F2839379D DSP Architecture for Power Electronics Development

Eng. Eugênio Piveta Pozzobon
Eng. João Victor Lopes
Eng. Vinícius Descovi Rodrigues
GEPOC - UFSM

December 18, 2024

Contents

Chapter 1

Introduction

Developing power electronics will always include Embedded Development with a microcontroller. However, accurate embedded software development requires time. Sometimes, the controller documentation struggles with the developer, introducing barriers to power electronics research.

This document aims to provide a flexible architecture for power electronics deployment. It is based on the Texas Instruments F28379D microcontroller, allowing connection with Hardware-In-The-Loop (HIL) tools.

The selected microcontroller features two 32-bit CPUs, each operating at a clock frequency of 200 MHz, enabling parallel execution of algorithms. It contains 1 MB of flash memory and 204 kB of RAM. The microcontroller also includes four analog-to-digital converters, each with six channels, resulting in 24 channels. In addition, it has communication peripherals, timers, PWM functionality, and other features [1]. This DSP is also available in a development kit as illustrated in Figure 1.1.

Figure 1.1: Motherboard TMSF28379D

The firmware is implemented with the Code Composer Studio using C language. The integrated development environment (IDE) used to program microcontrollers from Texas Instruments (TI) is called Code Composer Studio (CCS), and a software development kit (SDK) is also provided to accelerate the development of tools with TI's microcontrollers.

Additionally, Typhoon Hill is introducing innovative tools like the Rapid Control Prototype (RCP), which automates the generation and management of embedded code, as well as the underlying Code Composer Studio. While this tool simplifies the workflow for engineers who may not be familiar with DSP, it also limits the developer's options to prototype testing alone. Also, this tool is currently in a early stage of development, requiring more refinements to provide a good programming interface in the C block.

In contrast, this paper seeks to offer a practical starting point for individuals, regardless of their DSP expertise. The proposed architecture is designed to be flexible and adaptable, catering to a variety of use cases.

This paper is structured as follows: Section 2 introduces a general architecture for code execution. Section 3 explains the peripheral setup, enabling this architecture to run with GPIOs, ADCs, and more. Section 4 discusses the architecture's memory management, and Section 5 discloses the overall firmware functions. Section 6 presents a simulation of a converter based on a Finite Control Set – Model Predictive Control, implemented with the proposed architecture. Section 7 concludes the document.

Chapter 2

Software Architecture

When developing power electronics, it is often common and good practice to use the ePWM as a trigger source for a function that handles all the code necessary to acquire data and control the converter while the main loop stores data, and communicates.

However, large interrupt functions may encounter issues depending on the controller's stack memory size. This can limit development options, as excessive function calls may be restricted, potentially causing failures in a production environment [2, 3]. It limits the implementation of heavy algorithms [4].

Additionally, this approach does not fit well in a controller like the F28379D, which supports multiple CPUs and co-processors. Thus, enabling parallel processing, while providing tools that help to keep a clean code in small functions. Using all the peripherals may also reduce the code execution time at the CPU, which allows testing higher frequency converters.

The architecture shown in Figure 2.1 is proposed for power electronics development firmware at the F28379D. Key aspects of this approach include:

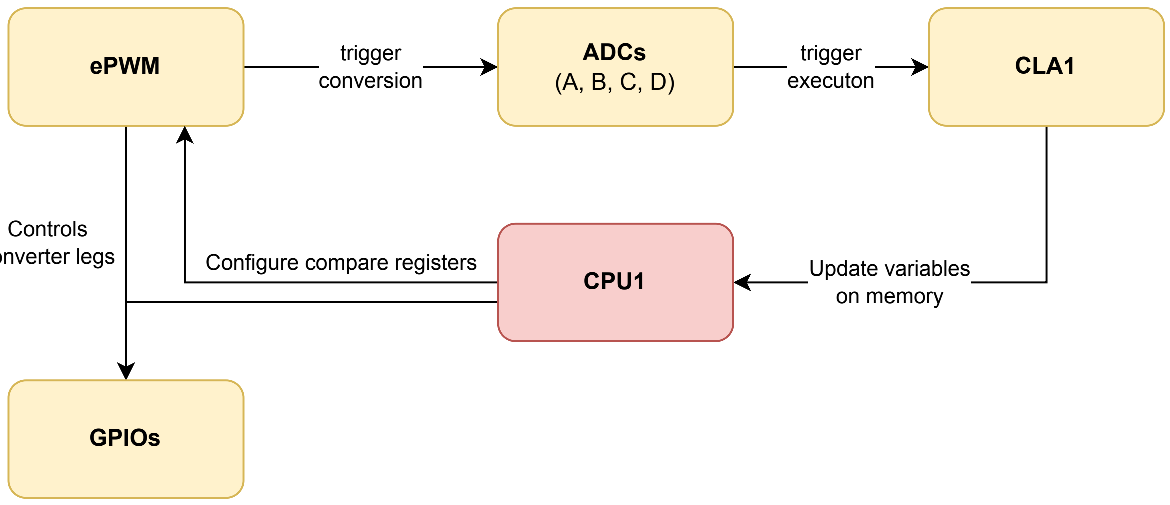


Figure 2.1: Proposed architecture

- The ADC sampling process is internally triggered by the ePWM module;
- A co-processor manages the reading and conversion of data from the ADCs;
- The main CPU executes the control code within the main loop function, avoiding any interrupt-driven execution.
- The code may be executed with function calls instead of a monolith interrupt function.

Chapter 3

Peripheral Setup

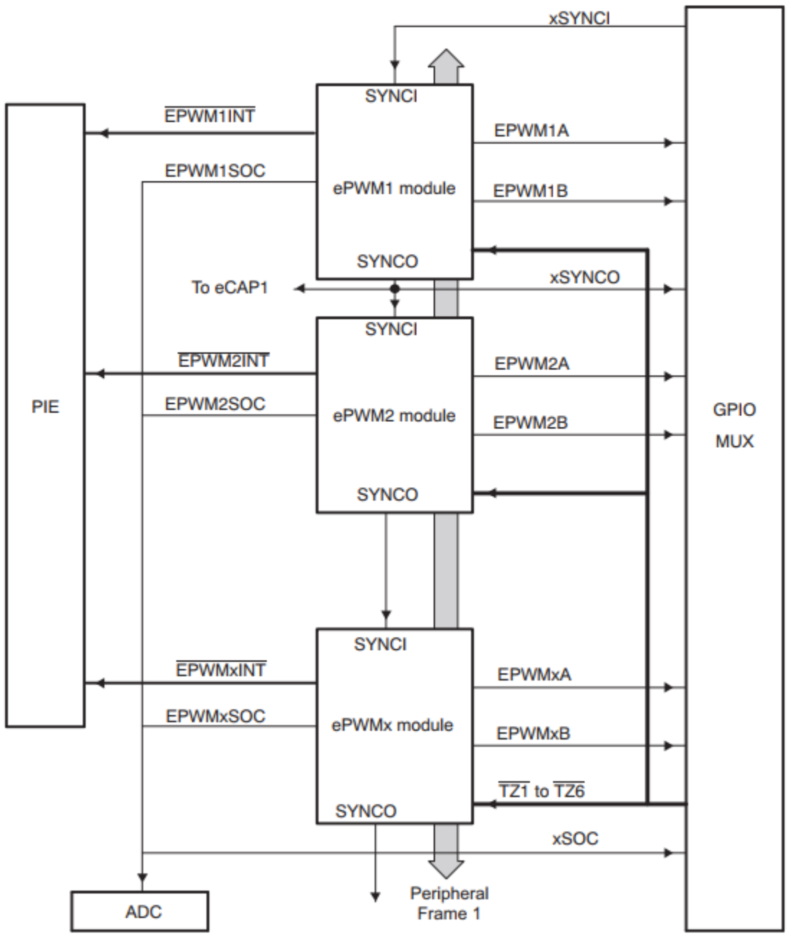
This section will present the microcontroller peripherals that must be set up to implement the proposed architecture. This includes the enhanced Pulse Width Modulator (ePWM), the General Purpose Input/Output (GPIOs), the Analogic to Digital Converters (ADCs), and the Control Law Accelerator (CLA)

3.1 ePWM module

The enhanced pulsed with modulator (ePWM) in DSP TMSF2839379D is built up from smaller single-channel modules with separate resources that can operate together as required to form a system. The ePWM module represents one complete PWM channel composed of two PWM outputs: EPWMxA and EPWMxB. Multiple ePWM modules are instanced within a device, each ePWM module is connected to the input/output signals as shown in Figure 3.1 [5]. There are seven submodules in each module of ePWM:

