UM1053
User manual

Advanced developers guide for STM32F103xx/STM32F100xx PMSM single/dual FOC library

Introduction

This manual describes the Motor Control Software Development Kit (generically called software library) designed for and to be used with STM32F103xx or STM32F100xx microcontrollers (STM32F103xx also called STM32 performance line, STM32F100xx also called STM32 value line). The software library implements the Field Oriented Control (FOC) drive of 3-phase Permanent Magnet Synchronous Motors (PMSM), both Surface Mounted (SM-PMSM) and Internal (I-PMSM).

The control of an AC induction motor equipped with encoder or tacho generator is described in the UM0483 user manual.

The STM32F family of 32-bit Flash microcontrollers is based on the breakthrough ARM Cortex™-M3 core, specifically developed for embedded applications. These microcontrollers combine high performance with first-class peripherals that make it suitable for performing both permanent-magnet and AC induction motor FOC.

The PMSM FOC library can be used to quickly evaluate ST microcontrollers and complete ST application platforms, as well as to save time when developing Motor Control algorithms to be run on ST microcontrollers. This PMSM FOC library is written in C language, and implements the core Motor Control algorithms (reference frame transformations, currents regulation, speed regulation, space-vector modulation, energy efficiency optimizations) as well as sensors reading/decoding algorithms (three shunts, ST-patented single DC link shunt, isolated current sensors, incremental encoder, hall sensors) and a sensorless algorithm for rotor position reconstruction.

When deployed with STM32F103xx High-Density / XL-Density devices (Flash memory density between 256 and 512 Kbytes / 768 Kbytes and 1 Mbyte) the PMSM FOC library allows simultaneous dual FOC of two different motors. The library can be customized to suit user application parameters (motor, sensors, power stage, control stage, pin-out assignment) and provides a ready-to-use Application Programming Interface (API).

A user project has been implemented to demonstrate how to interact with the Motor Control API. The project provides an LCD User Interface and a UART User Interface, represents a convenient real-time fine-tuning and remote control tool for the motor control application.

A PC Graphical User Interface (GUI), the ST MC Workbench, allows complete and easy customization of the PMSM FOC library. In conjunction with the STM3210B-MCKIT motor control starter kit, a PMSM motor can be made to run in a very short time using default parameters.

Basic knowledge of C programming, C++ programming (for customizing the LCD User Interface), PM motor drives and power inverter hardware is necessary for this programming. In-depth know-how of STM32F103xx or STM32F100x peripherals/functions is only required for customizing existing modules and for adding new ones for a complete application development.

Contents

1 Documentation architecture 5
1.1 Where to find the information you need 5
1.2 Related documents 6
Available from www.arm.com 6
Available from www.st.com or your STMicroelectronics sales office 6

2 Object-oriented programming 7
Object 7
Class. 7
Method 7
Inheritance 7
Interface 8

3 Advantages of object-oriented programming 9
3.1 Efficient multiple motor control 9
3.2 Increased safety through data hiding 9
3.3 Modularity 9
3.4 Abstraction 9

4 STM32 PMSM FOC FW library C implementation of OOP 10
4.1 Generic classes source files organization and content 10
ExampleClass.h 10
ExamplePrivate.h 11
ExampleClass.c 12
4.2 Inheritance implementation 14
4.3 Derived classes source file organization and content 15
Derived_ExampleClass.h 15
Derived_ExamplePrivate.h 16
Derived_ExampleClass.c 17
4.4 Motor control library related interrupt handling 19

5 How to create a user defined class 22

6 STM32 PMSM FOC FW library v 3.0 class list 23
6.1 Current reading and PWM generation (CPWMC) and its derived classes 23

6.2 Speed and position feedback (CSPD) and its derived classes 24
6.3 Field-oriented control drive (CFOC) and its derived classes 25
6.4 Bus voltage sensor (CVBS) and its derived classes 26
6.5 Temperature sensor (CTSNS) and its derived classes 26
6.6 Digital Output (CDOUT) class 26
6.7 Encoder Alignment Controller (CEAC) class 27
6.8 Rev-up controller (CRUC) class 27
6.9 Speed and torque controller (CSTC) class 27
6.10 State machine (STM) class 27
6.11 PI (CPI) and PID (CPID) controller classes 29

7 Class interaction 30

7.1 Field orientation, speed and torque control procedures 30
7.2 Procedure for motor ramp-up for sensorless algorithms 31
7.3 Rotor alignment for encoder calibration 32

8 Description of tasks 34

8.1 Low frequency task 34
8.2 Medium frequency task 35
8.3 High frequency task 35
8.4 Safety task 37

9 Bibliography 38

10 Revision history 39

About this document

This document provides important information about the STM32 FOC PMSM FW library V3.0 with specific focus on its object oriented programming implementation and its task organized structure.

It provides the following:

- Overview of object oriented programming, highlighting the advantages of this kind of approach.
- Description of objects, classes and relationships that have been implemented in C language in the FW library.
- Brief depiction for each of the implemented classes and the interaction between them for certain procedures.
- Description of motor control tasks.

1 Documentation architecture

1.1 Where to find the information you need

Technical information about the MC SDK is distinguished and organized by topic. The following is a list of the documents that are available and the subjects they cover:

- This manual (UM1052), STM32F103xx/STM32F100xx permanent-magnet synchronous motor single/dual FOC software library V3.0. This provides the following:
  - Features
  - Architecture
  - Workspace
  - Customization processes
  - Overview of algorithms implemented (FOC, current sensors, speed sensors)
  - MC API
  - Demonstrative user project
  - Demonstrative LCD user interface
  - Demonstrative serial communication protocol
- *Advanced developers guide for STM32F103xx/STM32F100xx PMSM single/dual FOC library* (UM1053). This provides the following:
  - Object-oriented programming style used for developing the MC library
  - Description of classes that belong to the MC library
  - Interactions between classes
  - Description of tasks of the MCA
- MC library source documentation (Doxygen-compiled HTML file). This provides a full description of the public interface of each class of the MC library (methods, parameters required for object creation).
- MC Application source documentation (Doxygen-compiled HTML file). This provides a full description of the classes that make up the MC API.
- User Interface source documentation (Doxygen-compiled HTML file). This provides a full description of the classes that make up the UI Library.
- STM32F10x Standard Peripherals Library source documentation (doxygen compiled html file).
- ST MC Workbench GUI documentation. This is a field guide that describes the steps and parameters required to customize the library, as shown in the GUI.
- In-depth documentation about particular algorithms (sensorless position/speed detection, flux weakening, MTPA, feed-forward current regulation).

Please contact your nearest ST sales office or support team to obtain the documentation you are interested in if it was not already included in the software package you received or available on the ST web site (*www.st.com*).

1.2 Related documents

Available from www.arm.com

- Cortex™-M3 Technical Reference Manual, available from: *http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337e/DDI0337E_cortex_m3_r1p1_trm.pdf*

Available from www.st.com or your STMicroelectronics sales office

- *STM32F103xx datasheet*
- *STM32F100xx datasheet*
- *STM32F103xx user manual (RM0008)*
- *STM32F100xx user manual (RM0041)*
- *STM32F103xx AC induction motor IFOC software library V2.0 (UM0483)*
- *STM32 and STM8 Flash Loader demonstrator (UM0462)*

2 Object-oriented programming

Object Oriented Programming is a programming paradigm whose roots can be traced to the 1960s. When the software started to become more complex, researchers studied ways to organize it in units in order to achieve a high level of modularity and code reusability. As a result, a new way of programming was conceived, which is able to decompose programs into self-sufficient modules (classes), each instance (object) of which contains all the information needed to manipulate its own internal data (representing the object state).

For more information on OOP, refer to the abundant literature on the subject, refer to *Section 9: Bibliography*. A description of OOP fundamental concepts and features is provided here.

Object

The object is a bundle of data structure (members) and functions (methods) allowed to operate on the data structure itself. The data structure contains both object properties and variables and can also be referenced as the state of the object.

Class

A class can be considered the factory from which individual objects are created. It is the user defined data type that contains variables, properties and methods.

Method

A method is an operation that can access the internal state of an object by reading and/or writing its variables and properties. It is important to point out that only an object method can modify its variables; the object internal variables are hidden to object users, who can interact with them only through the object methods. This fundamental principle of OOP is known as data encapsulation or data hiding.

Inheritance

Inheritance is the process through which a class inherits the member and the methods of another class. This type of relationship is called child-parent or derived-base class. Derived (child) classes are a more specialized version of the base (parent) class as they inherit attributes and behavior from the base (parent) class but can also introduce their own.

For example, a class speed sensor might have subclasses called encoder, hall and state observer. Supposing that speed sensor classes define a method called GetElSpeedDpp that exports the related internal variable, all of its derived classes inherit this method and the related internal variable so that the programmer only needs to write it once (and so link to it once).

In addition to speed sensor class methods, encoder, hall and state observer can have their own method (IsObserverConverged, for example) and their own implementation of base class methods through the so called virtual functions. This way, user can always call a base class method, CalcElectricalAngle for example, without knowing how derived the class is implementing it.

Interface

Objects define their interaction with the outside world through the method that they expose. Methods form the interface with the outside world allowing a class to become more formal about the behavior it promises and provides.

3 Advantages of object-oriented programming

This section describes the fundamental concepts and features of OOP, and the benefits of this type of approach with particular reference to STM32 FOC PMSM SDK V3.0.

3.1 Efficient multiple motor control

OOP makes it possible to create multiple instances of objects (for example, two object encoders) without duplicating the footprint of the flash memory necessary to handle them. This efficiency of OOP in terms of code size is even more marked when exploiting inheritance. Taking the example discussed in the previous section as a reference, the method GetElSpeedDpp is linked in the executable only once, no matter how many instances have been created of the derived classes encoder, hall or state observer.

3.2 Increased safety through data hiding

Object variables are bound to the object and only accessible through the object methods. This prevents the object variables from being accidentally modified, improving robustness for the final applications (fuel pumps, electric traction or applications related to human safety, for example).

3.3 Modularity

The source code for a class implementation can be written and maintained separately from other classes. This means that new versions of classes may be released separately from the others on the condition that the class interface and the method behavior are not modified.

3.4 Abstraction

You only need to know the object interface so you can focus on specific software developments.

