I am doing a self-balancing triangle project, which is using reaction wheel to balance outer triangular frame on a pivot. The hardware I am using:

1. RS n° 2163792 hall sensor bldc motor
2. STEVAL-SPIN3202
3. MPU6050
4. STM32F767 Nucleo-144

The software I am using:

1. MotorControl Workbench 6.1.2
2. STM32 Cube IDE
3. STM32 CubeMX

I plan to input the information from MPU6050 to Nucleo-144 and use open source algorithm, such as kalman filter, to calculate the expect speed (and other information, if needed to drive the reaction wheel) of the reaction frame. Then pass the result to Steval-spin3202, which then alter the motor status.

What I want you to do:

1. Generate a piece of code using MPU6050 information to calculate expect speed the reaction wheel.
2. Pass the result to steval-spin3202

Some thing you need to know:

1. Using MotorControl Workbench 6.1.2 is capable of automatically generate motor driving code. Please modify the code base on that
2. Steval-spin3202 support I2C1, SPI1, UART communication, please tell me which one is the best solution, and what I should do with STM32 CubeMX.

Tell me if theres any further information needed, thank you.

I am using STM32F767 nucleo-144 and mpu6050 to build a control system of self-balancing triangular. Could you provide me a of code that could be run in stm43 Cube ide that set the target angle and calculate the expected speed (with direction) of the reaction wheel base on the position angle of the outer frame, the angular velocity of the outer frame and the shaft velocity of the motor. Here are some examples:  
1. pwm\_s = -constrain(X1 \* robot\_angle + X2 \* gyroZfilt + X3 \* -motor\_speed, -255, 255);

2. target\_velocity = controllerLQR(pendulum\_angle, gyroZrate, motor.shaft\_velocity);

float controllerLQR(float p\_angle, float p\_vel, float m\_vel)

{

if (abs(p\_angle) > 5) //摆角大于5则进入非稳态，记录非稳态时间

{

last\_unstable\_time = millis();

if (stable) //如果是稳态进入非稳态则调整为目标角度

{

//target\_angle = EEPROM.readFloat(0) - p\_angle;

target\_angle = EEPROM.readFloat(0);

stable = 0;

}

}

if ((millis() - last\_unstable\_time) > 1000 && !stable) //非稳态进入稳态超过500ms检测，更新目标角为目标角+摆角，假设进入稳态

{

//target\_angle -= \_sign(target\_velocity) \* 0.4;

target\_angle = target\_angle+p\_angle;

stable = 1;

}

if ((millis() - last\_stable\_time) > 2500 && stable) { //稳态超过2000ms检测，更新目标角

if (abs(target\_velocity) > 5 ) { //稳态速度偏大校正

last\_stable\_time = millis();

target\_angle -= \_sign(target\_velocity) \* 0.2;

}

}

//Serial.println(stable);

float u;

if (!stable) //非稳态计算

{

motor.PID\_velocity.P = v\_p\_1;

motor.PID\_velocity.I = v\_i\_1;

u = LQR\_K3\_1 \* p\_angle + LQR\_K3\_2 \* p\_vel + LQR\_K3\_3 \* m\_vel;

}

else

{

motor.PID\_velocity.P = v\_p\_2;

motor.PID\_velocity.I = v\_i\_2;

u = LQR\_K4\_1 \* p\_angle + LQR\_K4\_2 \* p\_vel + LQR\_K4\_3 \* m\_vel;

}

return u;

}

Please try to use the command in the following .h files to generate the code that: use the value receive from tx/rx communication to constantly set and reset the speed of the motor. (negative/positive means direction) and update with current speed to tx/rx communication

These command are from the automatically generated code of mc workbench, please refer to that if you need

\* @file usart\_aspep\_driver.c

\* @author Motor Control SDK Team, ST Microelectronics

\* @brief This file provides firmware functions that implement the uart driver for the aspep protocol

\*

**#include** <stdint.h>

**#include** "mc\_stm\_types.h"

**#include** "usart\_aspep\_driver.h"

**void** **UASPEP\_DAMCONFIG\_TX**(UASPEP\_Handle\_t \*pHandle);

**void** **UASPEP\_DAMCONFIG\_RX**(UASPEP\_Handle\_t \*pHandle);

**void** **UASPEP\_INIT**(**void** \*pHWHandle)

{ UASPEP\_Handle\_t \*pHandle = (UASPEP\_Handle\_t \*)pHWHandle; //cstat !MISRAC2012-Rule-11.5