- Time-base(TB) module;
- Conuter-compare (CC) module;
- Action-qualifier (AQ) module;
- Dead-Band (DB) module;
- PWM-chopper (PC) module;
- Event-trigger (ET) module;
- Trip-Zone (TZ) module;

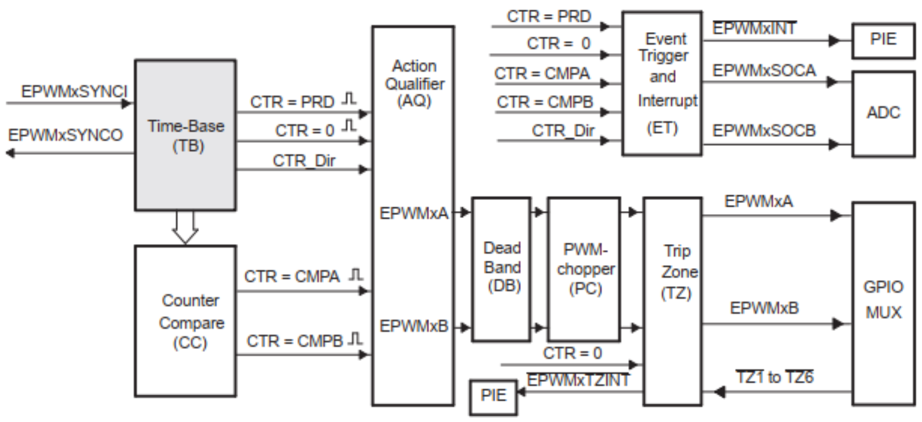
Figure 3.1: Multiple ePWM Modules [5].



3.1.1 Time-Base submodule

The time-base submodule determines all of the event timing for the ePWM module. The purpose is to specify the ePWM time-base counter (TBCTR) frequency or period to control how often events occur. Figure 3.2 presents the time-base submodule block diagram. The ePWM events can trigger actions like a GPIO toggle or start an analog channel conversion.

Figure 3.2: Time-Base Submodule Block Diagram [5].

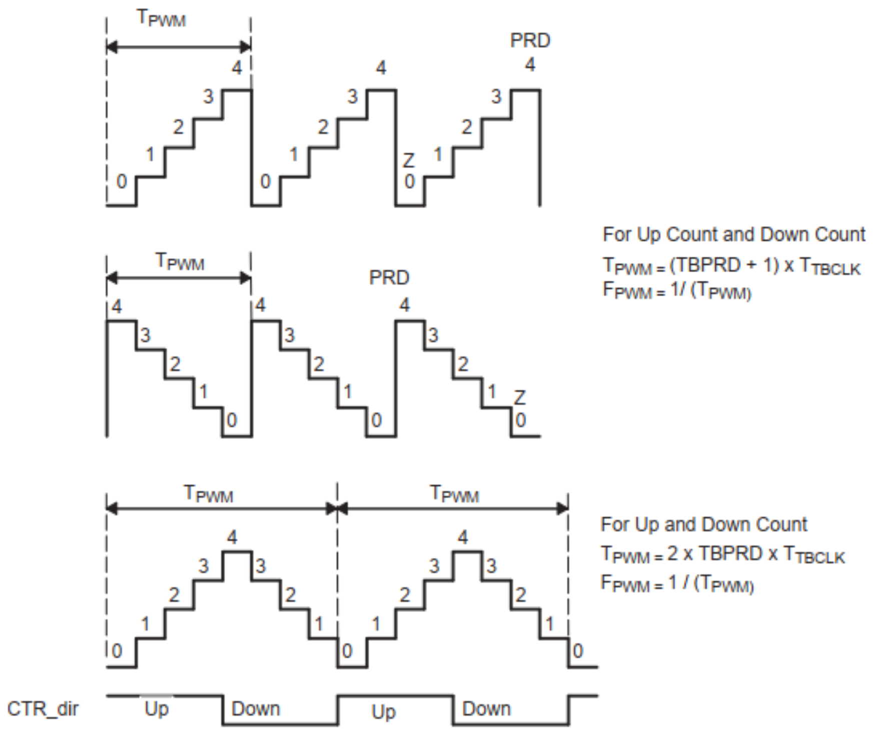


The time-base counter can be set to cout-up, count-down, or count-up-and-down mode as shown in Figure 3.3, generating the folowing events:

- CTR = PRD: Time-base counter equal to the specified period (TBCTR = TBPRD);
- CTR = Zero: Time-base counter equal to zero (TBCTR = 0x0000);

The time-base submodule is responsible for configuring the rate of the time-base clock, a prescaled version of the CPU system clock (SYSCLKOUT). This allows the time-base counter to increment/decrement at a slower rate.

Figure 3.3: Time-Base Frequency and Period [5]



T_PWM
PRD
For Up Count and Down Count
T_PWM = (TBPRD + 1) x T_TBCLK
F_PWM = 1/ (T_PWM)
For Up and Down Count
T_PWM = 2 x TBPRD x T_TBCLK
F_PWM = 1 / (T_PWM)
CTR_dir
Up
Down
Z

The following code presents a configuration for an ePWM to trigger the Start of Conversion (SOC) of the ADCs.

```
    // EPwm1Regs
    EPwm1Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // Count up-down
    EPwm1Regs.TBPRD = PWM_PERIOD; // Set timer period (max count)
    EPwm1Regs.TBCTL.bit.PHSEN = TB_DISABLE; // Disable phase loading
    EPwm1Regs.TBPHS.bit.TBPHS = 0x0000; // Phase is 0
    EPwm1Regs.TBCTR = 0x0000; // Clear counter
    EPwm1Regs.TBCTL.bit.HSPCLKDIV = 0; // Clock ratio to SYSCLKOUT
    EPwm1Regs.TBCTL.bit.CLKDIV = 0;

    //Enable the ADC Start of Conversion A (EPWMxSOCA) Pulse
    EPwm1Regs.ETSEL.bit.SOCAEN = 1;
    // Enable event (SOCA) time-base counter equal to zero or period
    EPwm1Regs.ETSEL.bit.SOCASEL = ET_CTR_ZERO;
    // ePWM ADC Start-of-Conversion on 1st ePWM Event
    EPwm1Regs.ETPS.bit.SOCAPRD = ET_1ST;

```

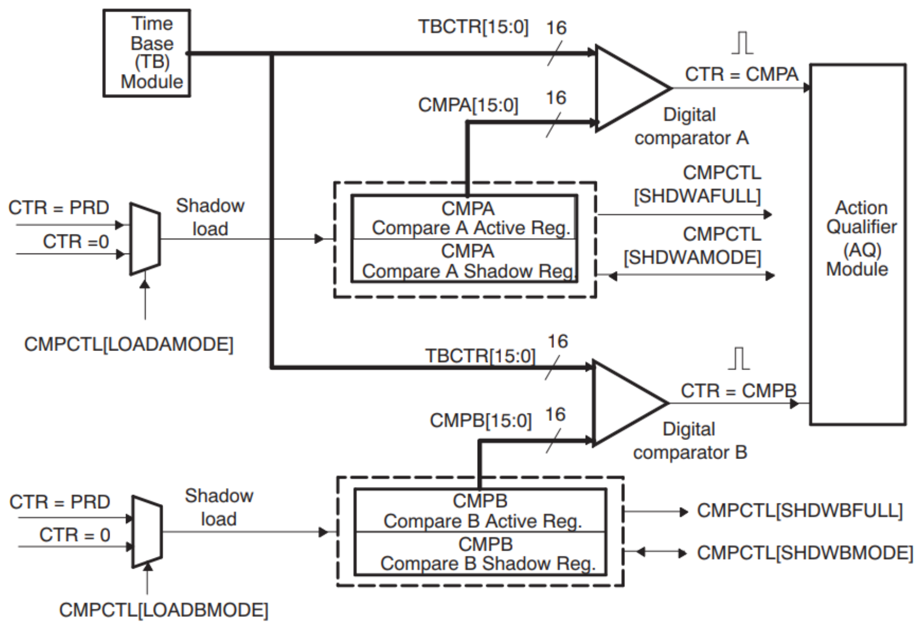
3.1.2 Counter-Compare submodule

The counter-compare submodule enables dynamic events by comparing the time-base counter with the counter-compare A (CMPA) and counter-compare B (CMPB) registers [5]. The counter-compare can:

- Generates events based on programmable time stamps using the CMPA and CMPB registers;
- CTR = CMPA: Time-based counter equals to counter-compare A register (TBCTR = CMPA);
- CTR = CMPB: Time-based counter equals to counter-compare B register (TBCTR = CMPB);
- Controls the PWM duty cycle if the action-qualifier submodule is configured appropriately;
- Shadows new compare values to prevent corruption or glitches during the active PWM cycle;

Figure 3.4 presents the detailed view of the counter-compare submodule.

