F28377D General Control Board: Software Description

Version 1.0

Preamble

The trouble with trying to write documentation on software is that the code is changing quite frequently. I am continually trying to improve the code and add new features as I learn more about the F28377D microcontroller. Obviously people who choose to use the F28377D based control hardware will develop their own code. I am trying to provide a piece of example code that provides some of the basic functions that may be useful in converter control applications. It is hoped that this example code will help users to get started quickly by providing basic code for controlling the hardware features of the F28377D General Control Board (GCB).

Another thing is that I am far from being a professional software engineer. I am definitely a "bottom-up" type of person who likes to get pieces of code working with the hardware first and then work upwards. I don't "design" software in the sense that I start with a grand functional specification and then work downwards. One of the reasons for this is that I am more concerned with code being fast and efficient which I think is not a bad attitude for real-time control applications.

In some applications it may be practical to use code generators such as the Matlab Real-Time Workshop. This is obviously an attractive option if you don't like programming but it is important to understand that the resultant code is unlikely to be as fast as "hand-coded" software. Also, in shielding you from the details of the hardware/software interface, this approach can also be a handicap when things aren't working.

Justification for using the TMS320F28377D microcontroller

The F28377D is a dual-core processor. We are all aware that multi-core processors are prevalent on PC's and other similar general purpose computing hardware. On these platforms it seems to me that the value of a multi-core processor depends how well the application software or the operating system can schedule tasks to distribute computational load to the various cores. For example if you are running one application such as Matlab/Simulink, it would be nice if you had a high simulation speed because all cores were sharing the computational load. In many cases this will require very sophisticated software to break up what amounts to a single task into smaller tasks to run on the different cores. It was never my intention that we should have to get involved with this approach.

The way I see multi-core processors being useful in our field is when we have power electronic applications which naturally have a number of "loosely-coupled" concurrent tasks. Examples include systems with more than one converter unit. Some of the fault tolerant drives fall into this category in which we need to drive multiple sets of windings with multiple inverters. Another example is a basic 4-quadrant drive which has two back-to-back converters. Each converter can be controlled from a different core. Sensor-less drives usually have a speed/position estimator running in parallel with a motor controller. These two tasks could be assigned to different cores. In some of these applications, being able to assign the tasks to different cores will allow higher control sampling rates which would allow good control performance over an extended frequency range. This is obviously of benefit to high speed drives applications.

Although the F28377D is described as a dual-core processor, it actually has four processing units on the chip. Each of the two CPUs has an associated Control Law Accelerator (CLA). The CLA is a 32-bit, floating-point processing unit with the same clock frequency as the CPU. So in terms of computational capability it is equal to the CPU.

In addition to the computational benefits, the F28377D offers other benefits. When compared to its F28335 predecessor, the number of PWM channels has doubled to 24. This is obviously important for multi-level converter work and any application which uses multiple converter stages. Interleaved converters are of interest for reducing switching ripple or reducing the size of magnetic components. Again these systems require a large number of PWM channels. There are also more ADC inputs, DAC outputs, analogue comparators, and many other features not present in the F28335 predecessor.

Each CPU within the F28377D also contains another two computational accelerators in the TMU (Trigonometric Maths Unit) and the VCU (Viterbi, Complex Maths Unit). These units speed up various numerical calculations. The speed enhancements provided by these accelerators are invoked automatically by the C compiler and so do not require any consideration when writing code.

So, even if you only need one core there are still considerable benefits in using the F28377D compared to the F28335 based on the extra peripheral functions and the maths accelerators. It is also worth noting that if you only want to use one core then you need not concern yourself with the presence of the second CPU or the CLAs which are inactive by default.

Basic structure of the software

There are two source files in the project code which I have written.

Lab3_cpu1-27-11-15.c
test.cla

I have also modified the linker command file to assign memory blocks to CPU1 and CLA1.

The other source files are unmodified files produced by TI.

At present, only CPU1 and CLA1 are being used. The source code which is executed in CPU1 resides in Lab3_cpu1-27-11-15.c and the CLA code resides in test.cla.

Although the CLA1 is an independent processing core, it needs to be set up by code that is executed by CPU1. It is also necessary to set up message RAM areas for communication between CPU1 and CLA1. C variables used for CPU/CLA communication are declared with #pragma statements near the start of the .c source file. Code which actually executes on the CLA1 is written in test.cla.

In this project CLA1 is used for inverter control and its code includes a reference frame transformation. This transformation makes use of look-up tables for sine and cosine. The setting up of these look-up tables is done by CPU1 code. The RAM used for the look-up tables is first made accessible to CPU1 during the set up and then access is transferred to CLA1.

The command file 2837xD_RAM_lnk_cpu1.cmd contains code for the assignment of memory blocks. At present blocks LS4 and LS5 are assigned as CLA1 program memory and blocks LS0 and LS1 are assigned as CLA1 data memory.