4 STM32 PMSM FOC FW library C implementation of OOP

As a result of its desirable characteristics (code portability and efficiency, ability to access specific hardware addresses, low runtime demand on system resources, for example), the C language is widely used in embedded system applications. On the other hand, the C language, unlike more complex languages such as C++ and Java, does not support object oriented programming. For this reason a dedicated implementation of OOP has been developed in C for the STM32 PMSM FOC FW library v3.0.

4.1 Generic classes source files organization and content

Depending on the proposed implementation, a class Example is generally composed of three source files:

ExampleClass.h

Located in the \MC library interface folder, this is the public header file that contains the interface of the class Example. As mentioned previously, the interface of a class exports the definitions of the methods applicable to the objects of that class. In general, in the STM32 PMSM FOC FW library implementation, this file contains everything necessary to work with that class. For this purpose, this file contains the public definition of the class type (CEXMP) and the type structure containing the constant parameters required for the object creation (`ExampleParams_t`).

In addition, and only if necessary, definitions of certain types required for using methods are stored in this file.

```
/*******************************************************************
  * @file    ExampleClass.h
  * @author  IMS Systems Lab and Technical Marketing - MC Team
  * @version V0.0.1
  * @brief   This file contains interface of Example class
********************************************************************
*/

/* Includes ------------------------------------------------------*/
#include "MC_type.h"

/**
  * @brief  Public Example class definition
  */
typedef struct CEXMP_t *CEXMP;

/**
  * @brief  Example class parameters definition
  */
typedef const struct
{
  unsigned int paramA; /*!< Example of parameter */
}ExampleParams_t, *pExampleParams_t;
```

```
/**
  * @brief  Creates an object of the class Example
  * @param  pExampleParams pointer to an Example parameters
structure
  * @retval CEXMP new instance of Example object
  */
CEXMP EXMP_NewObject(pExampleParams_t pExampleParams);

/**
  * @brief  Example of public method of the class Example
  * @param  this related object of class CEXMP
  * @retval none
  */
void EXMP_Func(CEXMP this);

/**
  * @brief  Example of virtual method of the class Example
implemented by derived class
  * @param  this related object of class CEXMP
  * @retval none
  */
void EXMP_VFunc(CEXMP this);
```

It is worth noticing that class type *CEXMP* is a pointer to a void structure (whose type is `CEXMP _t`). This prevents the user of the class from accessing object members and hidden data.

ExamplePrivate.h

Located in the \MC library\inc (available only for confidential distribution of STM32 FOC PMSM SDK V3.0), this is a class private header file that contains private definitions required by the class implementation. It contains definitions of object data structure type (object variable elements of this structure), virtual methods container structure (only for classes with derived, see next paragraph), parameters class private re-definition and the private class definition.

```
/**
******************************************************************
* @file    ExamplePrivate.h
* @author  IMS Systems Lab and Technical Marketing - MC Team
* @version V0.0.1
* @brief   This file contains private definition of Example class
******************************************************************
*/

/**
  * @brief  Example class members definition
  */
typedef struct
{
  unsigned int base_vars; /*!< Example of member */
}Vars_t,*pVars_t;
```

```
/**
  * @brief  Redefinition of parameter structure
  */
typedef ExampleParams_t Params_t, *pParams_t;

/**
  * @brief Virtual methods container
  */
typedef struct
{
void (*pIRQ_Handler)(void *this, unsigned char flag); /*!< Only if
class implementation requires to be
triggered by an interrupt */
  void (*pVFunc)(CEXMP this); /*!< Example of virtual function
pointer */ }Methods_t,*pMethods_t;

/**
  * @brief  Private Example class definition
  */
typedef struct
{
  Methods_t Methods_str ;/*!< Virtual methods container */
  Vars_t Vars_str; /*!< Class members container */
  pParams_t pParams_str; /*!< Class parameters container */
  void *DerivedClass;/*!< Pointer to derived class */
}_CEXMP_t, *_CEXMP;
```

If either the base or derived class implementation requires the execution of program lines to be triggered by an interrupt, a pointer to those program lines (pIRQ_Handler) is also defined. See *Section 4.4: Motor control library related interrupt handling* for more information about MC library IRQ handler management.

ExampleClass.c

Located in the \MC library\src (available only for confidential distribution of STM32 FOC PMSM SDK V3.0), this is the source file that contains the implementation of class methods. This file includes both the interface and the private definitions of the same class.

The method Example_NewObject merits some explanation. This method creates objects of class Example (CEXMP) on demand.

Two different implementations of Example_NewObject are proposed, depending on the availability of the definition *MC_CLASS_DYNAMIC in MCLibraryConf.h*. If *MC_CLASS_DYNAMIC* is defined, dynamic RAM allocation is enabled and objects are created through standard library subroutine calloc, resulting in an efficient exploitation of RAM memory. This approach is not compatible with MISRA C 2004 rules compliancy because of the potential risks of memory leaks and memory corruption introduced by dynamic memory allocation.

On the contrary, dynamic memory allocation is disabled when user comments the *MC_CLASS_DYNAMIC* definition. In this case, an array of objects is statically and previously allocated in RAM. The list of the number of objects that are reserved for each of the classes is defined in MCLibraryConf.h for both single motor and dual motor (MAX_EXMP_NUM and similar). In order to prevent the compiler from reserving RAM

memory for objects that will never be created, you can edit pool dimension accordingly to the final application.

Pool dimension tailoring is only permitted in STM32 FOC PMSM SDK v3.0 confidential distribution. In the case of web distribution, no additional objects can be instanced by the user. Only the following exceptions are allowed: up to 3 PID objects, up to 5 PI objects, up to 5 digital output objects.

```
/**
******************************************************************
  * @file    ExampleClass.c
  * @author  IMS Systems Lab and Technical Marketing - MC Team
  * @version V0.0.1
  * @brief   This file contains interface of Example class
******************************************************************
*/

#include "ExampleClass.h"
#include "ExamplePrivate.h"
#include "MCLibraryConf.h"
#include "MC_type.h"

#ifdef MC_CLASS_DYNAMIC
  #include "stdlib.h" /* Used for dynamic allocation */
#else
  _CEXMP_t EXMPpool[MAX_EXMP_NUM];
  unsigned char EXMP_Allocated = 0u;
#endif

/**
  * @brief  Creates an object of the class Example
  * @param  pExampleParams pointer to an Example parameters
structure
  * @retval CEXMP new instance of Example object
  */
CEXMP EXMP_NewObject(pExampleParams_t pExampleParams)
{
  _CEXMP _oEXMP;

  #ifdef MC_CLASS_DYNAMIC
    _oEXMP = (_CEXMP)calloc(1u,sizeof(_CEXMP_t));
  #else
    if (EXMP_Allocated  < MAX_EXMP_NUM)
    {
      _oEXMP = &EXMPpool[EXMP_Allocated++];
    }
    else
    {
      _oEXMP = MC_NULL;
    }
  #endif
  _oEXMP->pParams_str = (pParams_t)pExampleParams;
  return ((CEXMP)_oEXMP);
```

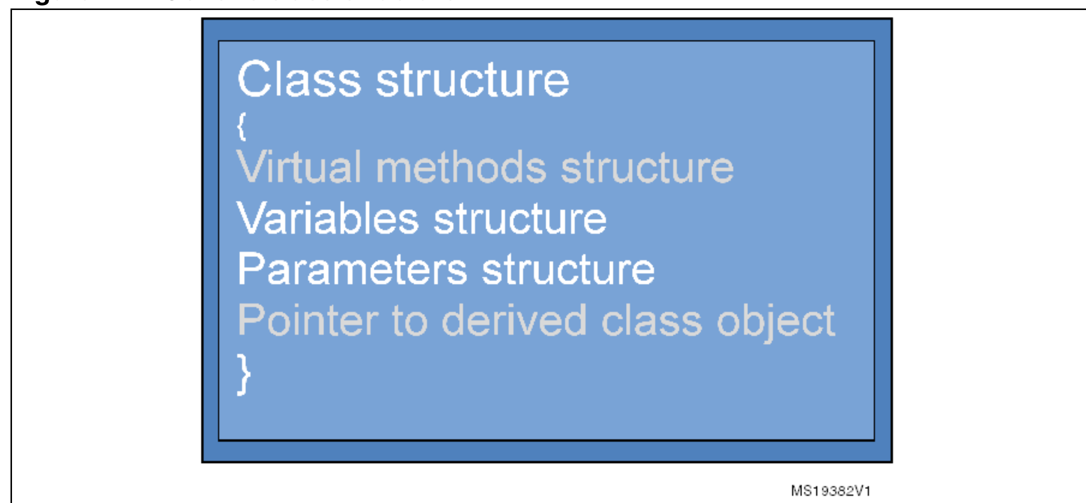
```
}

/**
  * @brief  Example of public method of the class Example
  * @param  this related object of class CEXMP
  * @retval none
  */
void EXMP_Func(CEXMP this)
{
  ((_CEXMP)this)->Vars_str.base_vars = 0u;
}

/**
* @brief  Example of virtual method of the class Example implemented
by derived *     class
  * @param  this related object of class CEXMP
  * @retval none
  */
void EXMP_VFunc(CEXMP this)
{
  ((_CEXMP)this)->Methods_str.pVFunc(this);
}
```

4.2 Inheritance implementation

As discussed previously, inheritance is one of the fundamental features of object oriented programming. This section describes how it has been achieved in the STM32 PMSM FOC SDK V3.0.

Figure 1 summarizes the private content of a generic class in the proposed implementation.