Figure 3.4: Detailed view of the Counter-Compare submodule [5].

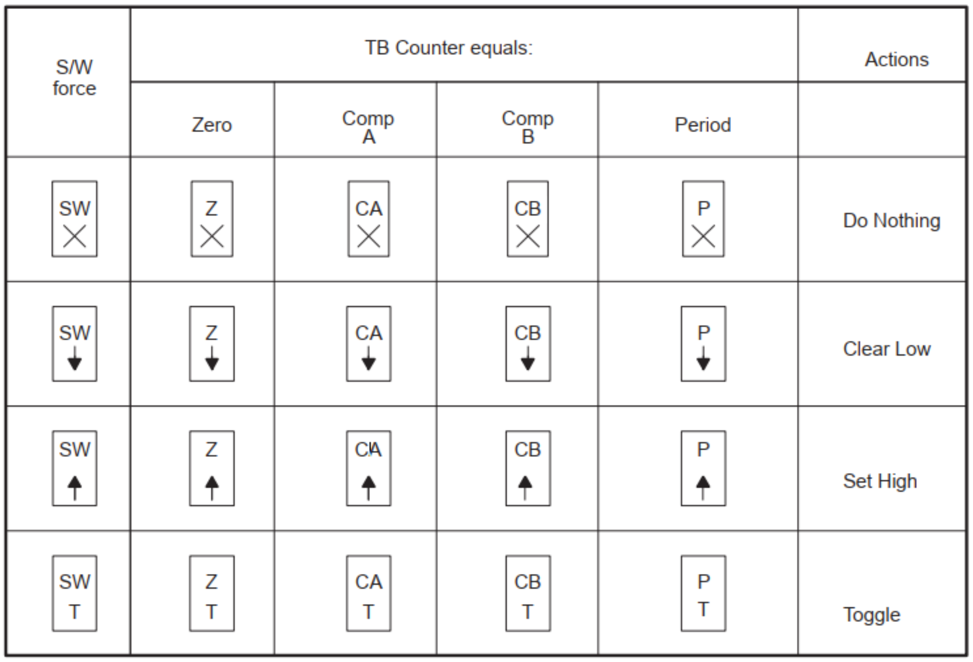


3.1.3 Action-Qualifier submodule

The action-qualifier submodule has an important role in waveform construction and PWM generation. It decides which events are converted into various action types, producing the required switched waveforms at the EPWMxA and EPWMxB outputs.

Figure 3.5 presents a set of symbolic actions. Each symbol represents an action as a marker in time. Some actions are fixed in time (zero and period) while CMPA and CMPB actions are flexible since their time and positions can be dynamically changed by setting a new value at the registers.

Figure 3.5: Possible Action-Qualifier Actions for EPWMxA and EPWMxB Outputs [5].



S/W force	TB Counter equals:				Actions
	Zero	Comp A	Comp B	Period	
SW ✕	Z ✕	CA ✕	CB ✕	P ✕	Do Nothing
SW ↓	Z ↓	CA ↓	CB ↓	P ↓	Clear Low
SW ↑	Z ↑	CA ↑	CB ↑	P ↑	Set High
SW T	Z T	CA T	CB T	P T	Toggle

3.2 GPIOs

The Texas Instruments F28379D microcontroller offers extensive GPIO (General-Purpose Input/Output) capabilities. With 97 GPIO pins, the F28379D provides flexibility for interfacing with external devices, sensors, and other peripherals. Each GPIO can be individually configured for input, output, or peripheral functions, allowing for versatile integration in complex systems. The GPIOs support fast switching speeds and can be configured with internal pull-up resistors.

```c
void DSP_setupGpios(void){
    ...
    GPIO_SetupPinMux(DEBUG_MPC_CPU1, GPIO_MUX_CPU1, 0);
    GPIO_SetupPinOptions(DEBUG_MPC_CPU1, GPIO_OUTPUT, GPIO_PUSHPULL);
    ...
}
```

After setting up GPIO pins, is possible to declare the GPIOs utilized for switching the controller with the ePMW, such as the configuration below.

```c
void main():
    DSP_setupGpios();

    // enable PWM1 and GPIOs
    CpuSysRegs.PCLKCR2.bit.EPWM1 = 1;
    CpuSysRegs.PCLKCR2.bit.EPWM2 = 1;
    CpuSysRegs.PCLKCR2.bit.EPWM3 = 1;
    InitEPwm1Gpio();
    InitEPwm2Gpio();
    InitEPwm3Gpio();
```

3.3 ADCs

The TMS320F28379D microcontroller features an integrated ADC subsystem with four modules (ADC-A, ADC-B, ADC-C, and ADC-D) enabling high-precision analog signal conversion. It offers up to 24 simultaneous sampling channels (six per module) supporting single-ended and differential input modes. While the ADC resolution is configurable from 12 to 16 bits, the setup used in this paper utilizes a 12-bit configuration to enable the use of all 24 channels.

The ADC supports both software and hardware triggered conversions using sample-and-hold circuits. An end-of-conversion (EOC) signal can trigger an ISR in either the CPU or the CLA [6]. While each ADC module can operate in parallel with a shared trigger source, channels within a single module are sampled sequentially. For example, channels A1 and A2 are sampled serially, whereas channels A1 and B1 can be sampled concurrently [7]. The total conversion time per channel is the sum of the sample and hold time (t_{SH}) and the conversion latency (t_{lat}).

Figure 3.6 shows the ADC input model for the microcontroller. The t_{SH} is defined by the time required for the capacitor C_h to reach the input signal voltage. It is possible to simulate the time dynamic in *LTspice*, depending on external circuit specificity. Figure 3.7 shows the implementation of the circuit in Figure 3.6 at the LTSpice, while 3.8 shows simulation results, which demonstrates that the $t_{SH} \leq 100$ ns [8] at an example circuit.

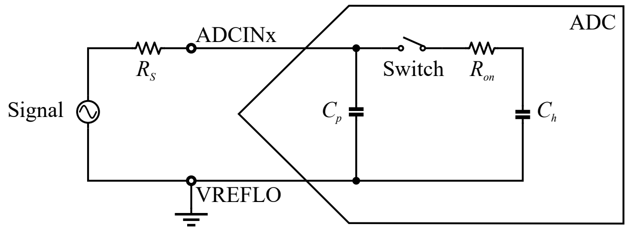


Figure 3.6: ADC general model [1]

On the other hand, the conversion time T_{lat} is the time from the end of the SH window until the ADC conversion results in a latch in the ADCRESULTx register. It is directly related to the clock prescaler of the ADC subsystem [1], which will be set up to $4\times$ to ensure acquisition reliability.

Therefore, $t_{SH} = 100$ ns, and the $T_{lat} = 260$ ns. Also, the start of the conversion process of all channels per ADC can be triggered by an internal ePWM, with a fixed sampling frequency of 250 kHz, as can be seen in the timing diagram of Figure 3.9 [9].