File: Lab3_cpu1-27-11-15.c

There now follows a description of the code in the function main() which needs further explanation above what is provided by comments within the source file.

A number of #define statements are included to set various parameters such as the converter switching frequency, dead-time etc.

A list of #pragma directives follow which associate a number of variables to the CPU1/CLA 1 message RAM and the CLA1 data RAM blocks.

The 256 element array `dbuff[]` was defined to hold a sinewave for testing purposes. It is currently being used to output sinewave signals from the DACs.

The arrays `RX_message_buffer[i]`, `RX_message[i]` and `TX_message_buffer[i]` are initialised to zero. These arrays are used by the serial communications ISR for data transfer to and from the host GUI.

The macro `Cla1ForceTask8andWait()` is used to carry out a software trigger of CLA1 Task8 i.e.it executes an IACK instruction. Initialisation of CLA global variables cannot be specified in the CLA C source file (i.e. test.cla) in the same way as in standard C. Any global variables defined in the .cla file must be initialised by executing a CLA Task. It makes sense to use the lowest priority Task which is Task8.

After execution of various initialisation functions there is an endless loop. This loop flashes an LED on the GCB to confirm that the code is running. The variable `LoopCount` is updated inside the loop to act as a loop counter. At present the value in `LoopCount` is sent to the host GUI for information and has no other function.

Next the functionality of the interrupt service routines and functions are described.

```
Interrupt void MainCPU_ISR(void)
```

This is the main ISR executed by CPU1 which carries out the following:-

Perform the data exchange with the host PC (GUI)

Switch off PWM if stop button is pressed. In the test rig an emergency stop button was connected to GPIO8. This allows the PWM to be deactivated in the event of the serial communications to host/GUI failing.

Get encoder data and calculate speed. This code is identical to that used in F28335 example code.

Implement manual enable/disable of PWM based on GUI button. This code uses a software force and clear in the PWM trip zone to switch on/off the PWM outputs.

```
item1 = values[0];
item2 = values[1];
item3 = values[2];
```

The above code is associated with a check box on the GUI but does not do anything.

Acquire data from selected ADC inputs. The statements commented out are now used in CLA1.

Turn on/off relay drive MOSFETs. This code is current not being used.

Get resolver position data by calling function `RDC_test()`

Send data to DACs. The GUI slider2 is used to set the magnitude of a sinewave output from the 3 DACs. The array `dbuff[]` contains 1 cycle of a sinewave. A 2048 offset is applied to set the average value to half the DAC data range. The DAC has a 3V full scale output so this produces is a 1.5V offset.

Implement data store operation. The data store process is activated from the GUI "Store" button. When the "store" button is pressed, variable `StoreCounter` is set to zero. The data store code is similar to that used in the F28335 example code.

```
Interrupt void AuxCPU_ISR(void)
```

At present this ISR is not executed. It was used for testing a reference frame transformation.

```
Interrupt void Timer1_ISR(void)
```

This ISR is executed but most of its code is commented out. It was written primarily for testing purposes.

```
Interrupt void Scib_RX_ISR(void)
```

This ISR is triggered every time a character is received from the host PC (GUI). It waits for a start character (0x7E) and then builds a message using the HDLC protocol. When the complete message is received it is transferred to string `RX_message[].`

```
void GuiDataExchange(void)
```

This function is called from **MainCPU_ISR** and it transfers data to and from the host PC (GUI). Data received from the host is assigned to global variables from elements of `RX_message[]`. Data sent back to the host is copied into the elements of `TX_message_buffer[]`. At present, this array has 144 elements and most of the data comes from the data stores. This data is transferred to the Matlab GUI and is displayed graphically.

The data store is used to log up to 8 variables on each execution of **MainCPU_ISR**. This store is similar that used on the F28335 based software. The store data is sent to the host in blocks. A block consists of 8 consecutive samples of the 8 variables.

```
void InitGPIO(void)
```

This function sets up various GPIO pins.

```
void InitADCs(void)
```

This function applies some general settings to each of the 4 ADC modules including the ADC clock frequency, single-ended operation, 12-bit resolution.

```
void InitDAC(void)
```

This function initialises the 3 DACs. The DAC outputs can be used to monitor software variables in real-time. For example I have used DAC outputs to monitor modulation signals.

```
void InitCMPSS1(void)
```

```
void InitCMPSS3(void)
```

```
void InitCMPSS6(void)
```

Within the F28377D there are 8 analogue comparators connected to various ADC inputs. In this project 3 comparators are set up to trip the PWM off when current sensor outputs exceed a pre-set threshold. The threshold values can be found in #define statements at the start of the Lab3_cpu1-27-11-15.c source file. At present there is code for 3 comparators (CMP1, CMP3 and CMP6) which monitor SENSOR1, SENSOR2 and SENSOR3 respectively. The above functions apply identical settings for the 3 comparators.

```
void InitOutputXbar(void)
```

The F28377D has a cross-bar interconnection system which allows various internal signals to be routed through to GPIO. While developing the comparator-trip code, I used the output cross-bar to monitor various internal signals. It was only used in the code development and is no longer used. I have left the code in for possible future use.

```
void SetupADCEpwm(void)
```

This function is used to set up the individual ADC input sampling specifications. Please ignore any comments that reference the LVB board as these should have been removed. The first code section (4 lines) sets up an interrupt (INT1) to be triggered by SOC0.

