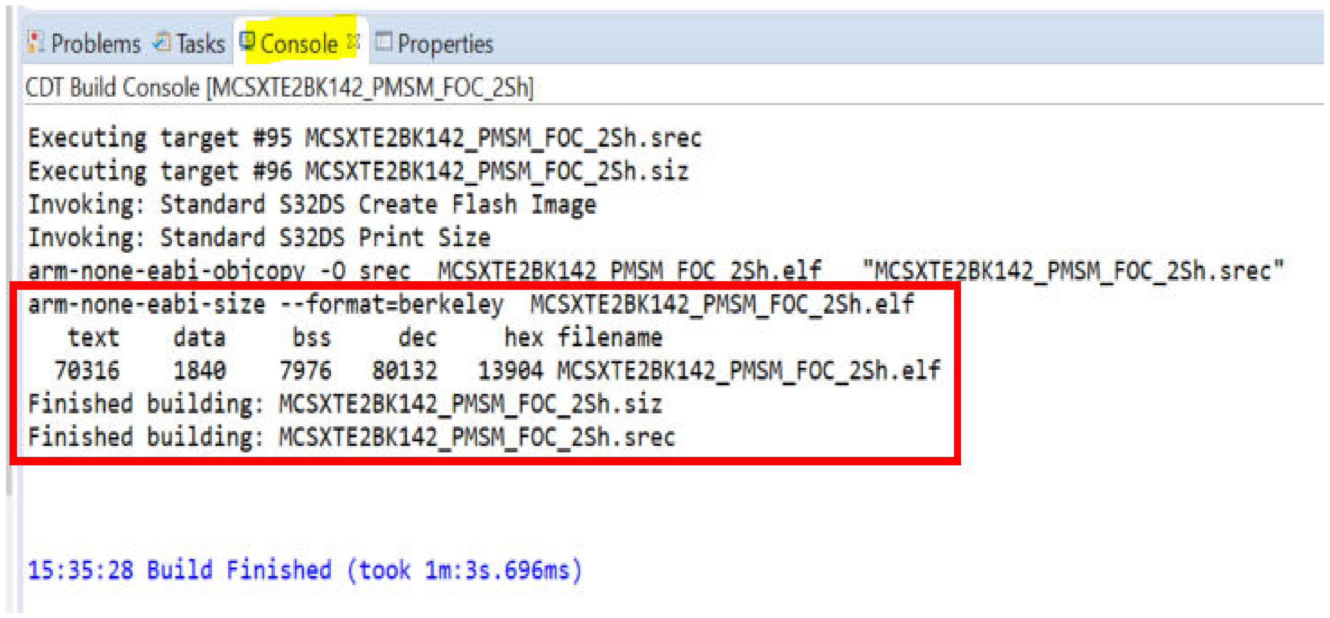
Chapter 7 Performance

This chapter gives out the performance of MCSXTE2BK142 board SW from the following three aspects:

1. Used MCU resource
2. CPU loading for motor control FOC algorithm
3. Temperature vs. power ouput

7.1 Used MCU resource

The SW compile result (S32DS for ARM v2018.R1 with compile optimization level= **-O1**) is as below.



```
Executing target #95 MCSXTE2BK142_PMSM_FOC_2Sh.srec
Executing target #96 MCSXTE2BK142_PMSM_FOC_2Sh.siz
Invoking: Standard S32DS Create Flash Image
Invoking: Standard S32DS Print Size
arm-none-eabi-objcopy -O srec  MCSXTE2BK142_PMSM_FOC_2Sh.elf  "MCSXTE2BK142_PMSM_FOC_2Sh.srec"
arm-none-eabi-size --format=berkeley  MCSXTE2BK142_PMSM_FOC_2Sh.elf
   text    data     bss     dec     hex filename
  70316    1840    7976   80132   13904 MCSXTE2BK142_PMSM_FOC_2Sh.elf
Finished building: MCSXTE2BK142_PMSM_FOC_2Sh.siz
Finished building: MCSXTE2BK142_PMSM_FOC_2Sh.srec



15:35:28 Build Finished (took 1m:3s.696ms)
```

Figure 63. Compiling result

Based on the size print output, the software uses the S32K142 memory as show in the following table.

Table 7. Memory size of the software

Memory Type	Total Size	Used	Free for user APP code
P-Flash	256KB	70.46KB(27.5%)	185.54KB(72.5%)
FlexNVM(D-Flash)	64KB	Not use (0%)	64KB (100%)
FlexRAM (EEE/SRAM)	4KB	Not use (0%)	4KB (100%)
SRAM	28KB	9.59KB(34.2%)	18.41KB(65.8%)

The peripherals used by the board for motor control are summarized as shown in the following table.