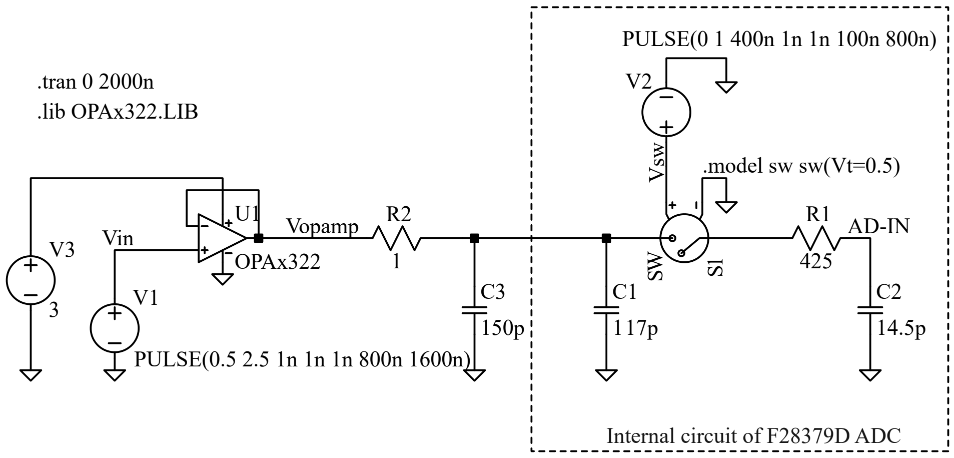


Figure 3.7: Circuit simulated in the LTspice

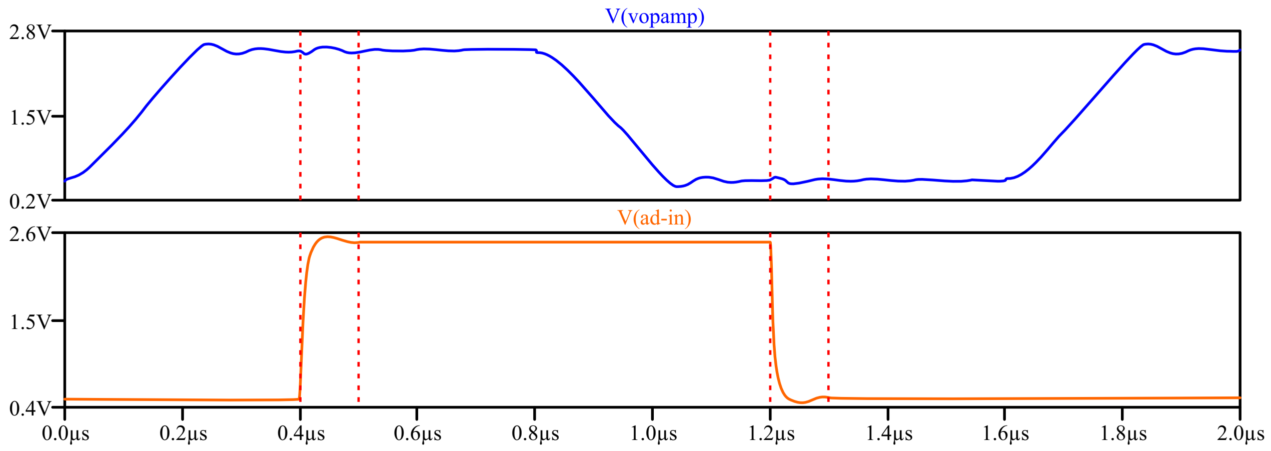


Figure 3.8: LTSpice simulation results. In blue is the real signal from the operational amplifier, while the red shows the sampled signal by the ADC model and the required timing of 100 ns for steady the ADC signal.

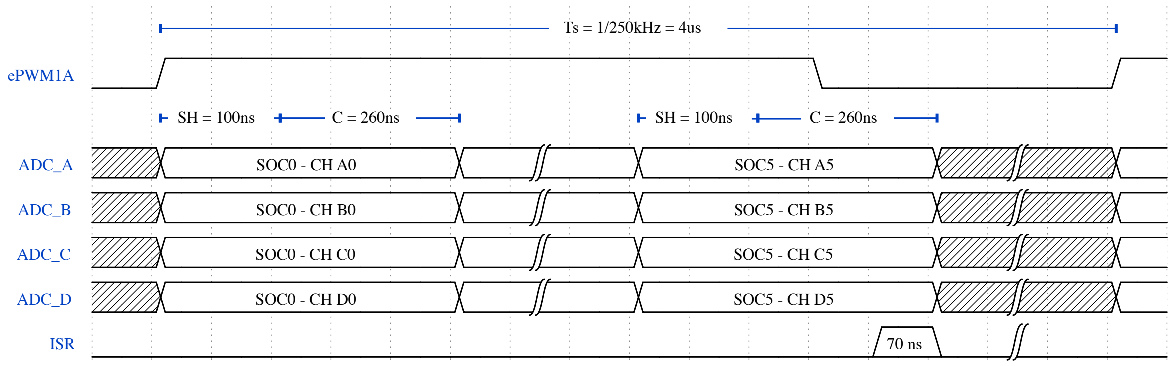
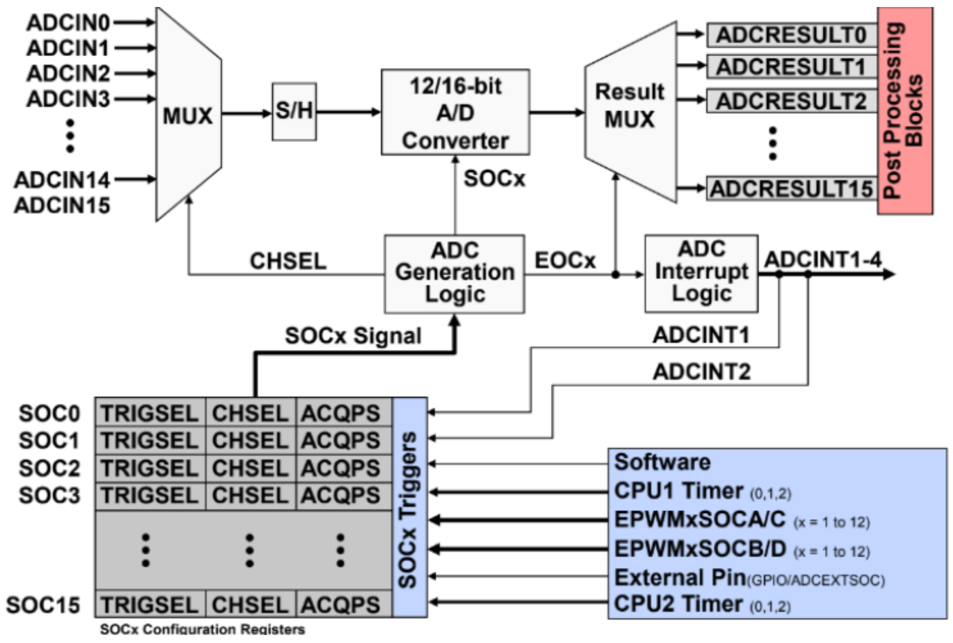


Figure 3.9: ADC Synchronous Operation for 24 channels — F28379D, ePWM1A trigger source

3.3.1 ADC Register setup

The ADC core consists of various analog circuits, including the channel selection multiplexer (MUX), sample-and-hold (S/H) circuit, successive approximation circuits, voltage reference circuits, and other supporting analog components. The control layer, made up of digital circuits, manages and configures the ADC. These digital circuits handle programmable conversion logic, result registers, communication with analog circuits, interfaces with peripheral buses, post-processing circuits, and connections to other on-chip modules.

Figure 3.10: ADC Block [1]



The ADC setup function may set up the following registers presented in this subsection.

Preescale

The PRESCALE register configures the frequency divider for the ADC (Analog-to-Digital Converter) module, allowing the ADC to operate at various clock frequencies based on the system clock and the desired conversion accuracy.

Currently, we are setting the PRESCALE to 8, which divides the system clock by 5, as detailed in Table 11-12 here [10], where ADC timing configurations for 12-bit resolution are provided. This will delay the conversion time. The resulting ADC frequency is calculated in Equation (3.1):

$$f_{CLOCKADC} = \frac{200MHz}{5} = 40MHz \tag{3.1}$$

Power UP/Down

The ADCPWDNZ bit controls whether the ADC module is powered on or off, reducing power consumption when the ADC is not needed.

- ADCPWDNZ = 0: Powers down the ADC module, placing it in an ”off” state. This reduces power consumption significantly, and no conversions can be performed while the ADC is off.
- ADCPWDNZ = 1: Powers up the ADC module, enabling it and making it ready to perform analog-to-digital conversions as configured.

In this setup, all four ADCs are configured in power-up mode. A 1 μs wait time is implemented to ensure the ADCs are fully activated before starting conversions.

```
    AdcaRegs.ADCCTL1.bit.ADCPWDNZ = 1;
    AdcbRegs.ADCCTL1.bit.ADCPWDNZ = 1;
    AdccRegs.ADCCTL1.bit.ADCPWDNZ = 1;
    AdcdRegs.ADCCTL1.bit.ADCPWDNZ = 1;
    DELAY_US(1000);
```

Select channel

The CHSEL function selects the ADC channel for each Start of Conversion (SOC) through the ADCSOCxCTL.CHSEL register, where ”x” represents the SOC number (SOC0, SOC1, ..., SOC15). Each ADCSOCxCTL register controls a specific SOC, allowing the user to assign the analog channel to be converted for that SOC.