The intension is that all ADC input samples are triggered by EPWM1. The Start-of-Convert (SOC) logic defines which ADC input channel is associated with a particular ADCRESULTx register. It also defines the sample and hold acquisition time and the trigger source for that particular input channel. As can be seen in the code, all input channels have the same acquisition time and trigger source.

```
void InitEPwm1(void)
```

```
void InitEPwm2(void)
```

```
void InitEPwm3(void)
```

```
void InitEPwm4(void)
```

The GCB has 4 complementary pairs of PWM outputs. The above functions each set up a pair. In this project EPWM1, EPWM2 and EPWM3 are used to drive the 3 legs of a 3-phase inverter. EPWM4 is used to drive a boost converter. The settings for EPWM1, EPWM2 and EPWM3 are the same with the exception that interrupt trigger code is present at the at the end of **`InitEPwm1()`** only.

IMPORTANT! – The GCB uses a MOSFET driver (SN75372) on each PWM output. As the SN75372 is an inverting driver, each PWM is set up as ACTIVE LOW (`PWM_ACTIVE_POL=1`) to compensate. It is therefore important to check the polarity of the gate switch signals and also that the dead-time is

correct. If this logic isn't correct, it is possible that the dead-time timing can result in overlapping ON gate drive signals causing shoot-through rather than preventing it.

The above functions also include code for the Trip Zone to determine what happens when an over-current trip event occurs. The settings in this code take into account the SN75372 inversion and therefore set the F28377D PWM output pins HIGH when a trip occurs.

IMPORTANT! – On the GCB it was found that when a trip event occurred, the F28377D outputs were switched to a high impedance state regardless of the trip zone settings. This appears to be a hardware bug on the F28377D. Further testing revealed that this bug is not present on the TI experimenter's board running identical software. The only theory we have at present is that the bug is in the PTP package version of the F28377D (used on GCB) and not in the ZWT package (used in TI experimenters board). Fortunately the presence of the logic inversion provided by the SN75372 allows us to accommodate the bug. The input of the SN75372 pulls high during the high impedance state which results in the SN75372 driving its output low for the trip condition. For most gate drive circuits this is the desired condition.

```
void InitScib()
```

This function sets up the SCI-B serial interface for host communications (GUI). This involves firstly setting the GPIO multiplexer so that the pins act as an SCI interface. Then code is included to set up the UART characteristic such as Baud rate, parity, stop bits etc. The final piece of code is concerned with the SCI-B FIFO settings. As programmed the FIFO triggers an interrupt on receipt of 1 word.

```
void InitScic()
```

This function is similar to **`InitScib()`** but is for the non-isolated serial interface which is not currently being used.

```
void ScibXmit(int a)
```

This function is called by **Scib_RX_ISR** and used to send a character to SCI-B.

```
void ScicXmit(int a)
```

This function is similar to **`ScibXmit`** but is not currently being used.

```
void InitRDC()
```

This function sets up SPI-A for communication with the AD2S1210 resolver to digital converter. The first section of code is concerned with programming the GPIO multiplexer to set the pins as an SPI interface. Then 5 GPIO pins are programmed as outputs to drive the SAMPLE\, A0, A1, RES0 and RES1 pins on the AD2S1210. The SPI baud rate was set to 12.5Mbaud.

The AD2S1210 has two modes of operation, Configuration mode and Normal mode. It was found that the SPI timing had to be adjusted to get successful data transfers for each of these two modes. The AD2S1210 is first put into Configuration mode in order to set the resolver excitation frequency to 6kHz. In Configuration mode, 8-bit SPI transfers are used. A 5us delay is then inserted to allow time for the SPI data transfer. Then the AD2S1210 is put into Normal mode and the SPI is changed to

16-bit transfers. Normal mode allows position or speed to be transferred depending on the state of address pins A0 and A1.

```
void RDC_test()
```

This function is called from **MainCPU_ISR** and is used read position or velocity from the AD2S1210. The AD2S1210 SAMPLE/ pin is first pulsed low for 1us to update its output registers. With the SPI it is necessary to send a dummy write before reading the data from the AD2S1210.

```
void Timer0_Init()
```