Figure 1. Generic class structure

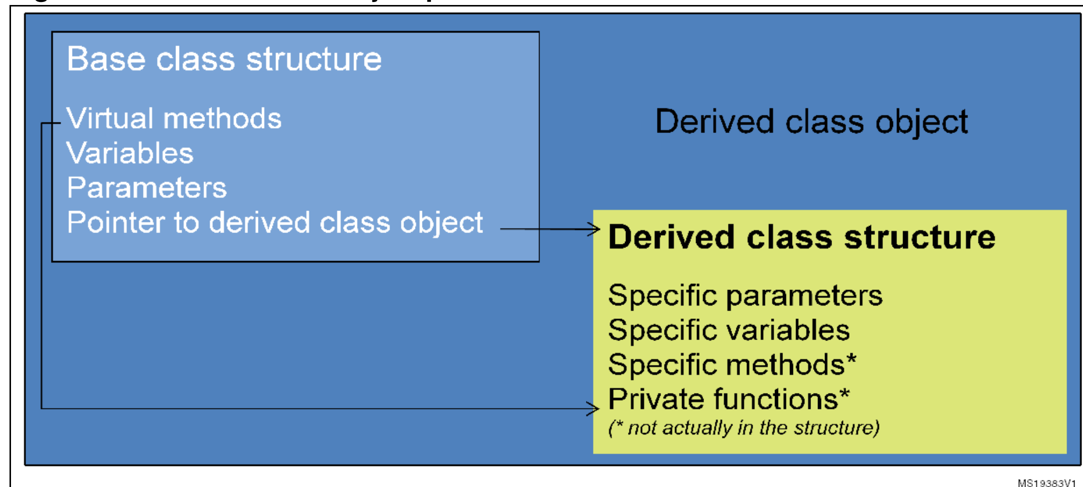


Not used in classes with no derived class objects, virtual methods structure and pointers to derived classes are keys to understanding inheritance accomplishment. Virtual method structure contains a list of pointers to those functions that - once properly initialized in the

derived class object creation process - link virtual methods exported by base class interface, together with their private implementation contained in each of the derived classes.

The pointer to a derived class object allows composing a derived class object by merging both its base and derived class portions as shown in *Figure 2*.

Figure 2. Derived class object private structure



The derived class portion of a derived class object is always accessed through its base class portion, which represents the public entry point for both base and derived class specific members.

4.3 Derived classes source file organization and content

In order to complete the picture of derived class source files, templates are shown here for the class Derived, derived from the base class Example.

Derived_ExampleClass.h

Located in \MC library interface, this is the public header file that contains the interface of the class Derived_Example. As for *ExampleClass.h*, this header file contains everything necessary to work with the related class. This file contains methods specific of the derived class, the public definition of the derived class type and the type structure that contains the constant parameters required to create the derived class object.

In addition and only if necessary, this file contains definitions of certain types required for using methods.

Creating a new instance of a derived class object requires pointers to both base and derived classes parameter structures (see also *Derived_ExampleClass.c*).

```
/**
******************************************************************
  * @file    Derived_ExampleClass.h
  * @author  IMS Systems Lab and Technical Marketing - MC Team
  * @version V0.0.1
  * @brief   This file contains interface of Derived class
******************************************************************
*/
#include "MC_type.h"
```

```
/**
  * @brief  Public Derived class definition
  */
typedef struct CDRV_EXMP_t *CDRV_EXMP;

/**
  * @brief  Derived class parameters definition
  */
typedef const struct
{
  unsigned int param1; /*!< Example of parameter */
}DerivedParams_t, *pDerivedParams_t;

/**
  * @brief  Creates an object of the class Derived
  * @param  pExampleParams pointer to an Example parameters
structure
  * @param  pDerivedParams pointer to an Derived parameters
structure
  * @retval CDRV_EXMP new instance of Derived object
  */
CDRV_EXMP DRV_NewObject(pExampleParams_t pExampleParams,
pDerivedParams_t pDerivedParams);
/**
  * @brief  Example of public method of the class Derived
  * @param  this related object of class CDRV_EXMP
  * @retval none
  */
void DRV_Func(CDRV_EXMP this);
```

Derived_ExamplePrivate.h

Located in \MC library\inc (available only for confidential distribution of STM32 FOC PMSM SDK V3.0), this is a class private header file that contains private definitions required for the derived class implementation.It contains the private definition of an object data structure type (object variables are elements of this structure), parameter class private redefinition and the private class definition.

Unlike the related base class private definition header file, a derived class structure type does not contain pointers to both further derived classes and virtual method containers. This limits levels of inheritance to one.

```
/** ******************************************************************
  * @file    Derived_ExamplePrivate.h
  * @author  IMS Systems Lab and Technical Marketing - MC Team
  * @version V0.0.1
  * @brief   This file contains private definition of Derived class
********************************************************************
*/

/* Define to prevent recursive inclusion -------------------------*/
#ifndef __DERIVED_EXAMPLEPRIVATE_H
#define __DERIVED_EXAMPLEPRIVATE_H
```

```
/**
  * @brief  Derived class members definition
  */
typedef struct
{
  unsigned int derived_Vars; /*!< Example of member */
}DVars_t,*pDVars_t;

/**
  * @brief  Redefinition of parameter structure
  */
typedef DerivedParams_t DParams_t, *pDParams_t;

/**
  * @brief Private Derived class definition
  */
typedef struct
{
  DVars_t DVars_str;/*!< Derived class members container */
  pDParams_t pDParams_str;/*!< Derived class parameters container
*/
}_DCDRV_EXMP_t, *_DCDRV_EXMP;
```

Derived_ExampleClass.c

Located in \MC library\src (available only for confidential distribution of STM32 FOC PMSM SDK V3.0), this is the source file that contains the implementation of both derived class specific methods and base class virtual methods. It includes both base and derived classes interface and private definitions.If the derived class requires the execution of program lines to be triggered by an interrupt, the file *MCIRQHandlerPrivate.h* is also included (refer to *Section 4.4: Motor control library related interrupt handling* for further information about interrupt handling).

The method DRV_NewObject merits mentioning. This method creates objects of the class Derived_Example (CDRV_EXMP) on demand and requires the pointers to both the parameters structure of base and derived classes as input. The creation of a derived class object encloses the creation of the related base class object. The two objects are then merged by initializing the base class pointer to the derived class object (_oExample->DerivedClass) with the address of the newly created derived class object (_oDerived). The base class pointers to the virtual methods and—if any—to the MC IRQ handler are also initialized with pointers to derived class private functions. The address of the base class portion of the derived class object is cast to the public derived class type (CDRV_EXMP) and returned.

```
/** ******************************************************************
  * @file    Derived_ExampleClass.c
  * @author  IMS Systems Lab and Technical Marketing - MC Team
  * @version V0.0.1
  * @brief This file contains private impelementation of Derived
class
********************************************************************
*/
```

```
#include "ExampleClass.h"
#include "ExamplePrivate.h"
#include "Derived_ExampleClass.h"
#include "Derived_ExamplePrivate.h"
#include "MCLibraryConf.h"
#include "MC_type.h"
#include "MCIRQHandlerPrivate.h" /*!< Only if derived class
implementation requires to be triggered by an interrupt */
#ifdef MC_CLASS_DYNAMIC
  #include "stdlib.h" /* Used for dynamic allocation */
#else
  _DCDRV_EXMP_t DRV_EXMPpool[MAX_DRV_EXMP_NUM];
  unsigned char DRV_EXMP_Allocated = 0u;
#endif

static void DRV_VFunc(CEXMP this);

/**
  * @brief  Creates an object of the class Derived
  * @param  pExampleParams pointer to an Example parameters
structure
  * @param  pDerivedParams pointer to an Derived parameters
structure
  * @retval CDRV_EXMP new instance of Derived object
  */
CDRV_EXMP DRV_NewObject(pExampleParams_t pExampleParams,
pDerivedParams_t pDerivedParams)
{
  _CEXMP _oExample;
  _DCDRV_EXMP _oDerived;

  _oExample = (_CEXMP)EXMP_NewObject(pExampleParams);

  #ifdef MC_CLASS_DYNAMIC
    _oDerived = (_DCDRV_EXMP)calloc(1u,sizeof(_DCDRV_EXMP_t));
  #else
    if (DRV_EXMP_Allocated  < MAX_DRV_EXMP_NUM)
    {
      _oDerived = &DRV_EXMPpool[DRV_EXMP_Allocated++];
    }
    else
    {
      _oDerived = MC_NULL;
    }
  #endif
  _oDerived->pDParams_str = pDerivedParams;
  _oExample->DerivedClass = (void*)_oDerived;
  _oExample->Methods_str.pVFunc = &DRV_VFunc;
  _oExample->Methods_str.pIRQ_Handler = &DRV_IRQHandler;
  Set_IRQ_Handler(pDerivedParams->IRQno, (_CMCIRQ)_oExample);
  return ((CDRV_EXMP)_oExample);
}
```

```
/**
  * @brief  Example of private method of the class Derived to
implement a virtual
  *      function of class Example
  * @param  this related object of class CEXMP
  * @retval none
  */
static void DRV_VFunc(CEXMP this)
{
   ((_DCDRV_EXMP)(((_CEXMP)this)->DerivedClass))-
>DVars_str.derived_Vars = 0u;
}
/**
  * @brief  Example of public method of the class Derived
  * @param  this related object of class CDRV_EXMP
  * @retval none
  */
void DRV_Func(CDRV_EXMP this)
{
   ((_DCDRV_EXMP)(((_CEXMP)this)->DerivedClass))-
>DVars_str.derived_Vars = 0u;
}
/**
  * @brief  Example of private method of the class Derived to
implement an MC IRQ function
  * @param  this related object
  * @param  flag used to distinguish between various IRQ sources
  * @retval none
  */
static void DRV_IRQHandler(void *this, unsigned char flag)
{
  if (flag==1u)
  {
    ((_DCDRV_EXMP)(((_CEXMP)this)->DerivedClass))-
>DVars_str.derived_Vars++;
  }
}
```