In this setup, the CHSEL register is configured in Single-Ended Mode. With CHSEL set to 1, the SOC selects channel ADCIN1 for conversion.

Below is an example of this configuration applied to ADC-A. The same setup is repeated for the other ADCs.

```
    AdcaRegs.ADCSOC0CTL.bit.CHSEL = 0;  // SOC0 will convert pin A0
    AdcaRegs.ADCSOC1CTL.bit.CHSEL = 1;  // SOC1 will convert pin A1
    AdcaRegs.ADCSOC2CTL.bit.CHSEL = 2;  // SOC1 will convert pin A2
    AdcaRegs.ADCSOC3CTL.bit.CHSEL = 3;  // SOC1 will convert pin A3
    AdcaRegs.ADCSOC4CTL.bit.CHSEL = 4;  // SOC1 will convert pin A4
    AdcaRegs.ADCSOC5CTL.bit.CHSEL = 5;  // SOC1 will convert pin A5
```

Acquisition time

ACQPS (Acquisition Prescale) is a setting that controls the acquisition time (or window) for the ADC. This value, set in the ADCSOCxCTL register for each Start of Conversion

(SOC), defines the number of ADC clock cycles allocated to capture the input signal before the A/D conversion begins.

```
    AdcaRegs.ADCSOC0CTL.bit.ACQPS = ADC_ACQPS;
    AdcaRegs.ADCSOC1CTL.bit.ACQPS = ADC_ACQPS;
    AdcaRegs.ADCSOC2CTL.bit.ACQPS = ADC_ACQPS;
    AdcaRegs.ADCSOC3CTL.bit.ACQPS = ADC_ACQPS;
    AdcaRegs.ADCSOC4CTL.bit.ACQPS = ADC_ACQPS;
    AdcaRegs.ADCSOC5CTL.bit.ACQPS = ADC_ACQPS;
```

The ACQPS value can range from 0 to 63, corresponding to 1 to 64 ADC clock cycles. In the example applications, ACQPS is set to 19, providing 20 clock cycles for acquisition. Equation (3.2) shows how to calculate the total acquisition time of the ADC.

$$t_{SAMPLE} = (ACQPS + 1) * T_{ADCCLK} \tag{3.2}$$

Trigger source

The TRIGSEL register specifies the trigger sources for starting ADC conversions. Each ADC channel (SOC1 to SOC15) can be triggered by different sources. In this implementation, all SOC channels are configured to be triggered by the ePWM1A module, following the reference manual [10] and as demonstrated in the code below.

```
    AdcaRegs.ADCSOC0CTL.bit.TRIGSEL = ADC_TRIG_SOURCE;
    AdcaRegs.ADCSOC1CTL.bit.TRIGSEL = ADC_TRIG_SOURCE;
    AdcaRegs.ADCSOC2CTL.bit.TRIGSEL = ADC_TRIG_SOURCE;
    AdcaRegs.ADCSOC3CTL.bit.TRIGSEL = ADC_TRIG_SOURCE;
    AdcaRegs.ADCSOC4CTL.bit.TRIGSEL = ADC_TRIG_SOURCE;
    AdcaRegs.ADCSOC5CTL.bit.TRIGSEL = ADC_TRIG_SOURCE;
```

Interrupt Register

The ADC module is capable of generating interrupts on CPU or CLA once an analog signal has been converted to digital. These interrupts serve to inform the CPU that the conversion is complete, allowing the processor to process the results or execute control routines.

The INTPULSEPOS function determines when the interrupt occurs in relation to the ADC conversion process:

- INTPULSEPOS = 0: The interrupt is generated one cycle before the result is stored in the result register. Ideal for reducing system response latency.
- INTPULSEPOS = 1: The interrupt is generated at the end of the conversion, ensuring that the value is ready and stored. This is the safest and most common mode for normal processing.

```
    AdcaRegs.ADCCTL1.bit.INTPULSEPOS = 0;
    AdcbRegs.ADCCTL1.bit.INTPULSEPOS = 0;
    AdccRegs.ADCCTL1.bit.INTPULSEPOS = 0;
    AdcdRegs.ADCCTL1.bit.INTPULSEPOS = 0;
```

Finally, the ADCINTSEL1N2 register configures the ADCINT1 and ADCINT2 interrupts for the ADC module. These interrupts are triggered by the ADC whenever a conversion is completed on one of the specified sampling channels (SOC - Start of Conversion).

```
    AdcaRegs.ADCINTSEL1N2.bit.INT1CONT = 0;
    AdcaRegs.ADCINTSEL1N2.bit.INT1E = 1;
    AdcaRegs.ADCINTSEL3N4.all = 0;
```

In the first line, interrupts are generated until the user clears the ADCINT1 flag. Next, the ADCINT1 flag interrupt will be enabled and the ADCINT3 flag will be disabled.

3.4 CLAs

The CLA is a specialized co-processor build-in the Texas Instruments C2000 family of microcontrollers. Also, the CLA is a task-driven machine that can support up to 8 user-defined tasks. In addition, while operating in parallel with the C28x CPU, the CLA provides computational capability with a unique combination of minimal latency and ease of access to the key control peripherals [11].

The following code shows how to configure the CLA and set up the trigger source. Cla1Task1 refers to the CLA interrupt function that will read the ADC registers, while cla1Isr1 refers to the ISR schedule in the main CPU with the Cla1Task1 end. Optionally, this last interrupt can be disabled.

```
void CLA_initCpu1Cla1(void){
    EALLOW;
    Cla1Regs.MVECT1 = (uint16_t)(&Cla1Task1);
    ...
    Cla1Regs.MIER.all = 0x00FF;
    PieVectTable.CLA1_1_INT = &cla1Isr1;
    ...
    // 1 -> ADC A Interrupt CLA
    DmaClaSrcSelRegs.CLA1TASKSRCSEL1.bit.TASK1 = 1;
    IER |= (M_INT11);
    EDIS;
}

__interrupt void Cla1Task1 ( void ){
    // Reset interrupt flags
    EPwm1Regs.ETCLR.bit.INT = 1;
    AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;
    if(1 == AdcaRegs.ADCINTOVF.bit.ADCINT1){
        // Clear overflow flag
        AdcaRegs.ADCINTOVFCLR.bit.ADCINT1 = 1;
        AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;
    }

    // Read ADC regs
}
```

Chapter 4

Memory Management

The F28379D has 204 kB, or 102 KW of RAM available. Within this block, each CPU has a restricted Local Scratchpad Block (LS), which stores data accessible by the respective CLA and the CPU itself. On the other hand, another block named Global Scratchpad (GS) is accessible only by the CPUs, as shown in Figure 4.1 [7].

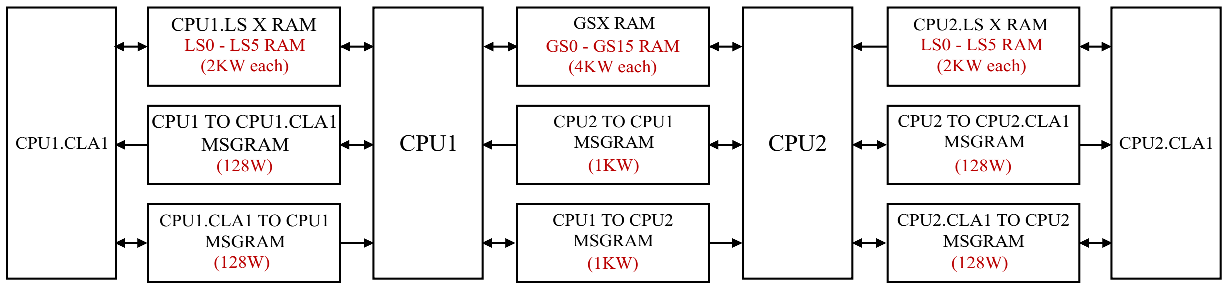


Figure 4.1: Simplified memory map of F28379D

Notably, the allocated memory space for CLA includes both program and data memory. Also, as it can be available for the respective CPU, the compiler determines that just the CLA has write access to these memory blocks [7].

Usually, from LS0 to LS5, the LS5 block is used for CLA program memory, and at least one beyond the remaining is dedicated to the run-time variables. It is worth mentioning that the developer must be aware of racing conditions between CLA and CPU. Once, the acquisition variables are constantly updated by CLA, the CPU may not overwrite it.

