V 0.2

Tasks to do :

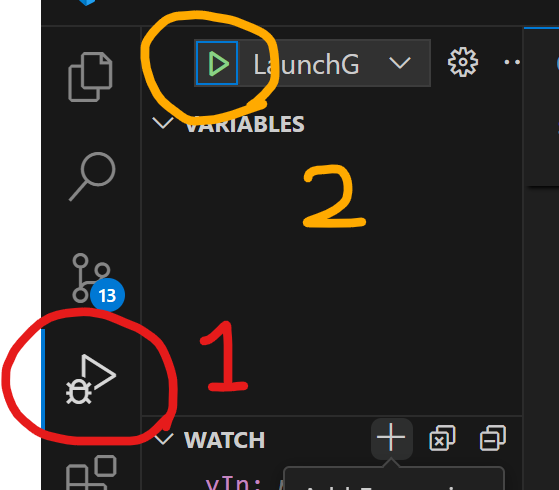
1. Study code loading without zephyr env if possible / If not possible propose a solution .

It is possible to upload to code to the microcontroller using CubeProgrammar.

1. **Write a quick note to describe the process and the different steps to load code in the mcu for operator (with or without zephyr env)**

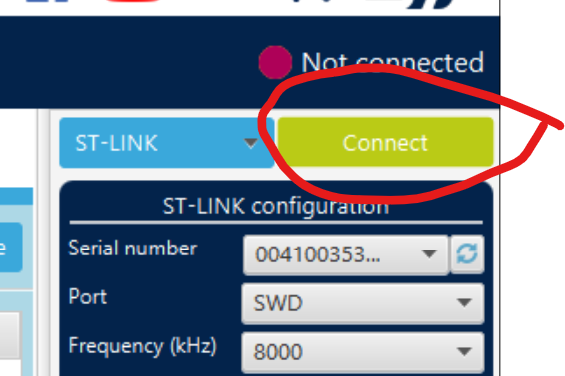
* **With zephyr env:**

**First select 1, next select 2. Everything will be done automatically.**

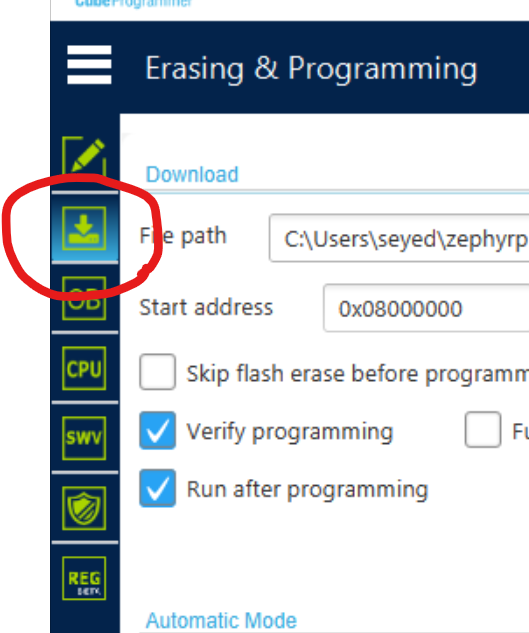
****

* **Without zephyr env:**

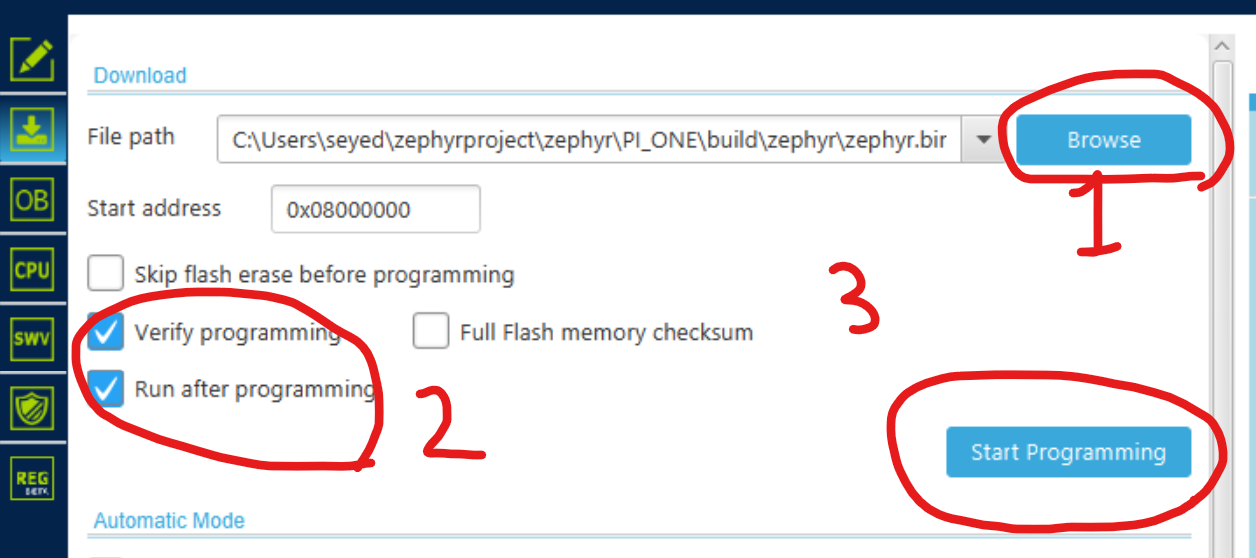
1. **Cube Programmer must be installed.**
2. **Connect the debugger to the PC.**
3. **Click on connect:**

****

1. **Click here**

****

1. **Navigate to the binary file, mark these checks and click on start programming.**

****

1. **PWM IP**

Open loop code (A simple code to generate PWM on HRTIM1\_CHA1 and HRTIM1\_CHA2)

The PWM are at a frequency of 100kHz

A deadtime of 150ns is implemented between both PWM

HRTIM1\_CHA2 is the complementary of HRTIM1\_CHA1

In this code the user can modify easily the duty cycle of the PWM

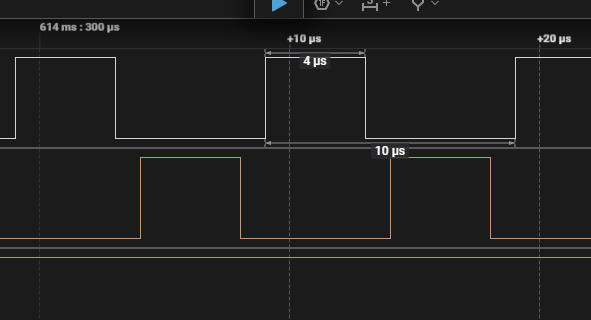