4.4 Motor control library related interrupt handling

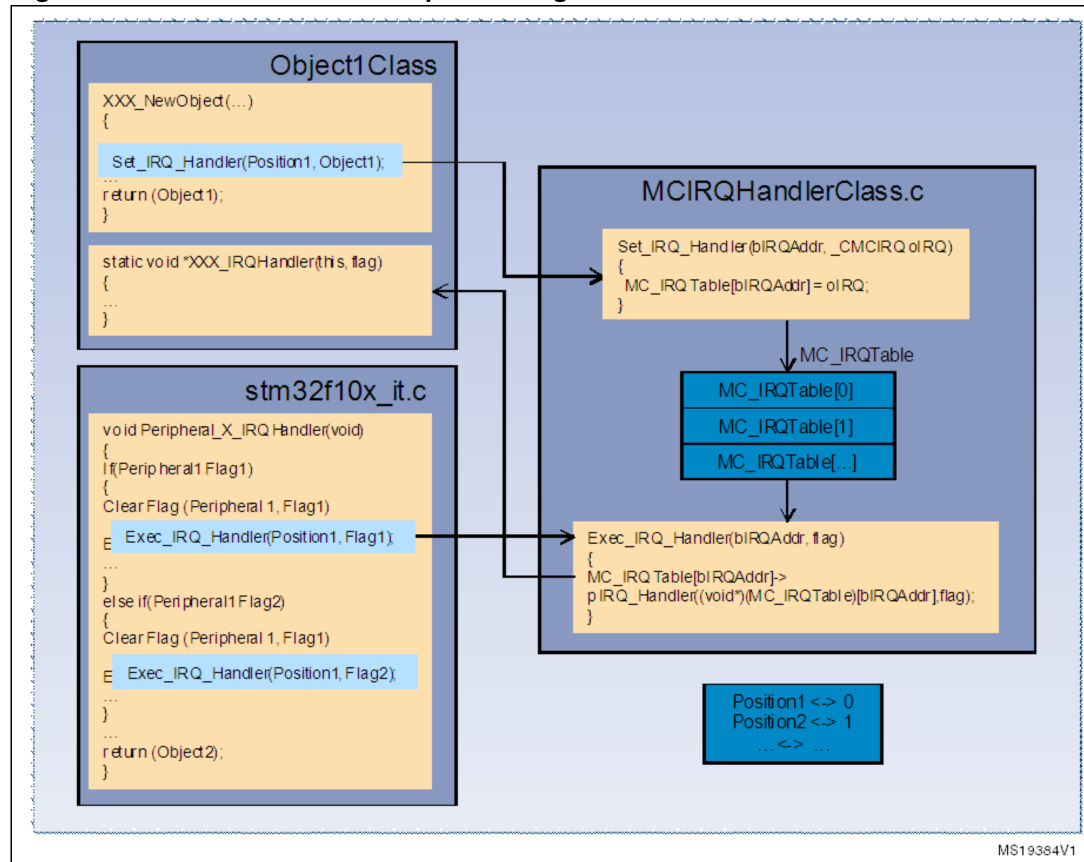
The implementation of certain classes (such as speed or current sensors) may require the execution of specific program lines (referenced below as MC IRQ Handler) when a specific event occurs, exploiting the related Interrupt Service Routine (ISR).

The same ISR must also be available at the User project level (see also UM1052) to permit customization of application software by adding personal code lines in the same ISR.

In order to keep the motor control library and the user project layers separate, it is necessary to implement a mechanism that enables triggering the execution of MC IRQ Handlers bundled within a given class without permitting any explicit reference to the motor control library objects from the user layer.

With this mechanism, *module stm32f10x_MC_it.c* (containing the definitions of all the IRQ Handlers that require certain MC code lines to be executed) is put at the disposal of the user by including it at the user project level. Both the *stm32f10x_MC_it.c* and the motor control libraries include a module, MCIRQHandlerClass, which privately holds a motor control vector table (MC_IRQTable) that contains the set of those objects that need to be triggered by an ISR. The filling of a given position in the table is performed when the corresponding object is created (inside the related XXX_New_Object method) by means of the Set_IRQ_Handler function call. *Figure 3* illustrates this process.

Figure 3. Motor control interrupt handling



When an interrupt event occurs, the related ISR (Peripheral_X_IRQHandler) is accessed. After clearing the proper interrupt flag and optionally executing user defined code lines, the function Exec_IRQ_Handler is called if it is required to execute a MC IRQ Handler.

In order to identify the MC IRQ handler to be executed, the MC_IRQTable position that corresponds with the proper object is passed as a function parameter (so *stm32f10x_MC_it.c* does not require object interface knowledge). Furthermore, as a MC IRQ Handler can be in general accessed from more than one interrupt, a flag that identifies the triggering event is also passed.

Once the object owner of the MC IRQ Handler to be executed has been identified by accessing the MC IRQ Table in the position passed to Exec_IRQ_Handler, this function can finally jump to the MC IRQ Handler itself.

The entire process, taking the program flow from Peripheral_X_IRQHandler to the MC IRQ Handler only requires two jumps: to Exec_IRQ_Handler and then to the MC IRQ Handler. In

this way, the overhead introduced by the SW architecture is minimized. This is achieved by making the addresses of both the object and its related MC IRQ handler (which is located in the first element of the class structure, as already shown in ExamplePrivate.h template) the same.

5 How to create a user defined class

Users can create their own classes and add them to the motor control library. To do this, use the templates described in *Section 4.1* for base classes and *Section 4.3* for derived classes.

If the newly created class requires the execution of a MC IRQ Handler on an interrupt occurrence, the MAX_MC_IRQ_NUM definition in MCIRQHandlerClass.c must be incremented and the corresponding MC IRQ table position defined, by adding the following line in *MCIRQHandlerClass.h*, for example:

```
#define MC_IRQ_USER_IRQ  4u
```

Note: The first four table positions are reserved for PWMnCurrFdbk (first and second instances) and SpeednPosFdbk (first and second instance) objects. In case of STM32 FOC PMSM SDK V3.0 web distribution, the maximum number of elements for the MC IRQ table is limited to 8 (elements 0 to 3 are already reserved and not available for the user, elements 4 to 7 are left for the user).

Add the function call *Exec_IRQ_Handler(MC_IRQ_USER_IRQ, flag*) in *stm32f10x_MC_it.c* in the proper peripheral IRQ handler function body. The flag is the identifier for the interrupt trigger event.

6 STM32 PMSM FOC FW library v 3.0 class list

This section provides a general view and a short description of the classes used in the MC library. For a detailed description of the methods and parameters of each of the classes, see *STM32 FOC PMSM FW library v3_0 developer Help file.chm*.

Note: *Source files of the MC library classes are only provided free of charge within STM32 FOC PMSM SDK V3.0 confidential distribution. Contact your nearest ST sales office or support team for further information.*

6.1 Current reading and PWM generation (CPWMC) and its derived classes

This class implements both the functionality of the current reading sensor and PWM generator. Any object of this class must be linked to a derived class object.

In order to increase the modularity of the library, access to the MCU peripherals has been moved to the derived classes, which have been additionally differentiated by the hardware current sensing topology. The derived classes are:

Table 1. Derived classes

Class	Definition
R1_VL1 (CR1VL1_PWMC)	Current sensing carried out via a single shunt resistor placed on the DC bus link and implemented on a STM32F100x (value line devices). It only supports a single motor drive.
R1_LM1 (CR1LM1_PWMC)	Current sensing carried out via a single shunt resistor and implemented on a STM32F103x x= 4, 6, 8, B (performance line, low and medium density devices). It only supports a single motor drive.
R1_HD2 (CR1HD2_PWMC)	Current sensing carried out via a single shunt resistor and implemented on a STM32F103x x= C, D, E (performance line, high density devices). Although it is designed to support dual motor drive, it can also be used when a single motor drive has been instanced.
R3_LM1 (CR3LM1_PWMC)	Current sensing carried out via three shunt resistors placed below low side switches on the three inverter legs and implemented on a STM32F103x x= 4, 6, 8, B (performance line, low and medium density devices). It only supports a single motor drive.
R3_HD2 (CR3HD2_PWMC)	Current sensing carried out via three shunt resistors placed below low side switches on the three inverter legs and implemented on a STM32F103x x= C, D, E (performance line, high density devices). Although it is designed to support dual motor drive, it can also be used when a single motor drive has been instanced.

Table 1. Derived classes

Class	Definition
ICS_LM1 (CILM1_PWMC)	Current sensing carried out through isolated current sensors and implemented on a STM32F103x x= 4, 6, 8, B (performance line, low and medium density devices). It only supports a single motor drive.
ICS_HD2 (CIHD2_PWMC)	Current sensing carried out through isolated current sensors and implemented on a STM32F103x x= C, D, E (performance line, high density devices). Although it has been specifically designed to support dual motor drive, it can also be used when a single motor drive has been instanced.

6.2 Speed and position feedback (CSPD) and its derived classes

This class carries out the speed/position sensor handling for both physical or FW emulated sensors. Any object of this class must be linked to a derived class object.

Access to hardware peripherals, if there is any, is asked to derived classes which are differentiated according to type of speed/position sensor. In the STM32 PMSM FOC FW library v3.0, hall sensors, quadrature encoder and sensorless are supported:

Table 2. Speed and position feedback (CSPD) and its derived classes

Class	Definition
ENCODER (CENC_SPD)	This derived class supports quadrature encoder and can be used with any STM32F103x or STM32F100x. By default, index signal is not handled.
HALL (CHALL_SPD)	This derived class supports three hall sensors. It can be used with any STM32F103x or STM32F100x.
STO (CSTO_SPD)	This derived class implements sensorless rotor position reconstruction based on current feedbacks, bus voltage and applied motor phase voltages information. The sensorless algorithm consists of a Luenberger state observer and a PLL.

Table 2. Speed and position feedback (CSPD) and its derived classes

Class	Definition
STO_CORDIC (CSTOC_SPD)	This derived class implements sensorless rotor position reconstruction based on current feedbacks, bus voltage and applied motor phase voltages information. The sensorless algorithm consists of a Luenberger state observer and an iterative algorithm for trigonometric arctg function computation.
Virtual speed sensor (CVSS_SPD)	This derived class is used mainly during ramp-up if an object of one of the sensor-less speed/position classes (CSTO_SPD or CSTOC_SPD) is used as main speed sensor. Used in conjunction with a rev-up controller and a speed and torque controller, it allows customizing ramp-up. An object of this class emulates a real sensor during motor rev up by returning (on demand) a virtual angle and/or a virtual speed in accordance with the time base and the acceleration (set by derived class specific method VSPD_SetMecAcceleration).

6.3 Field-oriented control drive (CFOC) and its derived classes

This class implements Field -Oriented Control (FOC) and additional methods that may be required, by internal permanent magnet motors for example. Any object of this class must be linked to a derived class object.

Generally, the key methods for this class are *FOC_CurrController,* which carries out the current regulation (field orientation) and must be called at PWM frequency (or an integer sub-multiple), and *FOC_CalcCurrRef,* which with the derived class implementation and required electrical torque, updates the reference stator current components I_{qref} and I_{dref}.