Also, the acquisition variables storage must be defined. This is set up by informing the memory block location of each variable using *pragma* definition. The three codes below show how to set up variables in the main.c code, and in the .cmd file. The .cmd file specifies memory allocation between LS and GS RAM [12]. Note that the variables are forcibly defined in the LS memory where the CLA is running.

```
// main.c file
...
#pragma DATA_SECTION(ADC_A_CH_RawData,"CLADataLS0");
```

```
uint16_t ADC_A_CH_RawData[6];

#pragma DATA_SECTION(ADC_B_CH_RawData,"CLADataLS1");
uint16_t ADC_B_CH_RawData[6];
...
```

```
// .cmd file

MEMORY{
    PAGE 0 :
        ...
    PAGE 1 :
        ...
        RAMLS0           : origin = 0x008000,   length = 0x000800
        RAMLS1           : origin = 0x008800,   length = 0x000800
        ...
}
SECTIONS{
    ...
    CLADataLS0  : > RAMLS0, PAGE=1
    CLADataLS1  : > RAMLS1, PAGE=1
    ...
}
```

4.1 Deployment options

Finally, three compiling options are related to the program memory setup and may be chosen depending on the deployment environment:

- In DEBUG mode compilation, the code is compiled and loaded into the RAM. If the microcontroller is turned off, it will not be able to execute the code since the RAM is a volatile memory. However, this is the ideal way to run tests with the microcontroller.
- In RELEASE mode, the program is loaded into the Flash memory. In this scenario, when the CPU boots, the system will copy the program from the Flash to the RAM to execute the code faster.
- The RELEASE STANDALONE mode has the same properties as the Release mode. This mode is used in the dual CPU implementation which will be addressed in the next section.

Each deployment option has a specific memory configuration, which is not detailed in this document. However, the provided examples include the necessary resources to run in any deployment mode.

Chapter 5

Main Function

The main function comprises the setup and run-time routines. The setup initializes the GPIOs, ePWM, and ADCs. For both CPUs, the setup routine may start each CLA, and instantiate the ISR functions and algorithm variables. Figure 5.1 represents the complete process executed in the CPU from setup to loop, which will be addressed in the next subsections.

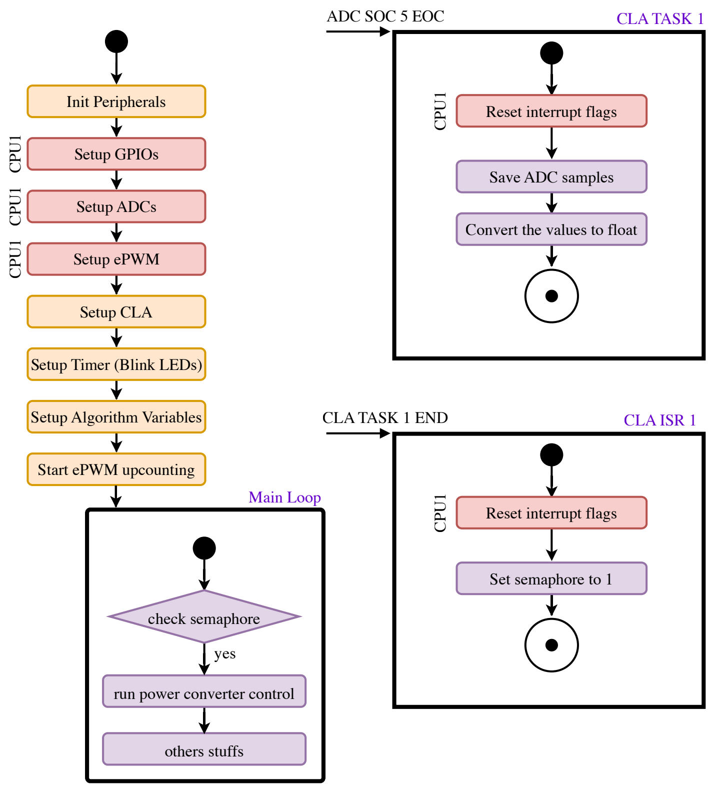


Figure 5.1: Firmware diagram.

5.1 Setup

CPU 1 sets up the GPIOs, ADCs, and ePWM. On the other hand, each CPU may configure its own CLAs, timers, and algorithm variables. Upon completing its setup, CPU 1 releases the ePWM to an up-counting state, thereby initiating the data acquisition process and power electronics controller.

5.2 Loop and CLA

The analog-to-digital sampling process starts for every ePWM trigger event. The ePWM triggers the ADC Sample-and-Hold; the ADC End Of Conversion (EOC) of ADC A, channel 5, triggers the CLA. Thus, the CLA can read any of the 24 channels based on user requirements, and the data is saved in LS RAM. An ISR at the main CPU is triggered upon the CLA tasks are finished. This task controls a semaphore that indicates to the loop function when is possible to run the control algorithm.

A semaphore is a synchronization basic concept used to control access to a shared resource by multiple threads. It holds a counter to manage concurrent access: threads decrease the counter when they enter (wait) and increase it when they leave (signal). When the counter is zero, additional threads trying to access the resource are blocked until others release it. This helps prevent race conditions and ensures controlled resource sharing [13].

The main loop function may include the control algorithm. It also can handle communication tasks and other stuff. However, it is mandatory to check the execution time of each function. While the control algorithm is running, the peripherals can work on sampling the next incoming sample data without depending on CPU commands.

Chapter 6

Storing Data

When working with the DSP, it is desirable to check the processed data and variables. One method for inspecting these values is by outputting them using the DSP's Digital-to-Analog Converter (DAC) and observing the analog signal with an external system like Typhoon HIL. However, this method only allows observation of a limited number of analog signals. A standard approach is to store a data buffer within variables in the DSP, and then access it with Code Composer.

The buffer approach can be used in RAM memory or Flash. Storing data in RAM is faster than storing it in Flash. Flash memory, on the other hand, allows for storing longer durations of real-time data, and retains it even after power interruptions due to its non-volatile nature.

Before implementing either approach, the available RAM or Flash memory should be determined after compiling the program without the buffer allocated. Finally, the buffer variable needs to be declared and mapped to the chosen memory section within the linker command (CMD) file. Here is an example of defining the Flash buffer:

```

// cmd file
PAGE 1 :
   FLASH_LOG           : origin = 0x084000,   length = 0x38000
   /* on-chip Flash C-D-E-F-G*/

// .c file
#pragma DATA_SECTION(log,"log")
float32 log[LOG_SAMPLES];
```

Updating a data buffer in RAM is a simple process, achieved by directly modifying the array values. Conversely, writing to a Flash memory buffer necessitates the use of the Texas Instruments Flash API, due to the specific programming procedures required for Flash memory. The subsequent code provides an illustrative example of how to save variables to Flash. The function `DSP_setup_flash_log` should be invoked once during the system's initialization phase, typically within `main.c`. In contrast, the function `DSP_log_variables` should be called repeatedly within the main program loop, enabling continuous logging.