**Write a simple descriptive note of the code architecture , the code description and how to use it .**

**In the code there are two different timer types**

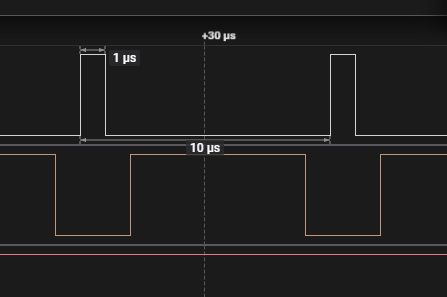
* **High Resolution Timer: This code is completely generated by CubeMX and imported into Zephyr Codes.**
* **Other Timers. These timers work using Zephyr APIs.**

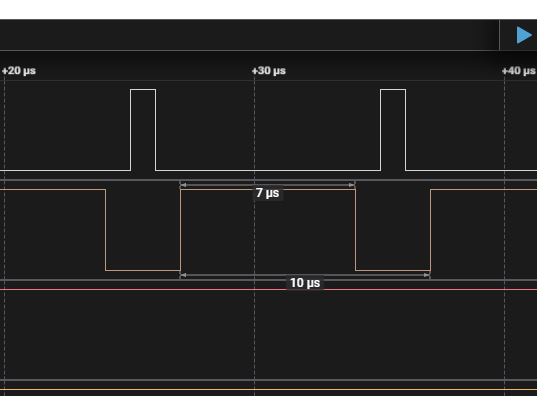
**In the code we have an initial function named initPwms inside a function called initBoard. In the initPwms function, the high-resolution timer frequency is set to 100KHz, and its duty cycle is configurable. Its default value is set to 0.5. The dead time is also about 1uS. There is another function called pwmSet. pwmSet takes index, frequency, and duty cycle as its input. For the high-resolution timer, the frequency is fixed to 100KHz, and only its duty cycle is configurable. The duty cycle is containing dead time value. For example, in the following figures, the duty cycle for the primary channel is 0.5. As the period is 10uS for 100KHz, the primary channel should stay 5uS high, but the dead time will be subtracted from 5uS. So, 5 – 1 = 4uS stays high. The secondary channel is complementary of the primary channel.**

