Tutorial n°2

GRID-TIED SINGLE-PHASE INVERTER

Written by: imperix SA, Rte. de l'Industrie 17, 1950 Sion, Switzerland
Nicolas Cherix <nicolas.cherix@imperix.ch>

Addressed topics:
– Use of the code libraries
– Implementation of a basic state machine

1 INTRODUCTION

This tutorial describes the procedure to control a grid-tied single-phase inverter using the BoomBox control platform. The considered system is depicted in Figure 1. Its main electrical parameters are indicated in Table 1:

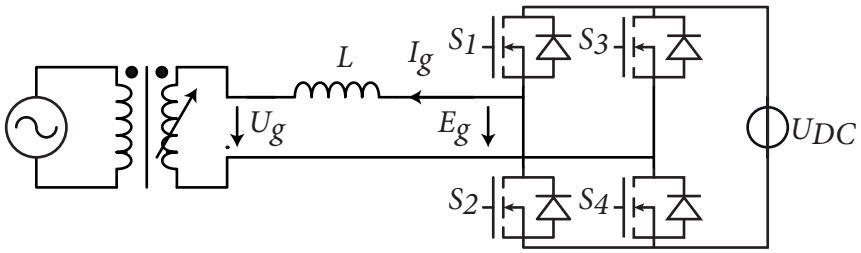

Figure 1 : Simplified electrical scheme of the considered system.

Name	Nom. value	Specification	Employed sensor	Channel #
U_g	50 V_{RMS}	Grid voltage at the injection point	LEM LV25-P	1
I_g	20 A_{RMS}	Current injected in the grid	LAH50-P	0
U_{DC}	100 V_{DC}	DC bus voltage (fixed)	LEM LV25-P	3
f_S	20 kHz	Switching frequency	N.A.	N.A.
L	0.5 mH	Smoothing inductor	N.A.	N.A.
f	50Hz	Grid frequency	N.A.	N.A.

Table 1 : Electrical parameters of the studied system.

The approach presented in this document – and detailed in the code example – consists in controlling the current exchanged between the inverter and an AC grid using a rotating reference frame (dq-type) synchronized with the grid frequency. The attractiveness of this approach is related to its excellent capability to decouple the control of the active and reactive power flows. Additionally, the mechanisms commonly used in three-phase applications can be re-used directly. On the other hand, this approach requires the emulate the components that are missing compared to a three-phase system, hence justifying the use of a dedicated emulation principle name *fictive-axis emulation* [2]. The overall strategy is depicted in Figure 2.

Furthermore, apart from the control itself, the coordinate transformations needed by this approach require a precise information on the phase angle of the grid voltage, what motivates the use of a phase-locked loop (PLL), also visible in Figure 2.