The logging mechanism in this code is activated by setting the variable `logToggle` to one. The function `DSP_log_variables` employs a selector variable, `log_select`, that increments sequentially after each set of 4 samples are processed. This selector is used to cycle through different sets of variables for logging. Specifically, during each call to `DSP_log_variables`, it selects a group of four variables to be logged. This approach is implemented to constrain the number of variables written to Flash within a single call to the `DSP_log` function to four, which is beneficial when handling a large number of variables. A selector is needed as the total number of variables sampled by `DSP_log_variables` is up to 16.

```c
#include "F2837xD_device.h"     // F2837xD
#include "F2837xD_Examples.h"   // F2837xD
#include "F021_F2837xD_C28X.h"

#pragma DATA_SECTION(log,"log")
float32 log[LOG_SAMPLES];

uint32 current_log_index;
Fapi_StatusType oReturnCheck;

int allow_log_next_sample;
int logToggle;
float value;

// todo: include the measured variables as extern here

void DSP_setup_flash_log(){

    // Gain pump semaphore
    SeizeFlashPump();

    EALLOW;

    oReturnCheck = Fapi_initializeAPI(F021_CPU0_BASE_ADDRESS,200);
    if(oReturnCheck == Fapi_Status_Success){
        oReturnCheck = Fapi_setActiveFlashBank(Fapi_FlashBank0);
    }

    // Erase a Sector, 2000 per sector
    oReturnCheck = Fapi_issueAsyncCommandWithAddress(
    Fapi_EraseSector, (uint32 *)0x0084000);
    oReturnCheck = Fapi_issueAsyncCommandWithAddress(
    Fapi_EraseSector, (uint32 *)0x0086000);
    oReturnCheck = Fapi_issueAsyncCommandWithAddress(
    Fapi_EraseSector, (uint32 *)0x0088000);
    oReturnCheck = Fapi_issueAsyncCommandWithAddress(
    Fapi_EraseSector, (uint32 *)0x0090000);
    oReturnCheck = Fapi_issueAsyncCommandWithAddress(
    Fapi_EraseSector, (uint32 *)0x0098000);
    oReturnCheck = Fapi_issueAsyncCommandWithAddress(
    Fapi_EraseSector, (uint32 *)0x00A0000);
    oReturnCheck = Fapi_issueAsyncCommandWithAddress(
    Fapi_EraseSector, (uint32 *)0x00A0000);
    oReturnCheck = Fapi_issueAsyncCommandWithAddress(
    Fapi_EraseSector, (uint32 *)0x00A8000);
    oReturnCheck = Fapi_issueAsyncCommandWithAddress(
    Fapi_EraseSector, (uint32 *)0x00B0000);

    while (Fapi_checkFsmForReady() != Fapi_Status_FsmReady){}

    if(oReturnCheck != Fapi_Status_Success){
        // Do blank check. Check: https://www.tij.co.jp/jp/lit/ug/spnu629/spnu629.pd
        // Verify that the sector is erased.
        // oReturnCheck = Fapi_doBlankCheck((uint32 *)0x0084000, Bzero_16KSector_u32
    }

    EDIS;
}



void DSP_log(float *buffer){

    // for save log: tools > save memory > binary > 0x84000 > words = SIZE
    if (logToggle != 1){
        return;
    }

    if(current_log_index >= (uint32) LOG_SAMPLES){
        ESTOP0;
        logToggle = 0;
        return;
    }

    EALLOW;

    oReturnCheck = Fapi_issueProgrammingCommand(
            (uint32*) &log[current_log_index],
            (uint16*) buffer, 8, 0, 0, Fapi_DataOnly);

    EDIS;

    current_log_index += 4;

}

#define BUFFER_SIZE 4
int log_select;

void DSP_log_variables(){

    if (logToggle != 1){
        return;
    }

    GPIO_WritePin(DEBUG_LOG_TIMMING, 1);


    log_select = log_select>=4? 0: log_select+1;
//
    if (log_select == 0) {
        // Buffer 00 - 03
        float buffer_00[BUFFER_SIZE] = {valfagrid, vbetagrid, ialfagrid, ibetagrid};
        DSP_log(buffer_00);

        float buffer_01[BUFFER_SIZE] = {sigma_alfa, sigma_beta, epsilon_alfa, epsilon
        DSP_log(buffer_01);

        float buffer_02[BUFFER_SIZE] = {teta_alfa[0][0], teta_alfa[1][0], teta_alfa[2
        DSP_log(buffer_02);

        float buffer_03[BUFFER_SIZE] = {teta_alfa[4][0], teta_alfa[5][0], teta_alfa[6
        DSP_log(buffer_03);

    } else if (log_select == 1) {
        // Buffer 04 - 07
        float buffer_04[BUFFER_SIZE] = {teta_beta[0][0], teta_beta[1][0], teta_beta[2
        DSP_log(buffer_04);

        float buffer_05[BUFFER_SIZE] = {teta_beta[4][0], teta_beta[5][0], teta_beta[6
        DSP_log(buffer_05);

        float buffer_06[BUFFER_SIZE] = {valfagrid_fase, valfagrid_quad, vbetagrid_fas
        DSP_log(buffer_06);

        float buffer_07[BUFFER_SIZE] = {iaconv, ibconv, icconv, iagrid};
        DSP_log(buffer_07);

    } else if (log_select == 2) {
    // Buffer 08 - 11
        float buffer_08[BUFFER_SIZE] = {ibgrid, icgrid, vaconv, vbconv};
        DSP_log(buffer_08);

        float buffer_09[BUFFER_SIZE] = {vcconv, vabgrid, vbcgrid, vcagrid};
        DSP_log(buffer_09);

        float buffer_10[BUFFER_SIZE] = {ref_alfa_real, ref_beta_real, u_alfa, u_beta}
        DSP_log(buffer_10);

        float buffer_11[BUFFER_SIZE] = {zeta_alfa[0][0], zeta_alfa[1][0], zeta_alfa[2
        DSP_log(buffer_11);


    } else if (log_select == 3) {
        float buffer_28[4] = {valfaconv, vbetaconv, ialfaconv, ibetaconv};
        DSP_log(buffer_28);

        float buffer_29[4] = {ym_alfa, ym_beta, e_alfa, e_beta};
        DSP_log(buffer_29);


        float buffer_22[BUFFER_SIZE] = {Pk_beta[0][0], Pk_beta[1][1], Pk_beta[2][2], 
        DSP_log(buffer_22);

        float buffer_23[BUFFER_SIZE] = {Pk_beta[4][4], Pk_beta[5][5], Pk_beta[6][6], 
        DSP_log(buffer_23);


    }

}
```

Chapter 7

Case studies

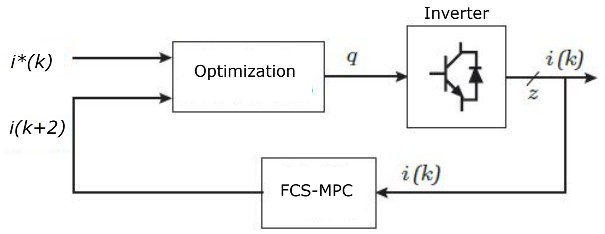
This section will demonstrate this architecture by implementing it on a FCS-MPC inverter.

7.1 FCS-MPC with two level three phase inverter

Model predictive control (MPC) has been a topic of research and development since the 1980s [14]. Originally, it was introduced in the process industry [15], but thanks to technological advancements in microprocessors, it has been proposed and studied as a promising alternative for power converter control.

The principle of this strategy is the prediction of the future states of the converter for all its switching vectors [16]. Through a cost function, the switching vector that results in the smallest error between the reference and the predicted state is chosen and implemented by the inverter as stated in [17]. In applications, both the absolute error equation and the quadratic error equation are used as cost functions. In Figure 7.1, the minimization diagram of the FCS-MPC is shown.

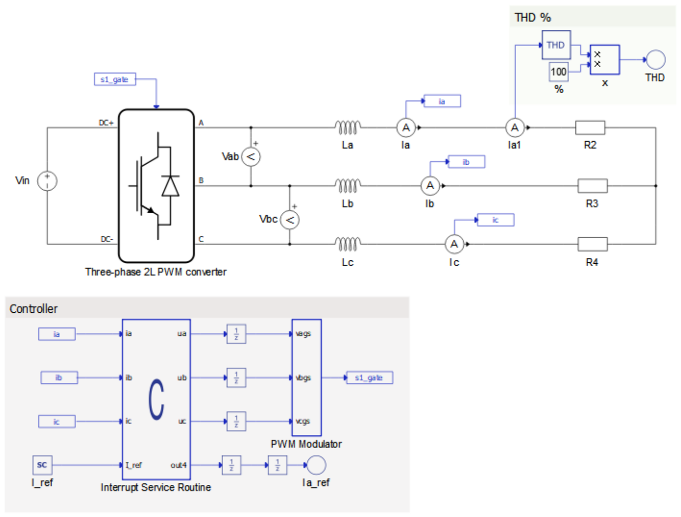
Figure 7.1: Diagram FCS-MPC [18]



7.1.1 Simulation

The proposed model was simulated in the Typhoon HIL software through the Schematic Editor, a feature available in the Typhoon HIL Control Center. In this way, through real-time simulation, it is possible to simulate the proposed model and verify potential corrections to be made.

Figure 7.2: Typhoon Schematic



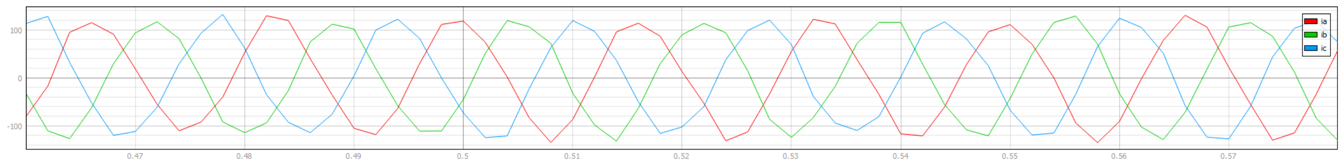
After validation in V-HIL, the control system of the model was implemented on the F28379D with the proposed architecture. In this context, it is important to note how the inverter legs were controlled.