****

****

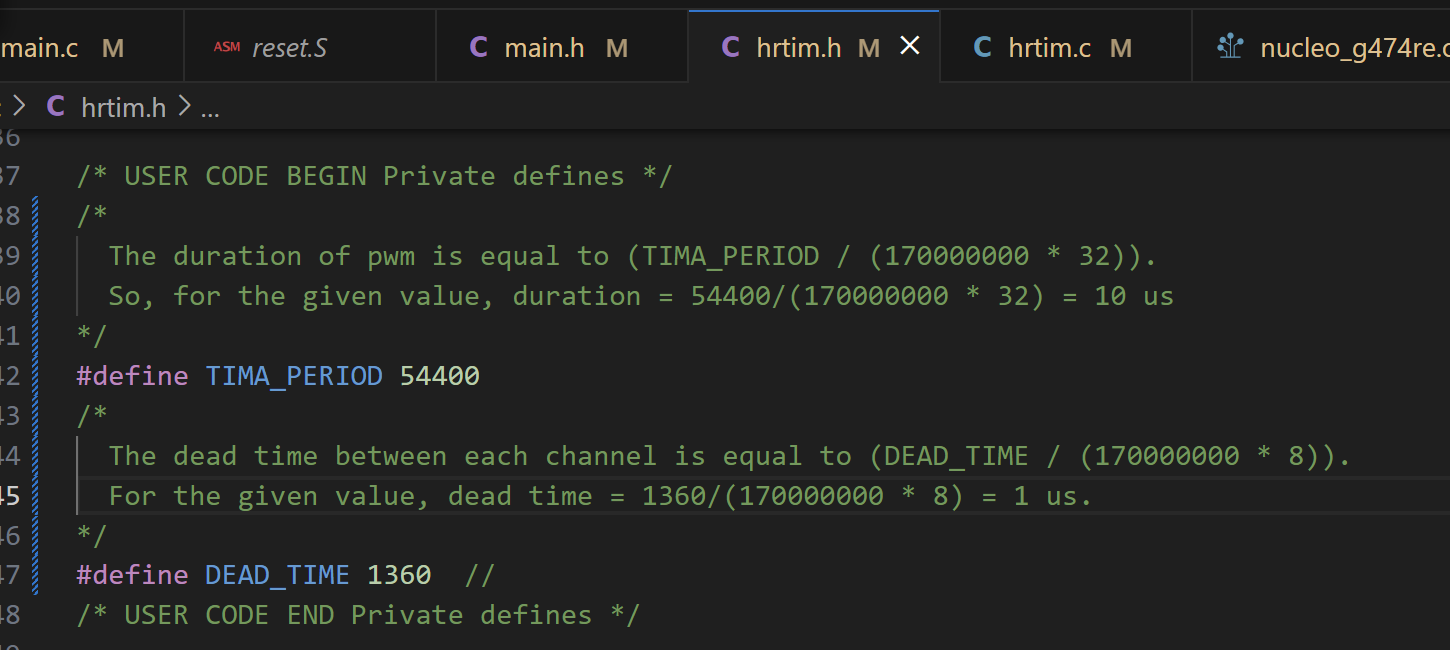
**As another example, in the following figures, the duty cycle is set to 0.2. That means, the primary channel should stay high for 2uS and the secondary channel should stay high for 8uS. There is 1uS dead time between the two channels. So, the primary channel stay 1uS high and the secondary channel stay 7uS high.**

