Tutorial n°1

BOOST CONVERTER WITH MPPT FOR PHOTOVOLTAIC APPLICATION

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

Addressed topics:
- Configuration of the analog inputs
- Configuration of the interrupts
- Configuration of the sampling
- Use of the command-line interface
- Implementation of a basic closed-loop control

1 INTRODUCTION

This tutorial describes a possible approach to control a boost converter with the BoomBox. The considered application aims at interfacing a photovoltaic panel to a higher voltage source and to control the system such that a maximum power is extracted from the PV panel. The studied system is depicted in Figure 1. Its main electrical parameters are indicated in Table 1:

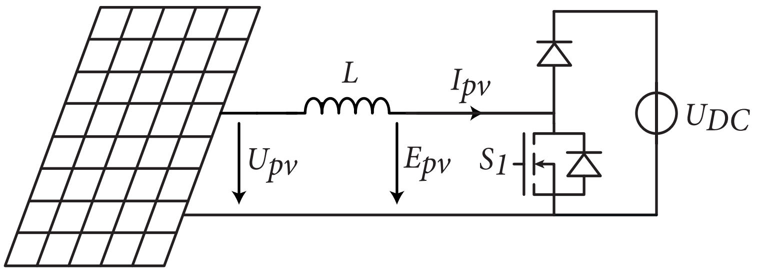

Figure 1 : Simplified electrical scheme of the system.

Name	Value	Specification	Employed sensor	Channel #
I_{PV}	0-8A	Current extracted from the solar panel	LEM LAH50-P	0
U_{PV}	10-25V	Voltage across the solar panel	LEM LV25-P	1
U_{DC}	100V	DC bus voltage (fixed)	IX AMC1200	2
I_{DC}	0-2A	Current injected into the DC bus	N.A.	N.A.
f_S	10kHz	Switching frequency	N.A.	N.A.
L	1mH	Smoothing inductor	N.A.	N.A.

Table 1 : Electrical parameters of the studied system.

Practically, a possible approach is to define the setvalue of the current extracted from the PV panel I_{PV} using an MPPT (maximum power point tracking) algorithm in order to maximise the exchanged power. Assuming that the DC bus voltage is constant, a reasonable implementation strategy is to place a fast control loop on the current I_{PV}, whose setpoint is updated periodically by an MPPT algorithm which is executed on a longer time basis.

2 HARDWARE SET-UP

In order to control the Boost (step-up) converter, a unique gating signal S1 is necessary, which is wired on the PWM channel PWM #0 (low). Provided that the reverse conduction of the power switch is attractive, the complementary signal (high) may be wired as well.

Additionally, three measurements are necessary in order to implement the corresponding maximal control structure. These measurements are reported in Table 1. Depending on the design choices, some of these measurements are dispensable.

2.a CONFIGURATION OF THE ANALOG INPUTS

Once the measurements are connected to the BoomBox using RJ45 cables, the coresponding analog input channels must be configured on the BoomBox. The procedure is the following:

1) Select « analog inputs » on the frontpanel.
2) Select one of the channels that must be configured 0 (I_{PV}), 1 (U_{PV}) or 2 (U_{DC})
3) Depending on the type of sensor, choose between:
   a) The **single-ended**, low-impedance input (100 Ω). This choice is recommended with most of the LEM sensors.
   b) The **differential**, high-impedance input (3kΩ). This choice is recommended in most other cases. Attention must however be paid to the fact that these inputs are NOT galvanically isolated.
4) Choose between the gains in order to maximize the use of the full scale of the analog-to-digital converter [-10V ; +10V]. The employed values are here 4 (I_{PV}), 8 (U_{PV}) and 8 (U_{DC}).
5) Activate or deactivate the pre-filtering on each input. It is chosen here not to use any filters (which are then completely bypassed) and to rely on a perfectly synchronous sampling of the measured quantities.
6) When applicable, choose the cutting frequency of the filters. The latter are irrelevant here as the filters are disabled.
7) Set the safety thresholds at appropriate values with respect to the safety of both the user and the application. These thresholds are taken into account at the input of the AD converters. The proposed values for the present case are indicated in Table 2. They account for the sensitivity of the employed sensors, which are characterized in Table 3.
8) Activate the safety thresholds.
9) Save the configured parameters.
10) Repeat the procedure for the other channels.