In Typhoon HIL, the control mode is configured to operate per leg. Thus, the DSP may control the configuration of three GPIOs (one for each leg) via the ePWM in the next sampling period. Since the ADCs are triggered when the counter reaches zero, the GPIO event response can be adjusted in the AQCTLA.bit.ZRO registers as follows:

```
void mpc(){
    ...
    EPwm1Regs.AQCTLA.bit.ZRO = (vags == 1)? 2:1;
    EPwm2Regs.AQCTLA.bit.ZRO = (vbgs == 1)? 2:1;
    EPwm3Regs.AQCTLA.bit.ZRO = (vcgs == 1)? 2:1;
}
```

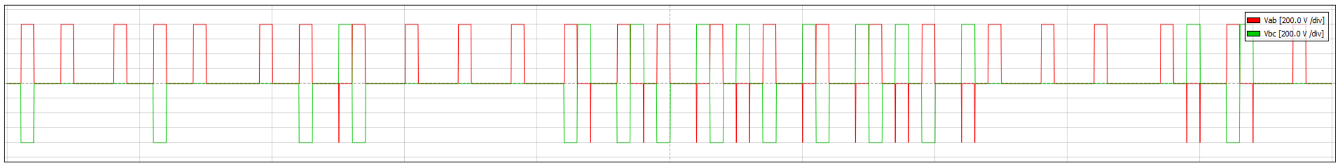
After the implementation on the F28379D using Code Composer Studio. The grid currents I_a, I_b, and I_c were presented in Figure 7.3. The current reference $I_{ref} = 150A$.

Figure 7.3: Grid Currents



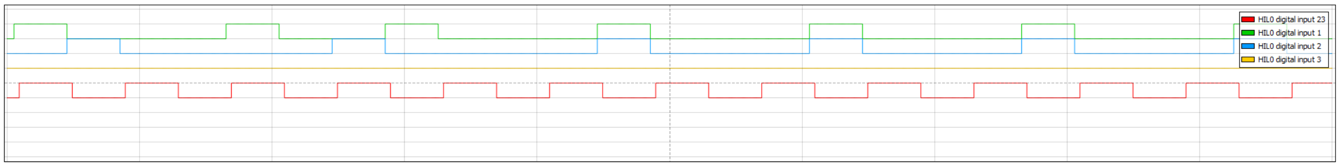
The line voltages of the grid V_{ab} and V_{bc} were presented in Figure 7.4.

Figure 7.4: Grid Voltages



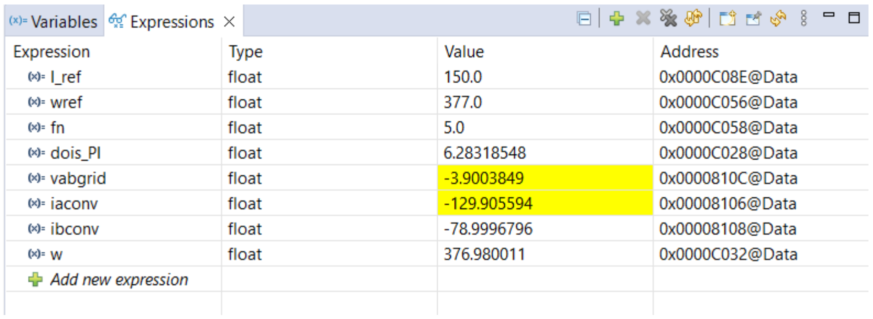
The switching signals of the inverter can be seen in Figure 7.5.

Figure 7.5: PWM



In n Figure 7.6, the variable monitoring environment in Code Composer Studio is presented, allowing both visualization and editing of the declared variables.

Figure 7.6: Monitoring Environment



Expression	Type	Value	Address
I_ref	float	150.0	0x0000C08E@Data
wref	float	377.0	0x0000C056@Data
fn	float	5.0	0x0000C058@Data
dois_PI	float	6.28318548	0x0000C028@Data
vabgrid	float	-3.9003849	0x0000810C@Data
iaconv	float	-129.905594	0x00008106@Data
ibconv	float	-78.9996796	0x00008108@Data
w	float	376.980011	0x0000C032@Data
Add new expression			

Chapter 8

Conclusion and Future Work

This document provides an introduction to F328379D components and their configuration. We have also successfully proposed and tested an embedded software architecture that enables the power electronics developer to use the maximum of the controller's capabilities.

The architecture code is available freely on GitHub as open-source code here: `https://github.com/Eugenio-Pozzobon/F28379D-GEPOC-Architecture`. The repository contains the FCS-MPC example project and a blank project to start as a template. Future work include developing examples of data buffering for storage in RAM or FLASH, as well as developing examples working with both CPUs.

Bibliography

[1] Texas Instruments, *TMS320F2837xD Dual-Core Delfino Microcontrollers*. Texas Instruments, 2021.

[2] RuntimeRec, “Optimizing response times in interrupt handling for embedded systems,” 2023. Accessed: 2024-11-06.

[3] SEGGER, “Memory allocation from interrupts in embedded systems,” 2023. Accessed: 2024-11-06.

[4] T. Instruments, “Ulp advisor rule 10.1: Minimize function calls from within isrs,” 2023. Accessed: 2024-11-06.

[5] Texas Instruments, “Reference guide: Tms320x2834x delfino enhanced pulse width modulator (epwm) module.” Disponível em: `https://www.ti.com/lit/ug/sprufz6b/sprufz6b.pdf?ts=1728438809435`. Acesso em: 08 de outubro 2024.

[6] Texas Instruments, “The TMS320F2837xD Architecture.” Application Note, 2016. Available at: `https://www.ti.com/lit/an/sprt720/sprt720.pdf`.

[7] Texas Instruments, *TMS320F2837xD Dual-Core Microcontrollers Technical Reference Manual*. Texas Instruments, 2019.

[8] Texas Instruments, “ADC Input Circuit Evaluation for C2000 MCUs.” Application Note, 2020. Available at: `https://www.ti.com/lit/an/spract6a/spract6a.pdf`.

[9] Texas Instruments, “C2000 ePWM Developer’s Guide.” Application Note, 2022. Available at: `https://www.ti.com/lit/an/sprad12a/sprad12a.pdf`.

[10] Texas Instruments, “Technical reference manual: Tms320f2837xd dual-core real-time microcontrollers.” Disponível em: `https://www.ti.com/lit/ug/spruhm8k/spruhm8k.pdf?ts=1730226022513`. Acesso em: 30 de outubro 2024.

[11] Texas Instruments, “Software Examples to Showcase Unique Capabilities of TI’s C2000™ CLA.” Application Note, 2020. Available at: `https://www.ti.com/lit/an/spracs0a/spracs0a.pdf?`

[12] Texas Instruments, “F2837xD Firmware Development Package.” Application Note, 2022.

[13] Oracle, *Using Semaphores*, 1997. Accessed: 2024-11-06.

[14] M. Morari and J. H. Lee, "Model predictive control: past, present and future," *Computers & Chemical Engineering*, vol. 23, no. 4, pp. 667–682, 1999.

[15] J. Richalet, A. Rault, J. Testud, and J. Papon, "Model predictive heuristic control: Applications to industrial processes," *Automatica*, vol. 14, no. 5, pp. 413–428, 1978.

[16] J. Rodriguez, M. P. Kazmierkowski, J. R. Espinoza, P. Zanchetta, H. Abu-Rub, H. A. Young, and C. A. Rojas, "State of the art of finite control set model predictive control in power electronics," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 2, pp. 1003–1016, 2013.

[17] S. Vazquez, "Model predictive control a review of its applications in power electronics," *IEEE industrial electronics magazine*, vol. 33, p. 16, 2014.

[18] J. V. Lopes Rosa, F. de Morais Carnielutti, H. Pinheiro, L. V. Belinaso, and L. F. Rissotto Menegazzo, "Validation of photovoltaic inverter controllers using automated tests and hardware-in-the-loop," in *2023 IEEE 8th Southern Power Electronics Conference and 17th Brazilian Power Electronics Conference (SPEC/COBEP)*, pp. 1–6, 2023.