This class does not contain references to peripherals and is thus hardware independent.

Derived classes are differentiated according to required additional methods:

Table 3. Field Oriented Control drive (CFOC) and its derived classes

Class	Description
SM (CSM_FOC)	Derived class designed for driving surface mounted motors. No additional methods have been here implemented
SMF (CSMF_FOC)	Derived classes used for surface magnet motors (SM-PMSM) when flux weakening is required
IMF (CIMF_FOC)	Derived classes used for internal permanent magnet motors (I-PMSM). Maximum-Torque-Per-Ampere (MTPA) and flux weakening additional methods are available for this class.
IMFF (CIMFF_FOC)	Derived classes used for internal permanent magnet motors (I-PMSM) high-end drives. Maximum-Torque-Per-Ampere (MTPA), flux weakening additional methods and auxiliary feed-forward current regulator are available for this class.

6.4 Bus voltage sensor (CVBS) and its derived classes

This class implements either a virtual or a real bus voltage, depending on sensor availability. Any object of this class must be linked to a derived class object.

If any, the access to MCU peripherals is asked from derived classes so that base class implementation is kept hardware independent. Derived classes are differentiated accordingly with the type of the physical sensor (if any):

Table 4. Bus voltage sensor (CVBS) and its derived classes

Class	Description
Rdivider (CRVBS_VBS)	Derived class that can handle all types of real voltage sensor with analog output. For example, hardware resistive voltage partitioning.
Virtual (CVVBS_VBS)	This derived class emulates a voltage sensor when no real sensors are available. It always returns a constant programmable voltage.

6.5 Temperature sensor (CTSNS) and its derived classes

This class implements either a virtual or real temperature sensor, depending on sensor availability. Any object of this class must be linked to a derived class object.

If any, the access to MCU peripherals is asked from derived classes so that base class implementation is kept hardware independent. Derived classes are differentiated accordingly with the type of the physical sensor (if any):

Table 5. Bus voltage sensor (CVBS) and its derived classes

Class	Description
NTC (CNTC_TSNS)	Derived class that can handle NTC sensor or more in general analog temperature sensors whose output is related to temperature by following formula: $V_{out} = V_0 + \frac{dV}{dT} \bullet (T - T_0)$
Virtual (CVTS_TSNS)	This derived class emulates a temperature sensor when no real sensors are actually available. It always returns a constant programmable temperature.

6.6 Digital Output (CDOUT) class

This class is used to abstract the concept of digital output driving from its hardware-dependent implementation. With particular reference to motor control, this class can be used to drive in- rush current limiter devices or handle resistive brake turn-on and turn-off for example.

6.7 Encoder Alignment Controller (CEAC) class

This class is used only if a quadrature encoder is used as main or auxiliary sensor. In conjunction with a virtual speed sensor, speed and torque controller and FOC drive objects this class handles the initial encoder calibration (which comprises a rotor alignment in a given position) necessary to make the information coming from a quadrature encoder absolute. See *Section 6.3* for more information about the alignment procedure.

In the case of dynamic allocation, the object may be destroyed after the alignment has been executed and created only when necessary.

6.8 Rev-up controller (CRUC) class

This class is only used if an object of one of the sensorless classes is used as main speed/position sensor. Used in conjunction with a speed and torque controller and a virtual speed sensor, this class enables complete customization of the motor phase current waveforms during motor ramp-up.

In the present implementation, the rev-up is divided into smaller portions called phases, where both speed and current amplitude can vary linearly. Each of the phases is characterized by its parameters (structure type RUCPhasesParams_t):

- duration (hDurationms)
- final motor speed (hFinalMecSpeed01Hz)
- final current amplitude (hFinalTorque)
- pointer to next rev-up phase parameters structure.

The Initial angle for the first phase can also be specified. See also *Section 6.2* for more information about ramp-up.

6.9 Speed and torque controller (CSTC) class

The speed and toque controller provides a FOC object with a target electrical torque depending on the control mode (speed or torque control) and executes target speed and torque ramps.

When in speed mode, the speed and toque controller computes the new target speed reference, if a ramp is being executed, and then performs the speed regulation loop. The return is an electrical torque, which is then used by the FOC object to get I_{qref} and I_{dref}.

When the speed and toque controller is in torque mode, it computes the new target electrical torque, if a ramp is being executed, and then returns target electrical torque.

6.10 State machine (STM) class

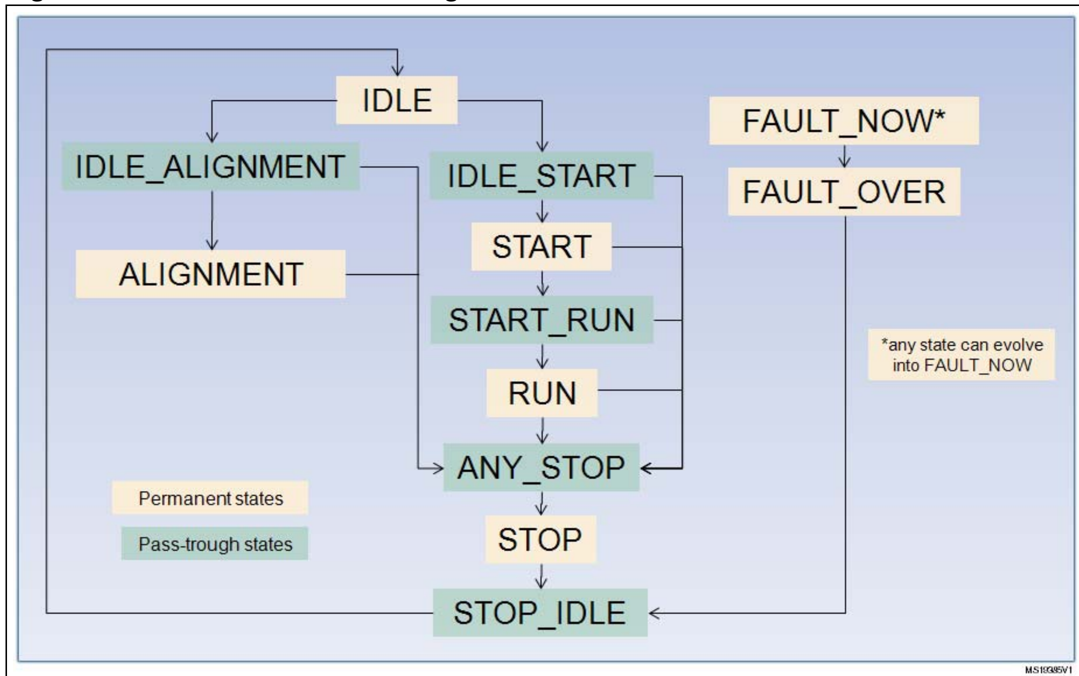
The state machine class handles transitions between the states of the drive that influence the actions that need to be taken by motor control tasks.

The following list of available states and a summarizing block diagram is provided for convenience.

Table 6. State machine (STM) class available states

State	Description
IDLE	Persistent state. The following state can be: – IDLE_START, if a start motor command has been given – IDLE_ALIGNMENT, if a start alignment command has been given
IDLE_ALIGNMENT	Pass-through state. The code to be executed only once between IDLE and ALIGNMENT states is executed here. The following state is usually ALIGNMENT but it can also be ANY_STOP if a stop motor command has been given.
ALIGNMENT	Persistent state. The following state is ANY_STOP.
IDLE_START	Pass-through state. The code to be executed only once between IDLE and START states is executed here. The following state is usually START but it can also be ANY_STOP if a stop motor command has been given.
START	Persistent state where the motor start-up is intended to be executed. The following state is usually START_RUN as soon as first validated speed is detected. ANY_STOP is also possible if a stop motor command has been executed.
START_RUN	Pass-through state. The code to be executed only once between START and RUN states is executed here. The following state is usually RUN, but it can also be ANY_STOP if a stop motor command has been given.
RUN	Persistent state with running motor. The following state is usually ANY_STOP when a stop motor command has been executed.
ANY_STOP	Pass-through state. The code to be executed only once between any state and STOP is executed here. The following state is usually STOP.
STOP	Persistent state. The following state is usually STOP_IDLE as soon as conditions for moving the state machine are detected.
STOP_IDLE	Pass-through state. The code to be executed only once between STOP and IDLE is executed here. The following state is usually IDLE.
FAULT_NOW	Persistent state. The state machine can be moved from any condition directly to this state by the STM_FaultProcessing method. As soon as all the fault conditions have disappeared, the state machine is moved into FAULT_OVER.
FAULT_OVER	Persistent state where the application is intended to stay after all the fault conditions have disappeared. The following state is usually STOP_IDLE. The state machine is moved as soon as the user has acknowledged the fault event.

Figure 4. State machine flow diagram



6.11 PI (CPI) and PID (CPID) controller classes

PI and PID controller classes realize PI and PID regulators respectively. The PID class is seen as a derived class from PI by adding the particular functionality of the derivative terms.

7 Class interaction

This section facilitates the understanding of the interactions between classes by describing how objects relate to achieve field orientation, speed and torque regulation, motor ramp-up and alignment.