UASPEP\_DAMCONFIG\_TX(pHandle);

UASPEP\_DAMCONFIG\_RX(pHandle);}

**void** **UASPEP\_DAMCONFIG\_TX**(UASPEP\_Handle\_t \*pHandle)

{**#ifdef** NULL\_PTR\_USA\_ASP\_DRV

**if** (NULL == pHandle) { /\* Nothing to do \*/ } **else** {**#endif**

/\* Enable DMA UART \*/ LL\_USART\_ClearFlag\_TC(pHandle->USARTx);

LL\_USART\_EnableIT\_TC(pHandle->USARTx);

/\* Enable DMA UART to start the TX request \*/

LL\_USART\_EnableDMAReq\_TX(pHandle->USARTx);

LL\_DMA\_SetPeriphAddress(pHandle->txDMA, pHandle->txChannel, (uint32\_t)&pHandle->USARTx->TDR);

/\* Clear UART ISR \*/ LL\_USART\_ClearFlag\_TC(pHandle->USARTx);

**#ifdef** NULL\_PTR\_USA\_ASP\_DRV }

**#endif }**

**void** **UASPEP\_DAMCONFIG\_RX**(UASPEP\_Handle\_t \*pHandle)

**#ifdef** NULL\_PTR\_USA\_ASP\_DRV

**if** (NULL == pHandle) {

/\* Nothing to do \*/ }

**else** {**#endif**

LL\_USART\_EnableIT\_ERROR(pHandle->USARTx);

LL\_DMA\_SetPeriphAddress(pHandle->rxDMA, pHandle->rxChannel, (uint32\_t)&pHandle->USARTx->RDR);

/\* Clear UART ISR \*/

LL\_USART\_ClearFlag\_TC(pHandle->USARTx);

LL\_USART\_EnableDMAReq\_RX(pHandle->USARTx);

**#ifdef** NULL\_PTR\_USA\_ASP\_DRV }

**#endif**}

bool **UASPEP\_SEND\_PACKET**(**void** \*pHWHandle, **void** \*data, uint16\_t length)

{ UASPEP\_Handle\_t \*pHandle = (UASPEP\_Handle\_t \*)pHWHandle; //cstat !MISRAC2012-Rule-11.5

bool result;

**if** (0U == LL\_DMA\_IsEnabledChannel(pHandle->txDMA, pHandle->txChannel))

{ //cstat !MISRAC2012-Rule-11.4 !MISRAC2012-Rule-11.6

LL\_DMA\_SetMemoryAddress(pHandle->txDMA, pHandle->txChannel, (uint32\_t)data);

LL\_DMA\_SetDataLength(pHandle->txDMA, pHandle->txChannel, length);

LL\_DMA\_EnableChannel(pHandle->txDMA, pHandle->txChannel);

result = true; }

**else** { result = false; }

**return** result;}

**void** **UASPEP\_RECEIVE\_BUFFER**(**void** \*pHWHandle, **void**\* buffer, uint16\_t length)

{ UASPEP\_Handle\_t \*pHandle = (UASPEP\_Handle\_t \*)pHWHandle; //cstat !MISRAC2012-Rule-11.5

LL\_DMA\_DisableChannel(pHandle->rxDMA, pHandle->rxChannel);

//cstat !MISRAC2012-Rule-11.4 !MISRAC2012-Rule-11.6

LL\_DMA\_SetMemoryAddress(pHandle->rxDMA, pHandle->rxChannel, (uint32\_t)buffer);

LL\_DMA\_SetDataLength(pHandle->rxDMA, pHandle->rxChannel, length);

LL\_DMA\_EnableChannel(pHandle->rxDMA, pHandle->rxChannel);}

**void** **UASPEP\_IDLE\_ENABLE**(**void** \*pHWHandle)

{ UASPEP\_Handle\_t \*pHandle = (UASPEP\_Handle\_t \*)pHWHandle; //cstat !MISRAC2012-Rule-11.5

LL\_USART\_ClearFlag\_IDLE(pHandle->USARTx);

LL\_USART\_EnableIT\_IDLE(pHandle->USARTx);}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @file mc\_api.c

\* @author Motor Control SDK Team, ST Microelectronics

\* @brief This file implements the high level interface of the Motor Control SDK.