Name	Max. limit	Min. limit	Output of the sensors	Limits on the BoomBox	Channel #
I_{PV}	10A	-2A	[-1mA ; 5mA]	[-0.4V ; 2.0V]	0
U_{PV}	25V	-5V	[-0.27mA ; 1.33mA]	[-0.2V ; 1.1V]	1
U_{DC}	120V	-5V	[-0.27mA ; 6.38mA]	[-0.2V ; 5.1V]	2

Table 2 : Safety thresholds programmed on the BoomBox.

Sensor	Type of output	Nominal sensitivity	Typ. uncalibrated accuracy
LEM - LV25-P	Current, single-ended	250 / RIN [V/V]	±1%
LEM - LAH50-P	Current, single-ended	50 [mV/A]	±0.25% @25°C
LEM - LAH50-NP/SP1	Current, single-ended	100 [mV/A]	±0.25% @25°C
LEM - LA25-NP	Current, single-ended	100 [mV/A]	±0.5%

Sensor	Type of output	Nominal sensitivity	Typ. uncalibrated accuracy
IX - AMC1200	Voltage, balanced diff.	5.4 [mV/V]	±1.8%
IX - ACS709	Voltage, balanced diff.	90 [mV/A]	±5%

Table 3 : Parameters of some sensors commonly used along with the BoomBox.

3 SOFTWARE CONFIGURATION

The configuration of the BoomBox's software is achieved through several ad-hoc routines that are presented hereafter. These routines are typically invoked during the initialization of the application, namely during the start-up of the BoomBox, i.e. in the UserInit() routine.

3.a CONFIGURATION OF THE PWMS

Each pair of complimentary PWM signals is controlled by its own modulator, which can be assigned/routed to any of the 4 frequency generators available in the BoomBox. Hence, the configuration of a PWM modulator is typically achieved in two steps:

- *Configuration of the frequency generator*

The following code allows to configure the frequency generator #3 with a period set to SWITCHING_PERIOD (here 100µs). In fact, the second argument of the routine is a number of clock ticks on a 30Mhz time basis. SWITCHING_PERIOD and FPGA_CLK_PERIOD are pre-compiler constants (#define) defined in user.h.

```
SetFreqGenPeriod(3, (int)(SWITCHING_PERIOD/FPGA_CLK_PERIOD));
```

- *Configuration of the PWM modulator (i.e. of the PWM channel)*

The following line configures PWM #0 such that it is triggered by the frequency generator #3 (defined above). It also selects a SAWTOOTH-type carrier and a dead-time of 400ns between the complimentary signals. This last argument is also defined in clock ticks at 30Mhz.

```
ConfigPWMChannel(0, 3, SAWTOOTH, (int)(400e-9/FPGA_CLK_PERIOD))
```

The following instruction defines PWM #0 as an active channel. This means that once the PWM outputs of the BoomBox are enabled (the command enable is passed in the command-line interface), the PWM outputs are directly produced. In more complex applications, or with several converters, this option allows to activate the PWM channels independently from each other, and independently from the main unblocking/blocking (enable/disable) of the gating signals.

```
ActivatePWMChannel(0);              // Activate the PWM #0 channel
```

Finally, the following line imposes a relative phase of 0 degree on PWM #0 with respect to its clock source (namely FreqGen #3). This is useful in multi-phase systems, but is irrelevant for the system considered here. The function call could hence be omitted here.

```
SetPWMPhase(0, 0.0);                // 0.0 degrees between FG #3 and PWM #0
```

3.b Configuration of the analog-to-digital conversion

Directly usable values can be made available in the control code, provided that the analog-to-digital conversion is also properly configured on the software side. In order to configure a given ADC channel, the routine **SetADCAdjustements()** must be invoked and fed with a sensitivity and an offset parameters corresponding to the conversion between the raw 16 bits conversion result (signed integer) and a meaningful floating-point quantity. Hence, when an ADC channel is read, the routine **GetADC()** directly returns a convenient quantity, according to:

$$y = ax + b$$

where y is the returned value, x the raw conversion result, and a and b the sensitivity and offset configured through **SetADCAdjustments()**, respectively. The sensitivity a of the entire conversion chain can be determined from the parameters of the employed sensors according to:

$$\frac{1}{a} = s \cdot G_{FE} \cdot \frac{32768}{10}$$

where s is the sensitivity of the sensor and G_{FE} is the gain programmed on the frontend of the BoomBox. In the proposed example, the following parameters must be used:

```
SetADCAdjustments(0, 6.10e-3/4.0, 0.0);  // Nominal sensitivity and x4 gain (Ipv)
SetADCAdjustments(1, 57.4e-3/8.0, 0.0);  // Nominal sensitivity and x8 gain (Upv)
SetADCAdjustments(2, 113e-3/8.0, 0.0);   // Nominal sensitivity and x8 gain (Udc)
```

The configuration of the sampling allows to define at which instant the measurements must be taken. In the proposed 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 ripple is equal to its average value. In order to do so, the sampling is based on the frequency generator #3 with a phase-shift of 180°. The corresponding instruction is the following:

```
ConfigSampling(3, 0.5);                  // Phase of 180° between FG #3 and sampling
```

3.c Configuration of the interrupts

As suggested earlier, it is proposed to configure two interrupts, hence defining two distinct sample-based control mechanisms:

- *Fast interrupt*

A fast interrupt is dedicated to the execution of the current control. The following code line configures the primary user-level service routine **UserInterrupt1()** to be triggered by the interrupt source #1 (there are two interrupt lines between the FPGA and the DSP). Additionally, this function also configures the interrupt source #1 to be mapped to the frequency generator #3 (that which also serves as the time base for the unique PWM modulator that is being used). The relative phase with respect to that clock is chosen to be 0 degrees (the interrupt is triggered at the beginning of the switching period) and no postscaling is used (each and every period of **FreqGen #3** produces a control interrupt).

```
RegisterExt1Interrupt(&UserInterrupt1, 3, 0.0, 0);
```

- *Slow interrupt*

A secondary user-level interrupt service routine such as UserInterrupt2() is dedicated to the execution of the MPPT algorithm. With the following line, this routine is mapped on the internal CPU timer, which is configured to generate interrupts every 10'000µs.

```
RegisterTimerInterrupt(&UserInterrupt2, 10000);
```

4 IMPLEMENTATION OF THE CONTROL APPLICATION

In this example, the overall amount of code is relatively limited. Therefore, the corresponding control operation can be directly coded in the user.c/.h files.

4.a DEFINITION OF THE USER STATE MACHINE

Being given the simplicity of this application example, no state machine is implemented. This is why PWM #0 is directly activated during the initialization. Hence, as soon as the BoomBox starts, the code contained in both user interrupts is directly executed. However, the PWM gating signals will not be physically produced until the enable command has been passed through the command-line interface.

4.b CONFIGURATION AND EXECUTION OF THE CURRENT CONTROL

Numerous useful routines that are common in the digital control of power electronic systems are available in the API folder. Among them, various types of controllers are available. For instance, the closed-loop current control suggested in this tutorial can be set up using such routines in only two steps:

- *Instantiation and configuration*

This step must be executed during the initialization phase (in UserInit()) and aims to create a pseudo-object corresponding to a controller and to configure it properly. To do so, the following code lines are necessary:

```
PIDController Ipv_reg;
ConfigPIDController(&Ipv_reg, Kp, Ki, Kd, 15, -15, SAMPLING_PERIOD, 10);
```

The exact definition of the prototype of this routine can be found in the file API/controllers.h.

It must be noted here that the variable Ipv_reg is inevitably a global variable, which is then preferably instantiated in the beginning of the user.c file (or any other pseudo-class containing the control routines).

- *Execution*

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

```
Epv = Upv - RunPIController(&Ipv_reg, Ipv_ref - Ipv);
```

It can be noted that several variants of controller structures are available (P, PI, I, PID) from a unique pseudo-object. In other words, the configuration routine is the same, irrespectively of what kind of controller the user desires to use.

4.c Definition of the commands that are available to the user

The command-line access offered by the BoomBox allows the user to pass specific commands to the application, independently from the debugging interface available in CodeComposer Studio. In addition to the blocking and release of the BoomBox (commands enable/disable), numerous functions can be freely defined by the user. The definition of the available actions is made in the cli_commands.c file.