7.1 Field orientation, speed and torque control procedures

Figure 5. Field orientation, speed and torque regulation

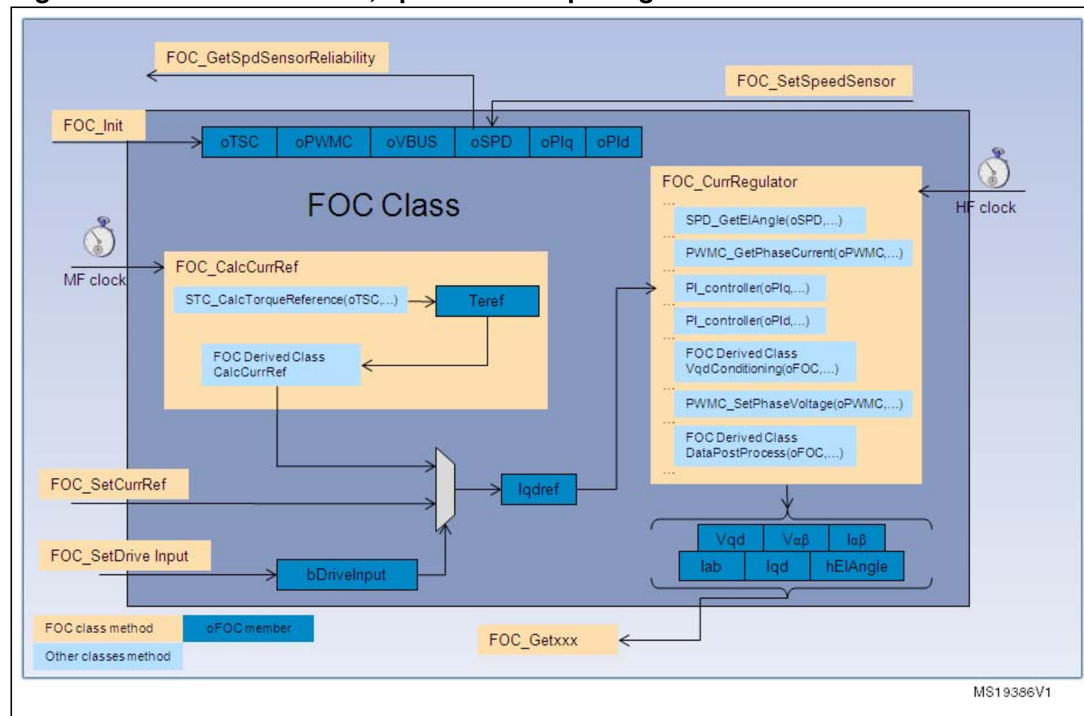


Figure 5 shows how the FOC drive class interacts with other classes in order to achieve both the speed and torque regulation and the field orientation. When the FOC drive object (oFOC) is initialized, the set of objects necessary to accomplish its duties are passed and stored in oFOC.

When the drive input is set to external (bDriveInput = EXTERNAL), stator current references can be provided from the outside via the method FOC_SetCurrRef. When the drive input is set to internal (bDriveInput = INTERNAL), the stator current reference components (Iqdref) are computed internally by the FOC_CalcCurrRef method (at the rate specified by medium frequency (MF) clock, which is 500Hz by default). This internal computation is performed in two steps:

1. The reference torque (Teref) is computed by the STC class method, STC_CalcTorqueReference, (running speed PI regulator when in speed mode, for example)
2. From Teref, I_{qref} and I_{dref} are computed by the FOC drive derived class method, CalcCurrRef, which implements MTPA and/or flux weakening if they are available.

Field orientation is executed at the rate specified by the High Frequency (HF) clock (equal to PWM frequency by default) using the FOC_CurrRegulator method. This method interacts with different objects (oPIq. oPId, oPWMC, oSPD) and computes the phase voltages to be applied to the motor with the purpose of achieving I_q and I_d regulation. As a result of the computation, the object members (Vqd, Vαβ, Iαβ, Iab, Iqd, hElAngle) are also updated.

7.2 Procedure for motor ramp-up for sensorless algorithms

Figure 6. Motor ramp-up procedure

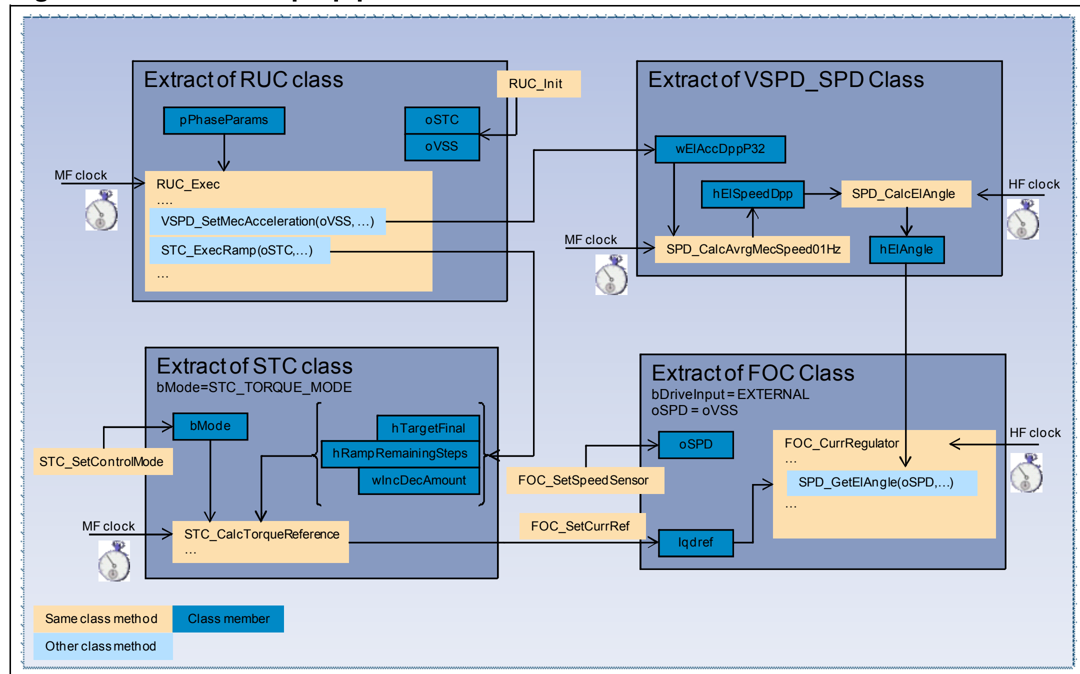


Figure 6 illustrates how motor rev-up is handled when STO_SPD or STOC_SPD objects are used as a main sensor. As already mentioned, the rev-up is divided into portions (also called 'phases' or 'stages') during which both the applied electrical frequency and the amplitude of the phase motor current change linearly.

Every time a new rev-up phase begins, the RUC_Exec method configures both the virtual speed sensor and the speed and torque controller in order to get the right electrical frequency and amplitude increases throughout the phase.

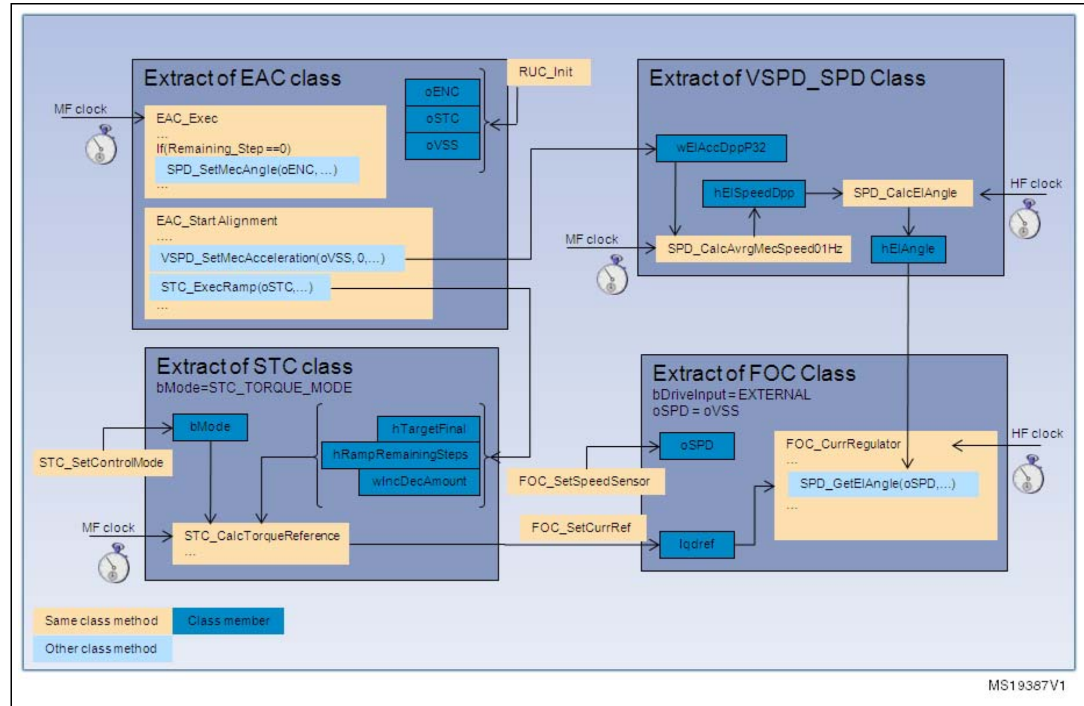
The electrical frequency increase is carried out by the virtual speed sensor which, at MF clock rate (default value 500Hz), updates the applied electrical frequency by integrating the acceleration (set by RUC via the VSPD_SetMecAcceleration method). The electrical angle is updated at the same time as the HF clock rate (default value is PWM frequency) by integrating the applied electrical frequency. As the oSPD held by the FOC drive object has been previously set to be equal to the virtual speed sensor object (oVss) using the SPD_SetSpeedSensor method, the oVss electrical angle is then used to orient stator current components I_q and I_d correctly.

In the meantime, the motor phase current target amplitude is also changed. This is handled by the method STC_CalcTorqueReference (clocked by MF clock) on the object oSTC (previously configured in STC_TORQUE_MODE for this purpose). The current component

references (I_{qdref}) provided by oSTC is fed to oFOC which is set in EXTERNAL mode so that it can accept such references.

7.3 Rotor alignment for encoder calibration

Figure 7. Rotor alignment for encoder calibration



The quadrature encoder is a relative position sensor. Because absolute information is required for performing field oriented control, it is necessary to establish a 0° position. This task is performed by means of an encoder calibration phase, which is carried out by default on user demand. This phase imposes a null reference flux (Id) and a torque reference flux (Iq) with a linearly increasing magnitude and a constant orientation.