\_\_weak bool **MC\_StartMotor1**(**void**)

{ **return** **MCI\_StartMotor**( pMCI[M1] );}

\_\_weak bool **MC\_StopMotor1**(**void**)

{ **return** **MCI\_StopMotor**( pMCI[M1] );}

\_\_weak **void** **MC\_ProgramSpeedRampMotor1**( int16\_t hFinalSpeed, uint16\_t hDurationms )

{MC**I\_ExecSpeedRamp**( pMCI[M1], hFinalSpeed, hDurationms );}

\_\_weak **void** **MC\_ProgramSpeedRampMotor1\_F**( **float** FinalSpeed, uint16\_t hDurationms )

{ **MCI\_ExecSpeedRamp\_F**( pMCI[M1], FinalSpeed, hDurationms );}

\_\_weak **void** **MC\_ProgramTorqueRampMotor1**( int16\_t hFinalTorque, uint16\_t hDurationms )

{ **MCI\_ExecTorqueRamp**( pMCI[M1], hFinalTorque, hDurationms );}

\_\_weak **void** **MC\_ProgramTorqueRampMotor1\_F**( **float** FinalTorque, uint16\_t hDurationms ){

**MCI\_ExecTorqueRamp\_F**( pMCI[M1], FinalTorque, hDurationms );}

\_\_weak **void** **MC\_SetCurrentReferenceMotor1**( qd\_t Iqdref ){

**MCI\_SetCurrentReferences**( pMCI[M1], Iqdref );}}

\_\_weak MCI\_CommandState\_t **MC\_GetCommandStateMotor1**( **void**){

**return** **MCI\_IsCommandAcknowledged**( pMCI[M1] );}

\_\_weak bool **MC\_StopSpeedRampMotor1**(**void**){

**return** **MCI\_StopSpeedRamp**( pMCI[M1] );}

\_\_weak **void** **MC\_StopRampMotor1**(**void**){

**MCI\_StopRamp**( pMCI[M1] );}

\_\_weak bool **MC\_HasRampCompletedMotor1**(**void**)

{ **return** **MCI\_RampCompleted**( pMCI[M1] );}

\_\_weak int16\_t **MC\_GetMecSpeedReferenceMotor1**(**void**)

{ **return** **MCI\_GetMecSpeedRefUnit**( pMCI[M1] );}

\_\_weak int16\_t **MC\_GetMecSpeedAverageMotor1**(**void**)

{ **return** **MCI\_GetAvrgMecSpeedUnit**( pMCI[M1] );}

\_\_weak int16\_t **MC\_GetLastRampFinalSpeedMotor1**(**void**)

{ **return** **MCI\_GetLastRampFinalSpeed**( pMCI[M1] );}

\_\_weak MC\_ControlMode\_t **MC\_GetControlModeMotor1**(**void**)

{ **return** **MCI\_GetControlMode**( pMCI[M1] );}

\_\_weak int16\_t **MC\_GetImposedDirectionMotor1**(**void**)

{ **return** **MCI\_GetImposedMotorDirection**( pMCI[M1] );}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @file mc\_interface.c

\* @author Motor Control SDK Team, ST Microelectronics

**#define** round(x) ((x)>=0?(int32\_t)((x)+0.5):(int32\_t)((x)-0.5))

\_\_weak **void** **MCI\_Init**(MCI\_Handle\_t \*pHandle, SpeednTorqCtrl\_Handle\_t \*pSTC, pFOCVars\_t pFOCVars, PWMC\_Handle\_t \*pPWMHandle )

{**#ifdef** NULL\_PTR\_MC\_INT

**if** (MC\_NULL == pHandle) { /\* Nothing to do \*/ }

**else** {**#endif**

pHandle->pSTC = pSTC;

pHandle->pFOCVars = pFOCVars;

pHandle->pPWM = pPWMHandle;

/\* Buffer related initialization \*/

pHandle->lastCommand = *MCI\_NOCOMMANDSYET*;

pHandle->hFinalSpeed = 0;

pHandle->hFinalTorque = 0;

pHandle->hDurationms = 0;

pHandle->CommandState = *MCI\_BUFFER\_EMPTY*;

pHandle->DirectCommand = *MCI\_NO\_COMMAND*;

pHandle->State = *IDLE*;

pHandle->CurrentFaults = MC\_NO\_FAULTS;

pHandle->PastFaults = MC\_NO\_FAULTS;