In the present case, the available commands can be advantageously completed by:

a) setmppt, which is meant to activate or deactivate the MPPT algorithm (the command setmppt 0 disables the MPPT, while the command setmppt 1 activates it).
b) setipv, which allows the user to define the setpoint for the current I_{PV} (this obviously implies that the MPPT must be inactive).

Practically, the configuration of these commands is achieved as follows:

1) Prototype the functions that will execute the chosen actions and be invoked through the command-line interface. In the present case, there are namely:

```
void SetMPPT(unsigned int argc, char *argv[]);
void SetIpv(unsigned int argc, char *argv[]);
```

2) Associate the commands with the corresponding routines, in other words "register" the above-defined routine among those that are available by the command-line. In order to do so, the routine LoadCLIUserFunctions() must be completed with the following lines:

```
fs_mkcmd_user("setmppt", SetMPPT);
fs_mkcmd_user("setipv", SetIpv);
```

3) Define the exact content of these functions, i.e. define the actions the latter are supposed to execute. An example is given hereafter:

```
void SetIpv(unsigned int argc, char *argv[]){
        if (*argv[1] == '?'){
                printf("\nSet the PV current value (Ipv).");
                printf("\nExample : setIpv 1.2");
                return;
        }
        Ipv_ref = atof(argv[1]);;
}
void SetMPPT(unsigned int argc, char *argv[]){
        if (*argv[1] == '?'){
                printf("\nSet the MPPT state. (1 to activate and 0 to deactivate)");
                printf("\nExample : setMPPT 1");
                return;
        }
        enable_MPPT = atoi(argv[1]);;
}
```

Once defined, these actions are available in the virtual folder « user » of the BoomBox. The can be invoked as presented in the following example:

```
user@boombox / > user
```

```
user@boombox /user > enable
user@boombox /user > setipv 3.0
user@boombox /user > setmppt 1
user@boombox /user > disable
```

4.d DEFINITION OF THE FAST INTERRUPT SERVICE ROUTINE

The routine **UserInterrupt1()** typically begins with the necessary call to **GetADC()**. The exact sampling instant is that previously configured through the call of the **ConfigSampling()** routine. This corresponds to:

```
Upv = GetADC(1);                // Voltage on the PV panel
Ipv = -GetADC(0);               // PV panel current (sensor is positive outbound)
Udc = GetADC(2);                // DC bus voltage
```

Subsequently, the necessary control tasks may be executed and, finally, the modulation parameters updated, as presented by the following lines. The new duty-cycle is here applied to **PWM 0** and this information transferred to the actual modulator, located in the FPGA:

```
SetPWMDutyCycle(0, Epv/Udc);    // Refresh the duty-cycle of PWM #0
UpdatePWMData();                // Send the new PWM parameters to the FPGA
```

4.e DEFINITION OF THE SLOW INTERRUPT SERVICE ROUTINE

In order to continuously operate the solar panel at its maximum, the operating point must be constantly adjusted along the voltage-current characteristic of the panel. A typical characteristic is depicted in Figure 2 :

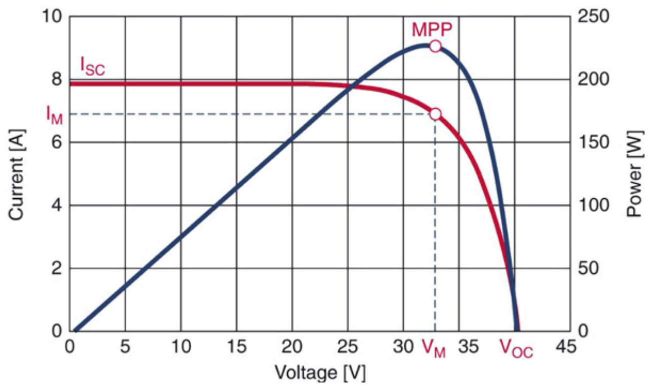

Figure 2 : Voltage-current characteristic of a photovoltaic solar panel (in red) and the corresponding profile of extracted power (in blue).

A simple and effective technique allowing to constantly maximize the extracting power consists in slightly perturbing the operating point and observing the corresponding effect on the transferred power (*perturb&observe* approach). A possible implementation is presented and commented in **UserInterrupt2().**