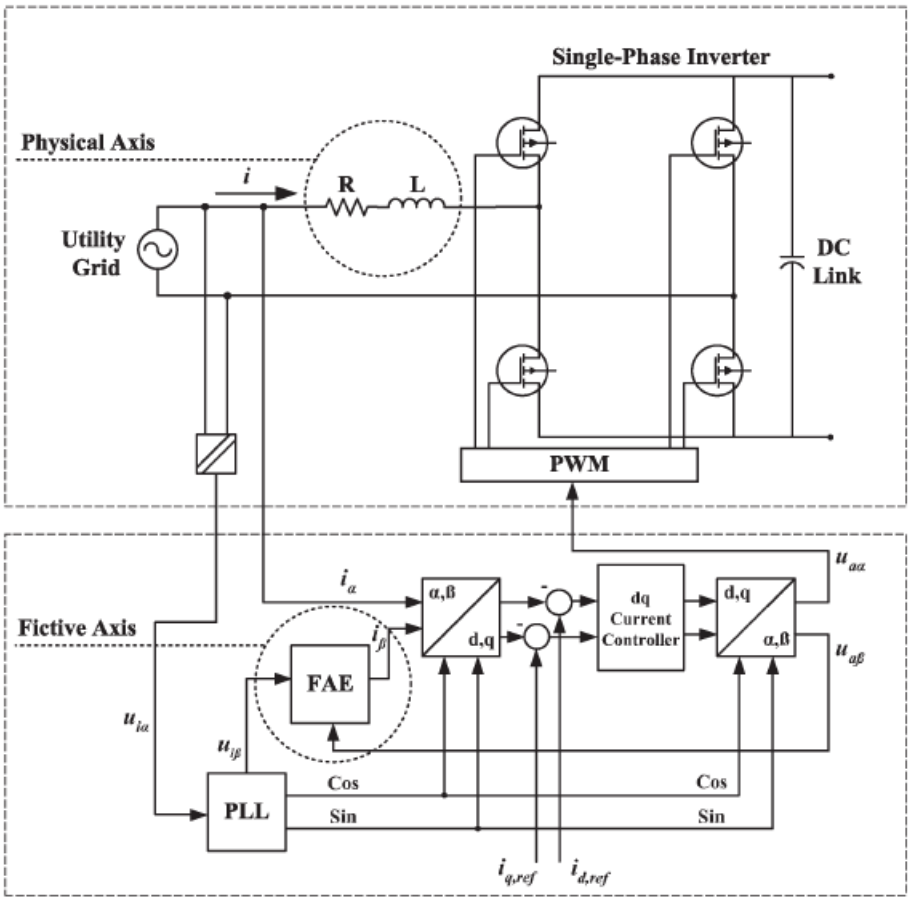

Figure 2 : Complete control structure with *fictive-axis emulation* and dq-type current control. Source [2].

Alternative approaches are obviously possible, such as relying on a Proportional-Resonant (PR) controller placed in a stationary reference frame. This second possibility is widely described in [5]. Section 4.a addressing the implementation of the state machine will detail how other control strategies can easily be integrated to the user code.

2 HARDWARE SET-UP

2.a CONFIGURATION OF THE ANALOG INPUTS

The analog input channels can be configured using the same procedure as described in the tutorial n°1 "Boost converter with MPPT for photovoltaic application". Hence, for the present application, the parameters presented in Table 2 are proposed:

Name	Channel #	Sensor	Sensitivity	Limits	Bbox gain	Bbox limits
U_g	1	LEM LV25-P	250 / 47k [V/V]	[-80V ; 80V]	8	[-3.4V ; 3.4V]
I_g	0	LAH50-P	50 [mV/A]	[-10A ; 10A]	4	[-2.0V ; 2.0V]
U_{DC}	3	LEM LV25-P	250 / 47k [V/V]	[80V ; 120V]	8	[3.4V ; 5.1V]

Table 2 : Recommended settings for the analog inputs.

3 SOFTWARE CONFIGURATION

3.a CONFIGURATION OF THE PWMS

As described in tutorial n°1, the configuration of the PWM modulators requires two steps:

- *Configuration of the frequency generator*

The following code configures the FreqGen #0 with a period defined by SWITCHING_PERIOD:

```
SetFreqGenPeriod(0, (int)(SWITCHING_PERIOD/FPGA_CLK_PERIOD));
```

- *Configuration of the PWM modulators (i.e. of the PWM channels)*

The following code lines allow to use the above-configured frequency generator as the time base for PWM #0 and PWM #1. They also select a triangular carrier signal for the corresponding modulators, as well as a dead-time of 400ns:

```
ConfigPWMChannel(1, 0, TRIANGLE, (int)(400e-9/FPGA_CLK_PERIOD));
ConfigPWMChannel(2, 0, TRIANGLE, (int)(400e-9/FPGA_CLK_PERIOD));
```

The phase of both PWM channels (with respect to FreqGen #0) can be adjusted freely. In the present example, identical carriers lead to the generation of a 3-level voltage waveform at the output of the inverter. Alternatively, the use of an inverted carrier (INVTRIANGLE) or of a relative phase of 180° on the second channel leads to the generation of a 2-level voltage waveform only.

```
SetPWMPhase(1, 0.0);
SetPWMPhase(2, 0.0);
```

Finally, in this example, the PWM channels are not immediately activated, because a state machine is specifically tasked to handle the various operating modes, and hence manage the activation of the PWM channels.