**#ifdef** NULL\_PTR\_MC\_INT }

**#endif**}

\_\_weak **void** **MCI\_ExecSpeedRamp**(MCI\_Handle\_t \*pHandle, int16\_t hFinalSpeed, uint16\_t hDurationms)

{**#ifdef** NULL\_PTR\_MC\_INT

**if** (MC\_NULL == pHandle) { /\* Nothing to do \*/ }

**else** {**#endif**

pHandle->lastCommand = *MCI\_CMD\_EXECSPEEDRAMP*;

pHandle->hFinalSpeed = hFinalSpeed;

pHandle->hDurationms = hDurationms;

pHandle->CommandState = *MCI\_COMMAND\_NOT\_ALREADY\_EXECUTED*;

pHandle->LastModalitySetByUser = *MCM\_SPEED\_MODE*;

**#ifdef** NULL\_PTR\_MC\_INT }

**#endif**}

\_\_weak **void** **MCI\_ExecSpeedRamp\_F**( MCI\_Handle\_t \* pHandle, **const** **float** FinalSpeed, uint16\_t hDurationms )

{**#ifdef** NULL\_PTR\_MC\_INT

**if** (MC\_NULL == pHandle)

{ /\* Nothing to do \*/ }

**else** {**#endif**

int16\_t hFinalSpeed = (int16\_t) ((FinalSpeed \* SPEED\_UNIT) / U\_RPM);

MCI\_ExecSpeedRamp(pHandle, hFinalSpeed, hDurationms);

**#ifdef** NULL\_PTR\_MC\_INT }

**#endif**}

\_\_weak **void** **MCI\_SetCurrentReferences**(MCI\_Handle\_t \*pHandle, qd\_t Iqdref)

{**#ifdef** NULL\_PTR\_MC\_INT

**if** (MC\_NULL == pHandle)

{ /\* Nothing to do \*/ }

**else** {**#endif**

pHandle->lastCommand = *MCI\_CMD\_SETCURRENTREFERENCES*;

pHandle->Iqdref.q = Iqdref.q;

pHandle->Iqdref.d = Iqdref.d;

pHandle->CommandState = *MCI\_COMMAND\_NOT\_ALREADY\_EXECUTED*;

pHandle->LastModalitySetByUser = *MCM\_TORQUE\_MODE*;

**#ifdef** NULL\_PTR\_MC\_INT }

**#endif**}

\_\_weak **void** **MCI\_SetSpeedMode**( MCI\_Handle\_t \* pHandle )

{ pHandle->pFOCVars->bDriveInput = *INTERNAL*;

**STC\_SetControlMode**( pHandle->pSTC, *MCM\_SPEED\_MODE* );

pHandle->LastModalitySetByUser = *MCM\_SPEED\_MODE*;}

\_\_weak bool **MCI\_StartMotor**(MCI\_Handle\_t \*pHandle)

{ bool RetVal; ..omit code body…

**return** (RetVal);}

\_\_weak bool **MCI\_StartWithMeasurementOffset**(MCI\_Handle\_t\* pHandle)

{ bool RetVal;

**if** ((*IDLE* == MCI\_GetSTMState(pHandle)) &&

(MC\_NO\_FAULTS == MCI\_GetOccurredFaults(pHandle)) &&

(MC\_NO\_FAULTS == MCI\_GetCurrentFaults(pHandle))) {

pHandle->DirectCommand = *MCI\_START*;

pHandle->CommandState = *MCI\_COMMAND\_NOT\_ALREADY\_EXECUTED*;

pHandle->pPWM->offsetCalibStatus = false;

RetVal = true; }

**else** { /\* reject the command as the condition are not met \*/

RetVal = false; }

**return** (RetVal);}

\_\_weak bool **MCI\_StartOffsetMeasurments**(MCI\_Handle\_t \*pHandle)

{ bool RetVal;

**if** ((*IDLE* == MCI\_GetSTMState(pHandle)) &&

(MC\_NO\_FAULTS == MCI\_GetOccurredFaults(pHandle)) &&

(MC\_NO\_FAULTS == MCI\_GetCurrentFaults(pHandle)))