****

****

**How to use the code:**

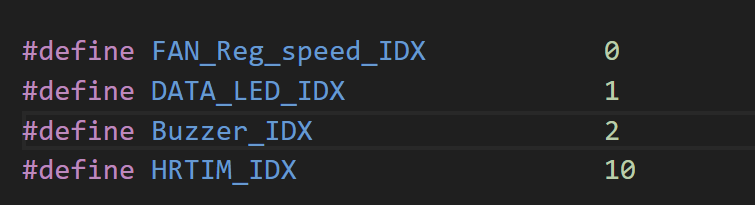
**To change the duty cycle and the dead time for high resolution timer, change the values in hrtim.h file.**

****

**To change the duty cycle and the frequency for other timers, use the pwmSet function. Also note that the duty cycle for the high-resolution timer if configurable using pwmSet function. Here idx is the index channel of the timer, frequency is the desire frequency, and the duty cycle is the desire duty cycle.**

pwmSet(uint8\_t idx, uint32\_t frequency, float dutycycle)

**The valid indices are:**

****

**To turn the all pwms off, call function**

turnOffAllPWMs()

1. **ADC IP 1**

Propose a simple code to measure the Input voltage of the board V\_in\_mcu PA0 and to display it .

The acquisition chain is composed of :

Resistor divider with gain : R203/R203+R202

Op amp circuit of gain : Vref + (12k /27k) \* ( V in output of resitor divider)

Make sure that you set Vref of the mcu at 2.048

According to my measurement, Vref changes between 1.99V and 2.00V. ST says Vref is around 2.048V not exactly 2.048.

**Write a simple descriptive note of the code architecture , the code description and how to use it .**

1. **ADC IP 2**

Propose a code that combine the code 3 with the code 4 .

The goal here is to sample and display the following values :

V\_in\_mcu (PA0)

V\_out\_mcu (PA1)

I\_in\_mcu (PC2)

I\_out\_mcu (PC3)

Temp (PB15)

Propose a simple code to sample theses values and display them with update every 3s while generating the PWM as described in 3.

The acquisition chain of voltage measurement is composed of :

Resistor divider with gain : R203/R203+R202

Op amp circuit of gain : Vref + (12k /27k) \* ( V in output of resitor divider)

Make sure that you set Vref of the mcu at 2.048

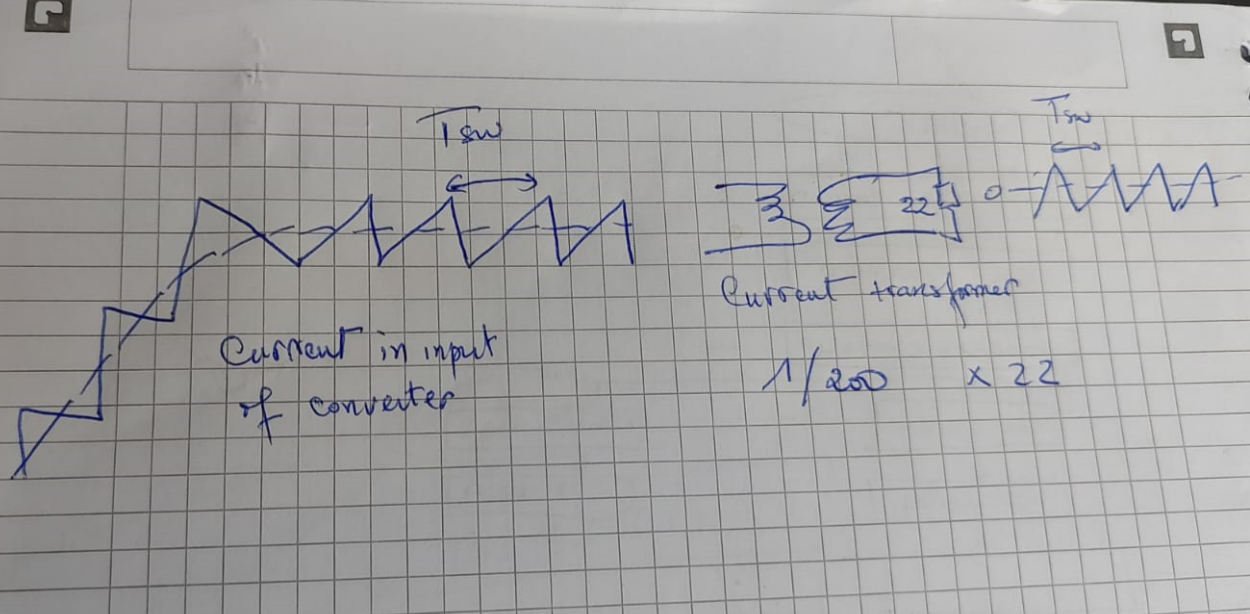
The acquisition chain of current measurement is composed of :