3.b CONFIGURATION OF THE ANALOG-TO-DIGITAL CONVERSION

As first described in the tutorial n°1, the analog-to-digital conversion must also be configured on the software side in order to directly exploit meaningful floating-point quantities in the user control routines. To this end, SetADCAdjustements() must be invoked during the initialization phase and be fed with the desired sensitivity and offset parameters. In the present example, the latter are as follows:

```
SetADCAdjustments(0, 6.10e-3/4, 0.0);     // Nominal sensitivity and x4 gain (Ig)
SetADCAdjustments(1, 57.4e-3/8, 0.0);     // Nominal sensitivity and x8 gain (Ug)
SetADCAdjustments(3, 57.4e-3/8, 0.0);     // Nominal sensitivity and x8 gain (Udc)
```

In this application, the sampling can advantageously be achieved in the middle of the switching period, that is to say at the exact instant when the current – including its ripple – is equal to its sliding-average value. To do so, the sampling is configured in the middle of the period defined by the frequency generator #0 (the time base used for the PWMs as well):

```
ConfigSampling(0, 0.5);
```

3.c CONFIGURATION OF THE INTERRUPTS

Unlike the example presented in the tutorial n°1, this application requires only one interrupt that can typically be mapped to the **UserInterrupt1()** service routine. This interrupt can typically be configured such that it is triggered at the beginning of the period defined by **FreqGen #0** by means of the following function call:

```
RegisterExt1Interrupt(&UserInterrupt1, 0, 0.0, 0);
```

4 IMPLEMENTATION OF THE CONTROL APPLICATION

4.a DEFINITION OF THE USER STATE MACHINE

In this application, several operating modes can be provided. For instance:

1) The **STANDBY** mode corresponding to the idle state, i.e. when inverter is completely inactive.
2) The **CLOSEDLOOP** mode, during which the current control is effective and the inverter is actually switching, provided that the PWM outputs have been released (the command **enable** has been passed in the command-line interface).
3) The mode **PASSIVELOAD**, during which no closed-loop control is achieved. On the contrary, the modulation indices are generated directly such that they produce sinusoidal waveforms, irrespectively of any measurement. This mode may typically be useful in case of preliminary operation in a passive load and/or used for debug purposes.

Obviously, depending on the application and the corresponding needs, other operating modes could be defined, corresponding to other control strategies (e.g. a PR-type current control in a stationary reference frame) or related to the operation of another converter (e.g. the boost of the tutorial n°1).

In practice, these three operating modes are defined (in this example) using an **enum** type declared in **user.h**. The handling of these operating modes can typically be made by a state machine that will switch during each interrupt. These mechanisms can simply be implemented with a **switch** operator placed inside **UserInterrupt1()**. Part of the code is then executed differently as a function of the desired operating mode. This mechanism will be further presented in section **Erreur ! Source du renvoi introuvable.**.

Finally, in order to properly manage the transitions between the different state, it is generally recommended to use a specific routine such as **SetOpMode()** (already proposed in the code example). This routine is mainly responsible for executing the tasks related to the *state transition*. This typically includes the activation/deactivation of the PWM channels, the switching of relays (using the digital outputs of the BoomBox), or the modification/initialization of setpoints. An example is given hereafter:

```
switch (newmode){
        case STANDBY:           DeactivatePWMChannel(1);
                                DeactivatePWMChannel(2);
                                Ig_dq0_ref.real = 0.0;
                                Ig_dq0_ref.imaginary = 0.0;
                                break;
        case CLOSEDLOOP:        ActivatePWMChannel(1);
                                ActivatePWMChannel(2);
```

```
                                    break;
            case PASSIVELOAD:       ActivatePWMChannel(1);
                                    ActivatePWMChannel(2);
                                    break;
            }
            opmode = newmode;
}
```

4.b DEFINITION OF THE MAIN INTERRUPT SERVICE ROUTINE

As in numerous control applications, the main interrupt typically contains several steps:

1) Retrieval of the measurements using **GetADC()**. The exact sampling instant is that configured earlier using **ConfigSampling()**.
2) Execution of PLL(s) and, in the present case, of the fictive axis emulation.
3) Switch between the possible operating modes. The main ones typically contain:

   a) Coordinate transformations. In this example, only the following two lines are necessary, involving pre-defined routines that are available in the **API** folder:

   ```
   ABG2DQ0(&Ug_dq0, &Ug_ABG, theta);     // Grid voltage
   ABG2DQ0(&Ig_dq0, &Ig_ABG, theta);     // Grid current
   ```

   b) Execution of control algorithms. In **CLOSEDLOOP** mode, this mainly involves the execution of the current controllers.

   c) Inverse coordinate transformations. In the present case, only the converter EMF must be transformed back to the stationary reference frame :

   ```
   DQ02ABG(&Eg_ABG, &Eg_dq0, theta);     // Compute the converter EMF in station. R. F.
   ```

   d) Computation and update of the modulation indices. In **CLOSEDLOOP** mode, this task corresponds to the two following code lines:

   ```
   duty_a = 0.5 + 0.5* (Eg_ABG.real/Udc);
   duty_b = 0.5 - 0.5* (Eg_ABG.real/Udc);
   ```

   Alternatively, in **OPENLOOP** mode, the same task is achieved entirely *a priori* by means of the following instructions:

   ```
   duty_a = 0.5 + moddepth * sin(theta);
   duty_b = 0.5 - moddepth * sin(theta);
   ```

4) Sending of the updated modulation indices to the FPGA-implemented modulators.
5) Update of the state machine (*next state logic*).

As proposed in the code example, it can be noted that the next state is determined/computed directly inside the main section of the interrupt service routine. This is a reasonable choice since no automated transition is being tested (no change of state is achieved automatically). However, provided that such event must take place autonomously (e.g. as soon as a pre-charge is complete), it is generally recommended to set up a separate routine to process the *next state logic* and request the state transition.

4.c DEFINITION OF THE COMMANDS THAT ARE AVAILABLE TO THE USER

In addition to the blocking and release of the PWM signals (commands **enable/disable**), the command-line access of the BoomBox allows the user to freely define numerous actions that can be executed independently from CodeComposer Studio. The definition of these actions is made in the **cli_commands.c** file.

For the present example, the existing commands can be completed with the following actions:

a) **start**, which is meant to switch to **CLOSEDLOOP** mode.
b) **stop**, which is meant to switch to **STANDBY** mode.
c) **opmode**, which is meant to force the change of an operating state.
d) **setid**, which aims to set the setvalue for the current on the direct axis.
e) **setiq**, which aims to set the setvalue for the current on the quadrature axis.

As previously described in tutorial n°1, the configuration of these commands is made as follows:

1) Prototype the routines that will achieve the desired actions once invoked through the command-line interface. In the present case, the following routines are necessary:

```
void DoStart(unsigned int argc, char *argv[]);
void DoStop(unsigned int argc, char *argv[]);
void DoOpMode(unsigned int argc, char *argv[]);
void SetId(unsigned int argc, char *argv[]);
void SetIq(unsigned int argc, char *argv[]);
```

2) Associate the commands with the corresponding routines. To do so, the following lines must be added to **LoadCLIUserFunctions()** :

```
fs_mkcmd_user("start", DoStart);
fs_mkcmd_user("stop", DoStop);
fs_mkcmd_user("opmode", DoOpMode);
fs_mkcmd_user("setid", SetId);
fs_mkcmd_user("setiq", SetIq);
```

3) Define the exact content of the routines, in other words define the actions that these routines must achieve. The details can be found in the corresponding code example.

5 IMPLEMENTATION OF THE CLOSED-LOOP CONTROL

5.a CONFIGURATION AND EXECUTION OF THE CURRENT CONTROL

Numerous pre-defined control-related routines are available in the **API** folder, such as various controllers and PLLs. For the current control suggested by this tutorial, two PI controllers are necessary. Their use involves two steps:

- *Instantiation and configuration*

This step must be executed during the initialization phase and aims to create the corresponding pseudo-objects and to configure them properly. To do so, the following lines are necessary:

```
PIDController Id_reg, Iq_reg;
ConfigPIDController(&Id_reg, Kp, Ki, 0, 30.0, -30.0, SAMPLING_PERIOD, 10);
ConfigPIDController(&Iq_reg, Kp, Ki, 0, 30.0, -30.0, SAMPLING_PERIOD, 10);
```