{

pHandle->DirectCommand = *MCI\_MEASURE\_OFFSETS*;

pHandle->pPWM->offsetCalibStatus = false;

RetVal = true; }

**else** { /\* reject the command as the condition are not met \*/ RetVal = false; } **return** (RetVal);}

\_\_weak bool **MCI\_GetCalibratedOffsetsMotor**(MCI\_Handle\_t\* pHandle, PolarizationOffsets\_t \* PolarizationOffsets)

{ bool RetVal;

**if** ( pHandle->pPWM->offsetCalibStatus == true ) {

**PWMC\_GetOffsetCalib**(pHandle->pPWM, PolarizationOffsets);

RetVal = true; }

**else** { RetVal = false; }

**return**(RetVal);}

\_\_weak bool **MCI\_SetCalibratedOffsetsMotor**( MCI\_Handle\_t\* pHandle, PolarizationOffsets\_t \* PolarizationOffsets){ bool RetVal;

**if** ((*IDLE* == MCI\_GetSTMState(pHandle)) &&

(MC\_NO\_FAULTS == MCI\_GetOccurredFaults(pHandle)) &&

(MC\_NO\_FAULTS == MCI\_GetCurrentFaults(pHandle)))

{ **PWMC\_SetOffsetCalib**(pHandle->pPWM, PolarizationOffsets);

pHandle->pPWM->offsetCalibStatus = true;

RetVal = true; }

**return**(RetVal);}

\_\_weak bool **MCI\_StopMotor**(MCI\_Handle\_t \* pHandle)

{ bool RetVal;

bool status;

MCI\_State\_t State;

State = MCI\_GetSTMState(pHandle);

**if** (*IDLE* == State || *ICLWAIT* == State)

{ status = false; }

**else** { status = true; }

**if** ((MC\_NO\_FAULTS == MCI\_GetOccurredFaults(pHandle)) &&

(MC\_NO\_FAULTS == MCI\_GetCurrentFaults(pHandle)) &&

status == true )

{ pHandle->DirectCommand = *MCI\_STOP*;

RetVal = true; }

**else** { /\* reject the command as the condition are not met \*/

RetVal = false; }

**return** (RetVal);}\

\_weak **void** **MCI\_ExecBufferedCommands**(MCI\_Handle\_t \*pHandle)

{**#endif**}

\_\_weak int16\_t **MCI\_GetImposedMotorDirection**(MCI\_Handle\_t \*pHandle)

{

int16\_t retVal = 1;

**#ifdef** NULL\_PTR\_MC\_INT

**if** (MC\_NULL == pHandle) { /\* Nothing to do \*/ }

**else** {**#endif**

**switch** (pHandle->lastCommand) { **case** *MCI\_CMD\_EXECSPEEDRAMP*:

**if** (pHandle->hFinalSpeed < 0) {

retVal = -1; } **break**; **case** *MCI\_CMD\_EXECTORQUERAMP*: **if** (pHandle->hFinalTorque < 0)

{ retVal = -1; } **break**;

**case** *MCI\_CMD\_SETCURRENTREFERENCES*:

**if** (pHandle->Iqdref.q < 0)

{ retVal = -1; }

**break**; **default**: **break**; }

**#ifdef** NULL\_PTR\_MC\_INT }

**#endif**

**return** (retVal);}

\_\_weak int16\_t **MCI\_GetLastRampFinalSpeed**(MCI\_Handle\_t \*pHandle)

{

**#ifdef** NULL\_PTR\_MC\_INT

int16\_t retVal = 0;

**if** (MC\_NULL == pHandle) { /\* Nothing to do \*/ }

**else** { retVal = pHandle->hFinalSpeed; }

**return** (retVal);

**#else** **return** (pHandle->hFinalSpeed);

**#endif**}

\_\_weak int16\_t **MCI\_GetLastRampFinalTorque**(MCI\_Handle\_t \*pHandle){

**#ifdef** NULL\_PTR\_MC\_INT

int16\_t retVal = 0;

**if** (MC\_NULL == pHandle) { /\* Nothing to do \*/ }

**else** { retVal = pHandle->hFinalTorque; }

**return** (retVal);

**#else** **return** (pHandle->hFinalTorque);

**#endif**}

\_\_weak uint16\_t **MCI\_GetLastRampFinalDuration**(MCI\_Handle\_t \*pHandle)