Current transformer : Gain = 1/200

Op amp circuit of gain : Vref + (12k /27k) \* ( V in output of current transformer)

Make sure that you set Vref of the mcu at 2.048

The current transformer is a very particular sensor it take in input the switched current and transform it according to it’s gain while also passing the average of the signal at 0. (eliminate the DC part of the signal )



The output current of the transformer is then the input current divide by 200 centered at 0.

That current pass in the resistor of 22O to give the signal in input of mcu .

The gain are :

Transformer gain : 1/200

Resistor gain : 22\* current in output of transformer

The current in the converter vary from 0 to 30Amp

What is interesting for us here is the peak current .

To be able to sample this peak current the current sampling need to be done with the same clock generating the PWM and at the same frequency than PWM frequency.

That peak current need to be measure accurately and display every 3 s

**Write a simple descriptive note of the code architecture , the code description and how to use it .**

6 – **Output voltage control**

By combining parts of the different code proposed before

Propose a code that :

Measure :

V\_in\_mcu (PA0)

V\_out\_mcu (PA1)

I\_in\_mcu (PC2)

I\_out\_mcu (PC3)

At the same frequency as PWM (100kHz) with the same clock as pwm clock so that the acquisition is phased with the peak of the current ripple.

A variable modifyable in real time while code is running is used to define the wanted voltage in output :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Float Output\_voltage\_ref

Output\_voltage\_ref is compared with the measured output voltage :

Error = Output\_voltage\_ref - V\_out\_mcu(adapted)

The error is fed to a PI controller that calculate the dutycycle of PWM and update it

All this process is executed at the same frequency as PWM (100kHz) with the same clock as pwm clock so that the acquisition and the duty cycle modification is phased with the peak of the current ripple.

**The PI controller function :**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**PI\_controller.H**

#ifndef CONTROLLERS\_H\_

#define CONTROLLERS\_H\_

#include <stdint.h>

/\*\*

\* Library written by ALEK GUEDEGBE for Workflow PSIM to IMPERIX

\* PI controller library

\* Definition and operation according to the Workflow PSIM to IMPERIX

\*/

typedef struct{

float kp,ti; // Gain and time constant of PI

float limup; // Upper saturation value of the output

float limlow; // Lower saturation value of the output

float ui\_prev; // Previous value of the integral component

float e\_prev; // Previous value of the error

float Fsamp; // sampling frequency of the PI controller

float A1 ;

} PIController;

/\*\*

\* Routine to configure the PI controller 'me' and pre-compute the necessary constants.

\* @param \*me the PID pseudo-object to be configured

\* @param kp proportional gain

\* @param ti integral time constant

\* @param limup upper saturation threshold of the output quantity

\* @param limlow lower saturation threshold of the output quantity

\* @param Fsamp sampling (interrupt) frequency

\* @return void

\*/

void ConfigPIController(PIController\* me, float kp, float ti, float limup, float limlow, float Fsamp);

/\*\*

\* Routines to run the pseudo-object 'me' depending of its actual nature (PI, PID, etc. controllers)

\* @param \*me the corresponding PID pseudo-object (parameters and state quantities)

\* @param error the setpoint value minus the measured value

\* @return the control variable for the measured quantity (output of the controller)

\*/

float RunPIController(PIController\* me, float error);

#endif /\*CONTROLLERS\_H\_\*/

**PI\_controller.Cpp**

/\*

\* @title Discrete closed-loop controllers

\* @author ALEK GUEDEGBE (LAEH CEA GRENOBLE)

\* @file controllers.cpp <--> WPI=Workflow PSIM to IMPERIX

\*/

#include "controllers.h" // Corresponding header file

#include <cmath> // Standard math library

#include "Core/core.h"

/\*

\* Routine to configure the PI controller and pre-compute the necessary constants.