Table 8. Peripherals for motor control

Peripherals		Total	Used	Free to Use
Timer	FlexTimer	4x (32)	4x (16)	4x (16 reuse)
	PIT	1x (4 channel)	1x (1 channel)	1x (3 channel)
	LPTMR	1x	1x	0
	PDB	2x	2x	2x (reuse, can add more triggers)
	RTC	1x	Not use	1x
	Systick Timer	1x	1x (for osif)	0
ADC		2x (32)	2x (10)	2x (22, reuse)
FlexCAN (CAN)		2x	1x	1x
LPUART(LIN)		2x	2x (LIN and FreeMASTER)	0
LPSPI		2x	1x	1x
LPI2C		1x	Not use	1x
FlexIO		1x	Not use	1x
GPIO		54	50	4
DMA		1x (16 channel)	Not use	1x (16 channel)

7.2 Motor control key algorithm run time statistics

The runtime of main code for motor control FOC algorithm is calculated using the following configurations:

1. CPU run @80 MHz core clock and bus clock@40 MHz, Flash clock @ 26.667 MHz (RUN mode) ， CPU cache enabled, 32-bit single-point FPU enabled
2. CPU run @112 MHz core clock and bus clock@56 MHz, Flash clock @ 28 MHz (HSRUN mode), CPU cache enabled, 32-bit single-point FPU enabled

For different GCC 6.3 (default toolchain for S32K1xx SDK RTM3.0.0 in S32DS for ARM v2018.R1 IDE) compile optimizations (-O0 : no optimization, -O1 : using optimization level 1) and phase current sample method(dual-shunt/tri-shunt), the test results are listed in the following table:

Table 9. Run time of key FOC algorithm code [1]

Algorithm	Time(us)							
	Dual-shunt				Tri-shunt			
	RUN@80MHz		HSRUN@112MHz		RUN@80MHz		HSRUN@112MHz	
	-O0	-O1	-O0	-O1	-O0	-O1	-O0	-O1
ADC1_Handler2[2]	85.8	48.5	66.8	38.6	96.8	54.8	74.2	43.5
Machine state Run	53.6	33.2	42.5	26.8	58.3	37.5	46.1	30.6
Fast loop	12.8	13.2	13.2	10.7	22	16.2	17.6	13.1
Slow loop	12	9.6	9.5	8.4	12	9.9	9.7	7.4
Fault detection	17.4	6.8	10.4	5.7	13.2	7.3	10.3	5.8

1. The data is measured via toggling MCU GPIO pin and acquired by oscilloscope, all the results are the average value of serval samples.
2. FreeMASTER is not included in the recorder API, it will be removed during production in real time environment.

7.3 Temperature rise reference

As the output power increases, the board temperature at the highest point on the board is shown in the figure below.

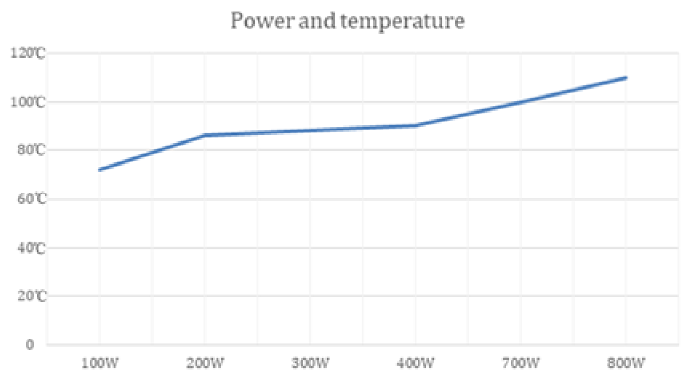


Figure 64. Temperature vs Power output

Chapter 8 Summary

The MCSXTE2BK142 board SW package provides an out-of-box runtime software based on NXP S32K1xx SDK RTM 3.0.0 and AMMCLIB v1.1.5. It offers rich BSP layer APIs of all the board onboard enablements for automotive motor control application, such as CAN/LIN/PWM communication, hall/encoder for sensor based FOC PMSM motor control.

With the overview and implementation details introduction of the SW package in this SW user manual, using the demo project/ code, user can easily boot-up the board and start his/her own motor control application with very few modifications.

Appendix A Reference

1. S32K1xx MCU Family - Data Sheet(REV 11)
2. S32K1xx MCU Family - Reference Manual(REV 11)
3. MC33GD3000, Three Phase Field Effect Transistor Pre-driver - Data Sheet(REV 7.0)
4. Automotive Math and Motor Control Library Set for NXP S32K14x devices, Rev.11
5. PWM Input Control for S12Z
6. AN12235, 3-phase Sensorless PMSM Motor Control Kit (REV 0)
7. AN5327.Three-phase Sensorless Single-Shunt Current-Sensing PMSM Motor Control Application with MagniV MC9S12ZVM (REV 0)

Date of release: April, 2020
Document identifier: MCSXTE2BK142SWUG