- *Execution*

This step consists in executing repeatedly the control routine with a constant interrupt period. In the present application, a recommended approach to invoke the current controllers as follows:

```
Eg_dq0.real =       Ug_dq0.real
                    + RunPIController(&Id_reg, Ig_dq0_ref.real - Ig_dq0.real)
                    - OMEGA*LGRID*Ig_dq0.imaginary;
Eg_dq0.imaginary =  Ug_dq0.imaginary
                    + RunPIController(&Iq_reg, Ig_dq0_ref.imaginary -
                    Ig_dq0.imaginary)
                    + OMEGA*LGRID*Ig_dq0.real;
```

5.b CONFIGURATION AND EXECUTION OF THE PLL

As for the controllers, several variants of PLLs are available in the **API** folder. For instance, the implementation of a single-phase PLL based on a second-order generalized integrator (SOGI) can be achieved through the following steps:

- *Instantiation and configuration*

Similarly to the configuration of the controllers, a pseudo-object must be built and configured during the initialization phase using the following code lines:

```
SOGIPLL1Parameters SOGIPLL;
ConfigSOGIPLL1(&SOGIPLL, 10.0, 0.5, 0.1, OMEGA, SAMPLING_PERIOD);
```

It is worth noting here that **SOGIPLL** is a global variable, which should be instantiated in the beginning of **user.c** (or another pseudo-class containing the control mechanisms).

In the second line, the parameters 10.0 and 5.0 correspond to the gains of the PI controller that is used to lock the phase loop. The parameter 0.1 corresponds to the gain of the SOGI. The exact definition of these routines can be found in **API/PLLs.h**.

- *Execution*

This step consists in executing repeatedly the controller contained in the PLL and extracting the angle of the grid voltage **theta**. This is hence a task which must typically be achieved in **UserInterrupt1()** as suggested by the following line:

```
theta = RunSOGIPLL1(&SOGIPLL,&Ug_ABG,Ug_measured);
```

5.c CONFIGURATION AND EXECUTION OF THE FAE

The routines corresponding to this functional bloc are available in the **API** folder. Once again, two steps are necessary:

- *Instantiation and configuration*

As for the other pseudo-objects, this step must typically take place in **UserInit()**, by instantiating a pseudo-structure and calling a configuration routine:

```
FAEParameters FAE;
ConfigFAE(&FAE, RGRID, LGRID, SAMPLING_PERIOD);
```

The employed parameters RGRID and LGRID are here used to intuitively configure the transfer function of the FAE bloc. These are pre-compiler constants defined in user.h. The exact definition of these routines can be found in API/PLLs.h.

- *Execution*

The execution of the FAE bloc is achieved at the same time as the retrieval of the measurements and aims to emulate the missing axis β of the grid current. This hence typically involves:

```
Ig_ABG.imaginary = RunFAE(&FAE, Eg_ABG.imaginary - Ug_ABG.imaginary);
```

6 REFERENCES

[1] B. Bahrani, S. Kenzelmann and A. Rufer, *"Multivariable-PI-based dq current control of voltage source converters with superior axis decoupling capability,"* in IEEE Trans. Ind. Electron., Vol. 58, N° 7, Jul. 2011.

[2] B. Bahrani, A. Rufer, S. Kenzelmann and L. Lopes, "Vector control of single-phase voltage-source converters based on fictive-axis emulation," in IEEE Trans. Ind. Appl., Vol. 47, N° 2, Apr. 2011.

[3] M. Ciobotaru, R. Teodorescu and F. Blaabjerg, *"A new single-phase PLL structure based on a second-order generalized integrator,"* in Proc. PESC Conf., Rhodos, Greece, June 2006.

[4] F.J. Rodríguez, E. Bueno, M. Aredes, L.G.B. Rolim, F.A.S. Neves and M.C. Cavalcanti, *"Discrete-time implementation of second order generalized integrators for grid converters,"* in Proc. IECON Conference, Orlando, Nov. 2008.

[5] R. Teodorescu, F. Blaabjerg, M. Liserre, and P. C. Loh, *"Proportional resonant controllers and filters for grid-connected voltage-source converters,"* in IEE Proc. on Electr. Power Appl., Vol. 153, N°. 5, Sep. 2006.