{**#ifdef** NULL\_PTR\_MC\_INT

uint16\_t retVal = 0;

**if** (MC\_NULL == pHandle) { /\* Nothing to do \*/ }

**else** { retVal = pHandle->hDurationms; }

**return** (retVal);

**#else** **return** (pHandle->hDurationms);

**#endif**}

\_\_weak **float** **MCI\_GetLastRampFinalSpeed\_F**(MCI\_Handle\_t \*pHandle)

{

**float** RetVal = 0.0;

**if** (MC\_NULL == pHandle)

{ /\* Nothing to do \*/ }

**else**

{ RetVal = (**float**)((pHandle->hFinalSpeed \* U\_RPM) / SPEED\_UNIT); }

**return** (RetVal);}

\_\_weak bool **MCI\_RampCompleted**(MCI\_Handle\_t \*pHandle)

{ bool retVal = false;

**#ifdef** NULL\_PTR\_MC\_INT

**if** (MC\_NULL == pHandle)

{ /\* Nothing to do \*/ }

**else** {

**#endif**

**if** (*RUN* == MCI\_GetSTMState(pHandle)) {

retVal = **STC\_RampCompleted**(pHandle->pSTC); }

**else** { /\* Nothing to do \*/ }

**#ifdef** NULL\_PTR\_MC\_INT }

**#endif**

**return** (retVal);}

\_\_weak bool **MCI\_StopSpeedRamp**(MCI\_Handle\_t \*pHandle)

{**#ifdef** NULL\_PTR\_MC\_INT

**return** ((MC\_NULL == pHandle) ? false : STC\_StopSpeedRamp(pHandle->pSTC));

**#else** **return** (**STC\_StopSpeedRamp**(pHandle->pSTC));

**#endif**}

\_\_weak **void** **MCI\_StopRamp**(MCI\_Handle\_t \*pHandle)

{**#ifdef** NULL\_PTR\_MC\_INT

**if** (MC\_NULL == pHandle) { /\* Nothing to do \*/ }

**else** {**#endif**

**STC\_StopRamp**(pHandle->pSTC);

**#ifdef** NULL\_PTR\_MC\_INT }

**#endif**}

\_\_weak bool **MCI\_GetSpdSensorReliability**(MCI\_Handle\_t \*pHandle)

\_\_weak int16\_t **MCI\_GetAvrgMecSpeedUnit**(MCI\_Handle\_t \*pHandle)

{ int16\_t temp\_speed;

**#ifdef** NULL\_PTR\_MC\_INT

**if** (MC\_NULL == pHandle)

{ temp\_speed = 0; }

**else** {**#endif**

SpeednPosFdbk\_Handle\_t \* SpeedSensor = **STC\_GetSpeedSensor**(pHandle->pSTC);

temp\_speed = **SPD\_GetAvrgMecSpeedUnit**(SpeedSensor);

**#ifdef** NULL\_PTR\_MC\_INT }

**#endif** **return** (temp\_speed);}

\_\_weak **float** **MCI\_GetAvrgMecSpeed\_F**(MCI\_Handle\_t \*pHandle)

{ SpeednPosFdbk\_Handle\_t \*SpeedSensor = **STC\_GetSpeedSensor**(pHandle->pSTC);

**return** ((**float**)((**SPD\_GetAvrgMecSpeedUnit**(SpeedSensor) \* U\_RPM) / SPEED\_UNIT));}

\_\_weak int16\_t **MCI\_GetMecSpeedRefUnit**(MCI\_Handle\_t \*pHandle)

{**#ifdef** NULL\_PTR\_MC\_INT

**return** ((MC\_NULL == pHandle) ? 0 : STC\_GetMecSpeedRefUnit(pHandle->pSTC));

**#else** **return** (**STC\_GetMecSpeedRefUnit**(pHandle->pSTC));

**#endif**}

\_\_weak **void** **MX\_MotorControl\_Init**(**void**)

(**void**)**HAL\_SYSTICK\_Config**(**HAL\_RCC\_GetHCLKFreq**() / SYS\_TICK\_FREQUENCY);

**HAL\_NVIC\_SetPriority**(*SysTick\_IRQn*, uwTickPrio, 0U);

**MCboot**(pMCI);

**mc\_lock\_pins**();}