\*/

void ConfigPIController(PIController\* me, float kp, float ti, float limup, float limlow, float Fsamp)

{

// Set the controller parameters:

me->kp = kp;

me->ti = ti;

me->limup = limup;

me->limlow = limlow;

me->Fsamp = Fsamp;

me->A1 = kp / (ti\*Fsamp);

// Initialize the state quantities:

me->e\_prev = 0.0;

me->ui\_prev = 0.0;

}

/\*

\* Routine to run the PI controller 'me' .

\*/

float RunPIController(PIController\* me, float error)

{

float up;

float ui; // Integral part of the output

float u;

up= me->kp \* error;

ui = ( me->A1 \* error) + (me->ui\_prev) ;

u = up + ui ;

if ( (u>= me->limup) || (u <= me->limlow) )

{

ui = me->ui\_prev;

}

else

{

ui = ( me->A1 \* error) + (me->ui\_prev) ;

}

//Stores sample [K] in variables [K-1] in order to be used in the next sampling instant

me->e\_prev = error;

me->ui\_prev = ui;

//Limiter at the output of the compensator

if (u>= me->limup) {

u = me->limup;

}

if (u <= me->limlow) {

u = me->limlow;

}

// Reset the integral when the outputs are inhibited (when the converter is blocked):

if(GetCoreState() != OPERATING)

me->ui\_prev = 0.0; // Avoid integrating when core has been disabled

return u;

}

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Exemple of complete code with pi controller**

// Current command

//Current ref in Amp

//float IL\_cons= 12;

//IL\_ref = IL\_cons \* (1/20) \* 1 \* (32768/10.0);

float IL\_ref = 1966.08;

float IL\_mes=0;

float Duty\_cycle=0;

// Paramètres correcteur PI\_I

float Kp\_i=191.781e-6;

float Ti\_i=364.628e-6;

float Up\_limit\_i=1;

float Low\_limit\_i=0;

float F\_samp=2\*SW\_FREQ;

PIController PI\_courant;

/\*\*

\*Initialisation routin

\* will be executed only one time in the beginning of run , used for initialisation and configuration of the IPs

\*/

tUserSafe UserInit(void)

{

/\*\*

\* Config of the main interrupt (main interrupt):

\* - Config of the clock 0 at the desired frequency

\*/

Clock\_SetFrequency(CLOCK\_0,F\_samp);

Clock\_SetFrequency(CLOCK\_1,SW\_FREQ);

ConfigureMainInterrupt(UserInterrupt,CLOCK\_0,0);

/\*\*

\* Sample in peak and valley of pwm triangle (switching current)

\*/

/\*\*

\* Configuration ADC

\*/

Adc\_ConfigureInput(ADC1,IL\_GAIN,0.0);

// no gain on ADC l'ADC

/\*\*

\* Configuration de la PWM

\* - mapping sur CLOCK\_0

\* - triangle carrier

\* - 1 microsecond dead-time between complementary signals

\* -

\*/

CbPwm\_ConfigureChannel(PWM\_CHANNEL\_0\_H,CLOCK\_1,TRIANGLE,1e-6);

// Configuration PI controller

//ConfigPIController(PIController\* me, float kp, float ti, float limup, float limlow, float Fsamp);

ConfigPIController(&PI\_courant,Kp\_i,Ti\_i,Up\_limit\_i,Low\_limit\_i,F\_samp);

return SAFE;

}

// Main : Interrupt main

tUserSafe UserInterrupt(void)

{

IL\_mes=Adc\_GetValue(ADC1);

// samplng in peak and valley of pwm (carrier based PWM)

Duty\_cycle=RunPIController(&PI\_courant,IL\_ref-IL\_mes);

// Update the PWM duty-cycles

CbPwm\_SetDutyCycle(PWM\_CHANNEL\_0\_H,Duty\_cycle);

return SAFE;

}

// Routine executed when the core state goes into FAULT mode

void UserError(tErrorSource source)

{

}