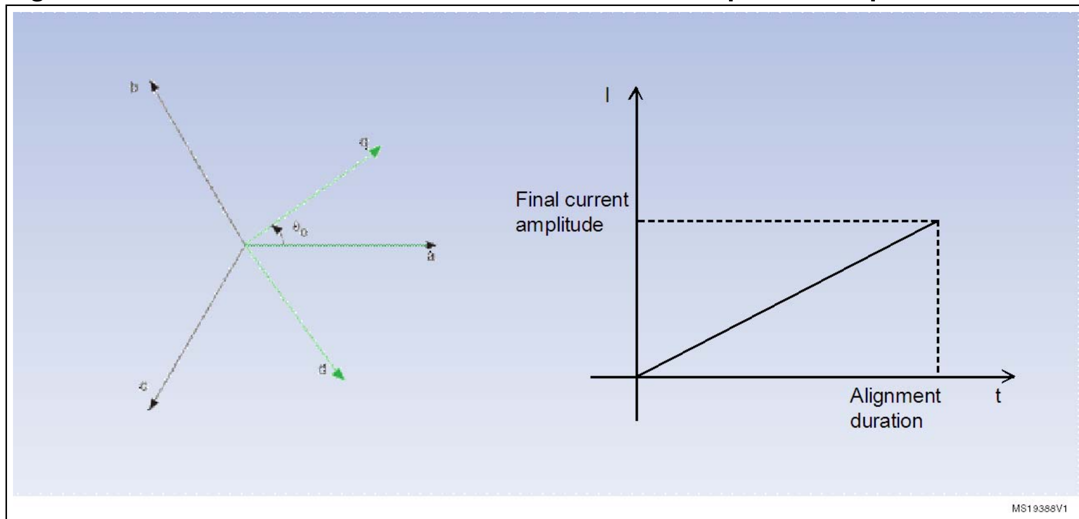
When properly configured, at the end of this phase, the rotor is locked in a well-known position and the encoder timer counter is initialized accordingly.

To perform this task (see *Figure 7*), the Encoder Alignment Controller (oEAC) configures a virtual speed sensor object in order to provide a constant programmable angle throughout the alignment duration. Mechanical acceleration is set equal to 0. oEAC also configures the speed and torque controller in STC_TORQUE_MODE and commands the start of a ramp with proper duration and final current amplitude. The STC_CalcTorqueReference method (clocked at MF) works as a ramp generator and its output is fed through the FOC_SetCurrRef method to the oFOC object (previously set in EXTERNAL mode).

As soon as the alignment duration is finished, oEAC initializes the speed/position sensor electrical angle correctly using the SPD_SetMechanicalAngle method.

Figure 8 illustrates the temporal Iq current amplitude variation and the convention used for the current orientation.

Figure 8. Stator current orientation convention and amplitude temporal variation



8 Description of tasks

This section describes the four tasks that are necessary for each of the motor(s) in order to manage the motor drives correctly.

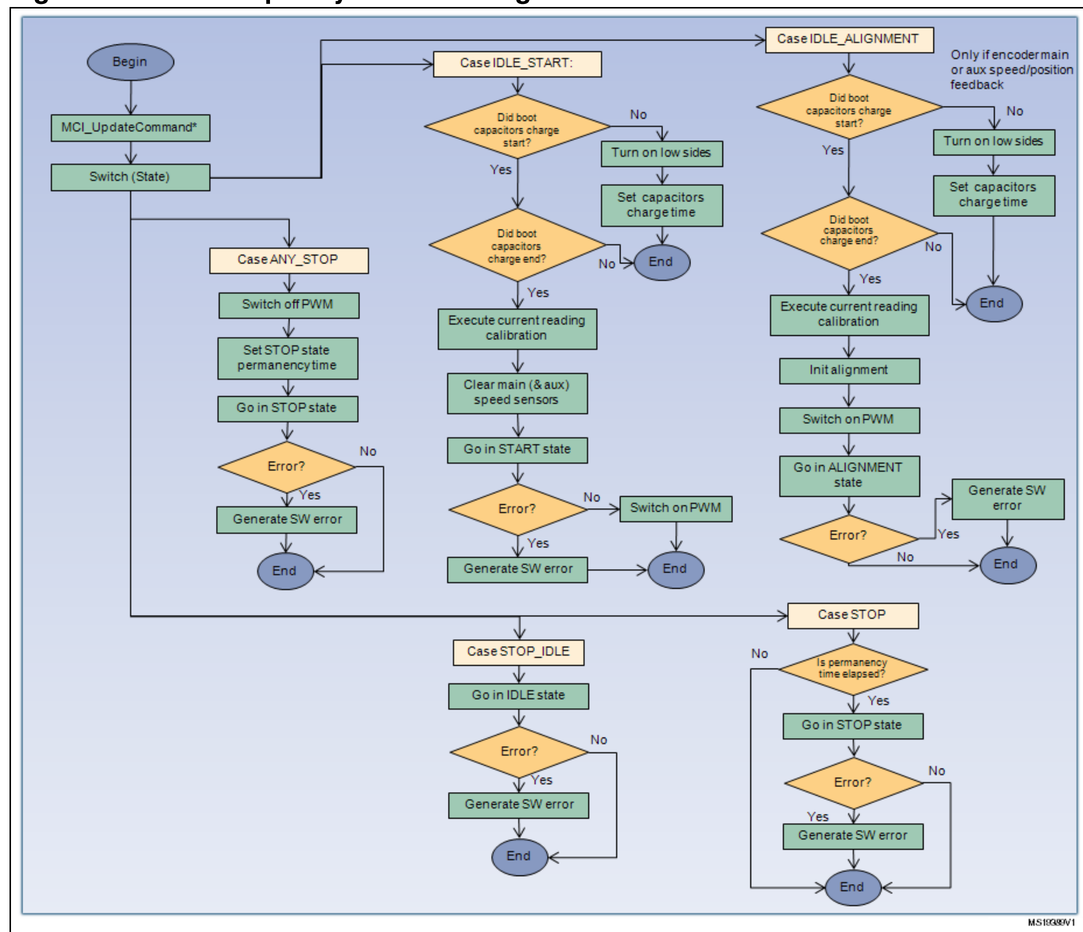
8.1 Low frequency task

The low frequency task executes the low frequency tasks related to each of the drives in sequence. It includes those duties that do not require a very precise timing and/or that need a low refresh rate, such as stop state permanency time or boot capacitors charge time counting. The default refresh rate is 100Hz and the priority should be set just above background (main) priority (tskIDLE_PRIORITY+1 in the case of FreeRTOS based applications, for example)

User commands such as run or stop motor are also processed in this task. Refer to (UM1052) for more information about user commands that can be provided to the MC application layer.

Figure 9 shows low frequency task flow diagram, please notice that IDLE_ALIGNMENT state is only available if the encoder is being used either as main or auxiliary sensor.

Figure 9. Low frequency task flow diagram

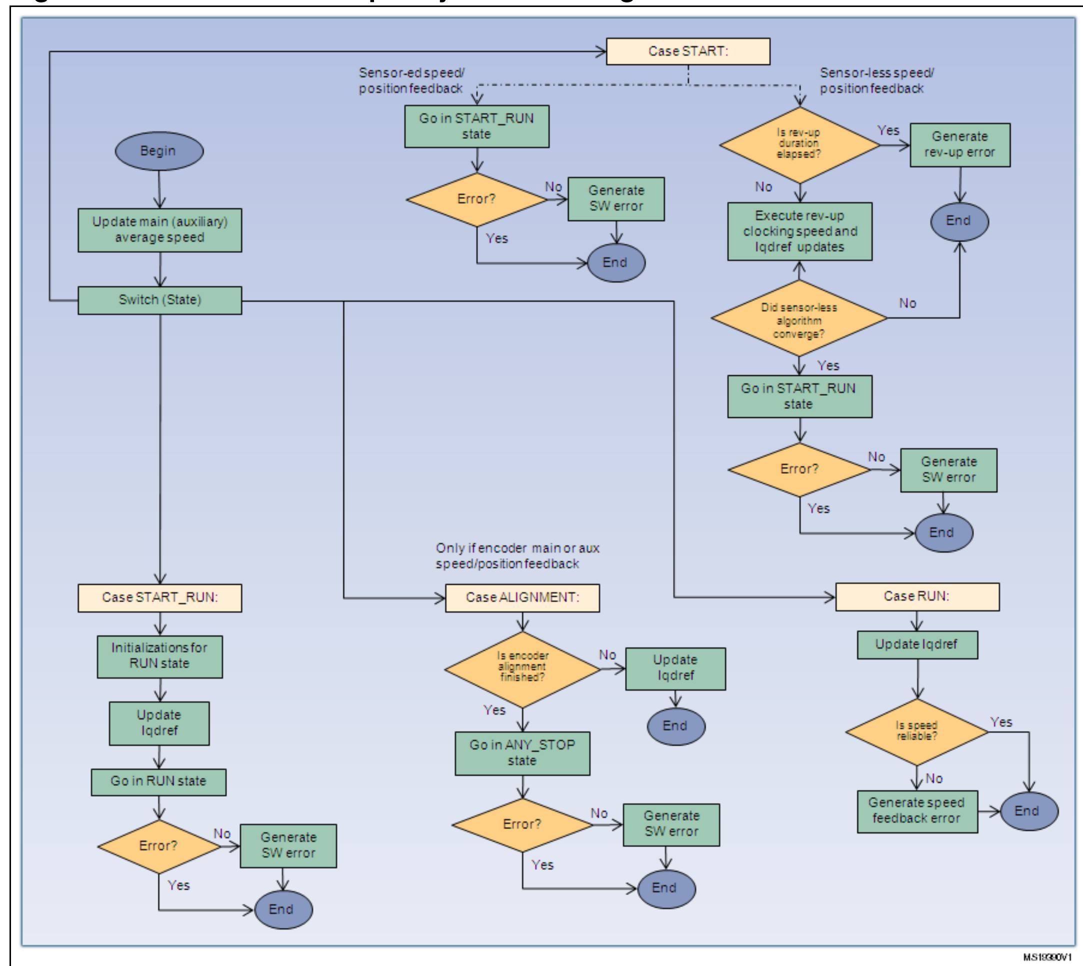


8.2 Medium frequency task

The medium frequency task executes the medium frequency tasks related to each of the drives in sequence. It executes certain control duties depending on the state of the related state machine. Duties requiring a specific timing, such as speed controller are executed with a default task refresh rate of 500Hz. To function correctly, the priority of this task must be higher that the low frequency task priority.

Figure 10 shows medium frequency task flow diagram.

Figure 10. The medium frequency task flow diagram



8.3 High frequency task

For a given motor and depending on the present state of the related state machine, the high frequency task executes the motor control duties that require a high frequency rate and precise timing such as FOC current control loop.

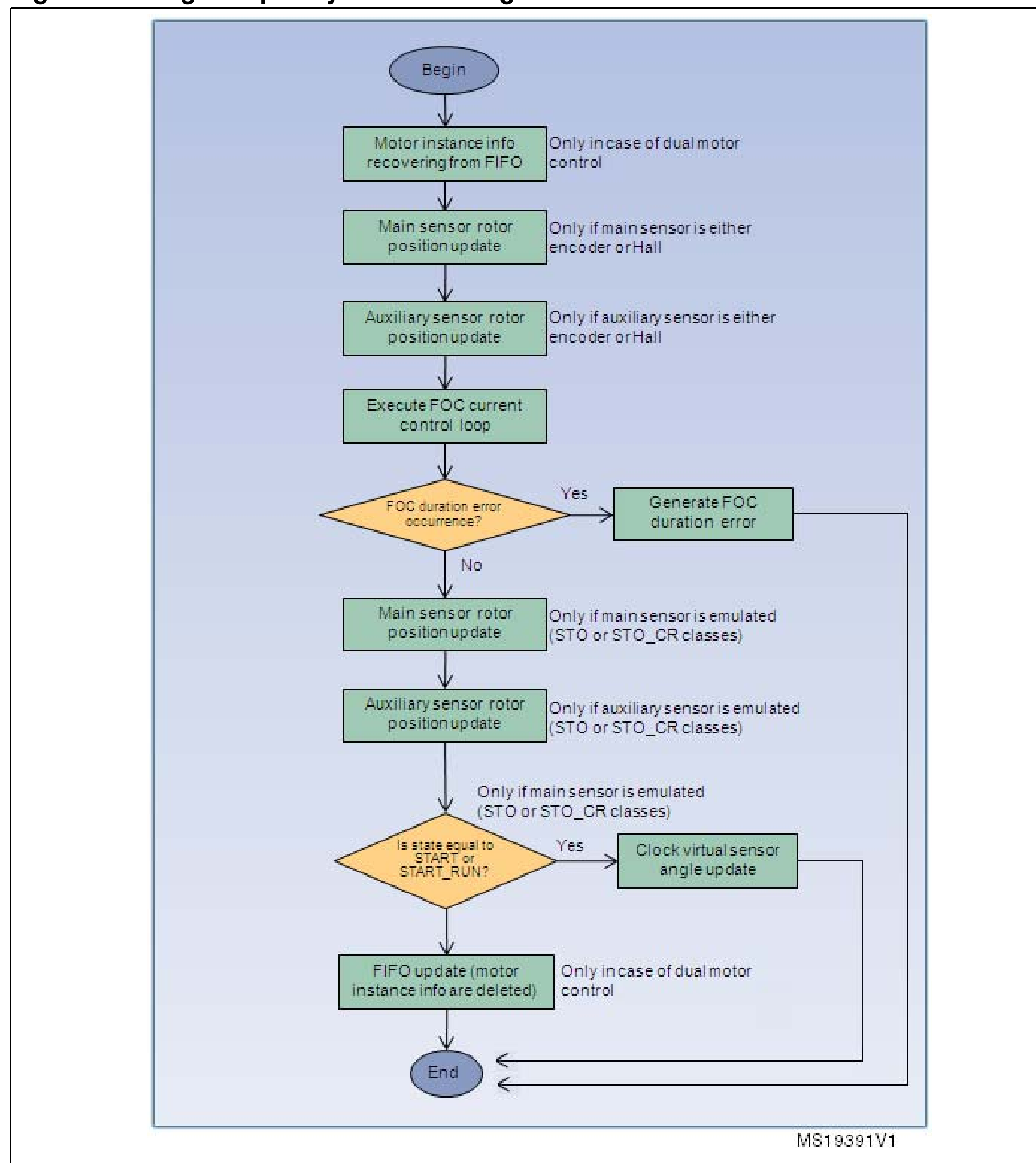
The high frequency task execution is triggered by the ADC JEOC interrupt, which sanctions the end of the related motor phase currents reading. Because this trigger is only available in the states START, START_RUN, IDLE_ALIGNMENT, ALIGNMENT, the high frequency task is only executed in these states and it is not triggered in the resting cases.

When being executed, the high frequency execution rate is strongly related to the PWM frequency. This execution rate can be computed as the corresponding drive PWM frequency divided by parameter REGULATION_EXECUTION_RATE in *Drive parameters.h* (for motor 1) or REGULATION_EXECUTION_RATE2 in *Drive parameters motor 2.h* (for motor 2).

In the case of dual motor control, a FIFO mechanism has been put in place in order to execute the FOCs of both the motors in the right sequence. The FOC execution related to a given motor is booked inside the TIMxUpdate ISR leading the A/D conversions for that motor currents reading by approximately half the PWM period.

In order to function correctly, the priority of this task must be set as the highest available in the application.

Figure 11. High frequency task flow diagram



8.4 Safety task

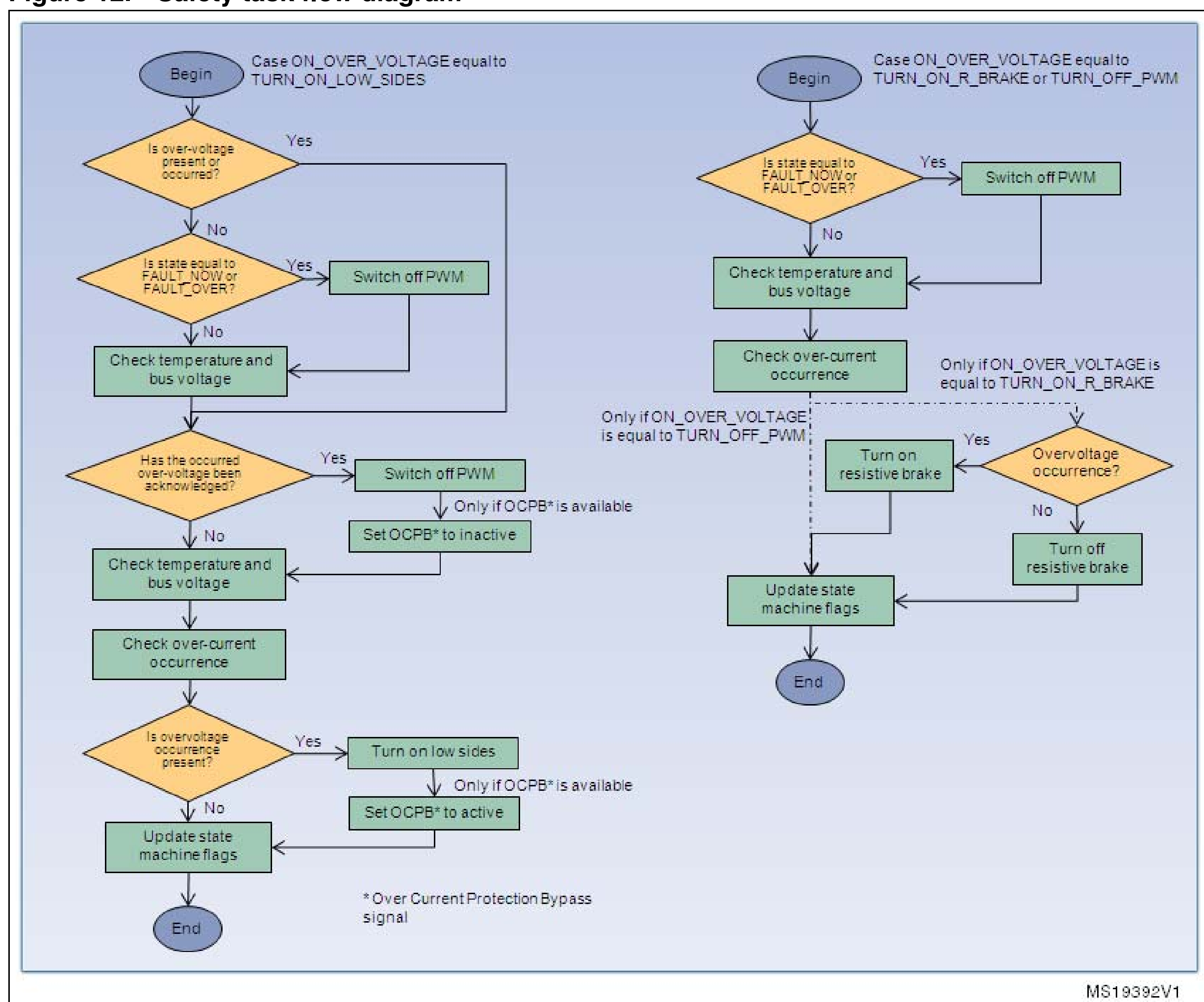
The safety task executes the safety checks (bus voltage and temperature, for example) related to each of the drives, in sequence. The actions to be taken in the case of over-voltage are managed here. These tasks are:

- turning on low side switches
- turning off PWM or turning on the brake resistor, depending on the ON_OVER_VOLTAGE definition in *Drive parameters.h.*

The default execution rate for this task is 2kHz.

Figure 12 shows the safety task flow diagram.

Figure 12. Safety task flow diagram



9 Bibliography

[1]Armstrong, The Quarks of Object-Oriented Development. In descending order of popularity, the "quarks" are: Inheritance, Object, Class, Encapsulation, Method, Message Passing, Polymorphism, Abstraction.

[2]Pierce, Benjamin (2002). MIT Press. ISBN 0-262-16209-1. , section 18.1 "What is Object-Oriented Programming?".

[3]John C. Mitchell, Concepts in programming languages, Cambridge University Press, 2003, SBN 0-521-78098-5, p.278.

[4]Michael Lee Scott, Programming language pragmatics, Edition 2, Morgan Kaufmann, 2006, ISBN 0-12-633951-1, p. 470 vikas.

[5]Abadi, Martin; Cardelli, Luca (1996). A Theory of Objects. Springer-Verlag New York, Inc.. ISBN 0387947752. Retrieved 2010-04-21.

10 Revision history

Table 7. Document revision history

Date	Revision	Changes
08-Apr-2011	1	Initial release.
24-May-2011	2	Added references for web and confidential distributions of STM32 FOC PMSM SDK v3.0

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS EXPRESSLY APPROVED IN WRITING BY AN AUTHORIZED ST REPRESENTATIVE, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER’S OWN RISK.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

© 2011 STMicroelectronics - All rights reserved

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com