This function sets up Timer 0 which is used for measuring code execution times.

```
void Timer1_Init()
```

Timer 1 was set up to trigger interrupt **Timer1_ISR** which is direct into CPU1 INT13 and not through the PIE block.

```
void InitEQep1Gpio(void)
```

This function sets up GPIO so that the EQEP1 is enabled on the pins.

```
void Eqep1_Init()
```

This function sets up the EQEP1 interface. This set up is the same as used in the F28335 code.

```
void InitDataStoreArrays()
```

This function sets up 8 16-bit integer arrays of 1000 elements. These arrays are used for logging data during run-time. As is recommended, large arrays are set up in the heap. The for-loop immediately after was used for diagnostic purposes only.

```
void SetUpLookUpTable(void)
```

This function sets up cosine and sine look-up tables of 360 elements each. The floating-point arrays cos_t[360] and sin_t[360] are defined as global and are located in RAM LS0. This RAM block (Local Shared RAM) can be made accessible to the CLA1. At present they are used to make a reference frame transformation which is executed in Task1 of CLA1.

```
void CLA_configClaMemory(void)
```

After a processor reset there are no memory resources allocated to the CLA. This function sets up the memory resources for CLA1. Firstly the CPU1/CLA1 message RAMs are configured. Then the CLA1 program and data memory blocks are configured. Blocks within the Local Shared RAM (LS0-LS5) area can be made available to the CLA1. See the linker command file 2837xD_RAM_lnk_cpu1.cmd for further details on memory block allocation.

```
void CLA_initCpu1Cla1(void)
```

The only way code can be executed on the CLA is as Task. A CLA Task is like an interrupt service routine and there are a maximum of 8 possible Tasks (Task1 – Task8). CLA Tasks cannot be interrupted and once started will run until completion.

The first part of the code puts the address of each Task into the 8 MVECT registers. When the CLA receives a Task trigger (like and interrupt) the contents of the MVECT register is used to vector to the appropriate Task.

It is also possible to start a CLA Task from software in the CPU with the use of an IACK instruction. The next piece of code enables this feature.

When a CLA1 Task is completed, an interrupt to CPU1 is triggered. The next piece of code sets up these interrupts (one interrupt for each CLA Task). Although these end-of-tasks ISRs are executed they do not do anything significant.

The final piece of code in this function sets up the trigger sources for each CLA1 Task. At present Task1 is triggered by the ADC-A. All other Tasks can only be triggered by executing an appropriate IACK instruction on CPU1. Task1 is triggered from ADC-A which in turn is triggered from EPWM1 and so Task1 is synchronised to EPWM1.

```
__interrupt void cla1Isr1()

__interrupt void cla1Isr2()

__interrupt void cla1Isr3()

__interrupt void cla1Isr4()

__interrupt void cla1Isr5()

__interrupt void cla1Isr6()

__interrupt void cla1Isr7()

__interrupt void cla1Isr8()
```

The above interrupts are triggered to inform CPU1 that a CLA1 Task has ended. If an end-of-task interrupt is triggered then it is important that the associated ISR re-enables the interrupt flag in the PIE.

At present only Task1 and Task8 are being used. Although no longer the case, I have also experimented with running Task2-Task4 but without any significant code.

File: test.cla

This file contains all the CLA1 C-code which includes global variable definitions, Tasks and functions. It does not have a function main and as mentioned earlier, global definition statements do not contain initialisation.

```
interrupt void Cla1Task1 (void)
```

Task1 is the highest priority task and thus is executed first if there happened to be more than one Task triggered at the same time. As mentioned earlier, once started CLA Tasks run to completion before any other Task can start. In this project Task1 is set up to do the inverter control.

The statement `GpioDataRegs.GPBSET.bit.GPIO54 = 1` is used for checking timing of Tasks and code executions times on an oscilloscope.

```
sen1 = AdcaResultRegs.ADCRESULT2;           // Sensor1   (J17-1)      (HSEC-15)
sen2 = AdcbResultRegs.ADCRESULT1;           // Sensor2   (J17-2)      (HSEC-18)
sen3 = AdccResultRegs.ADCRESULT0;           // Sensor3   (J17-5)      (HSEC-31)
```

The above statements are used to sample 3 sensor inputs (current sensors). At present the sensor values are not used in control processing. They are however sent to CPU1 via the CLA1 to CPU1 message RAM. CPU1 code then sends the values to the host GUI for display for testing purposes.

The following code is used to modulate 3 PWM channels for open-loop operation of a 3-phase inverter. This is done with a reference frame transformation, angle-generation code and a space vector modulation (SVM) function. The SVM function is similar to that used in the F28335 example code.

After the SVM function call there are statements to apply the phase modulation values to the EPWM modulation registers.

At present the code activated (i.e. not commented out) is applying a fixed modulation of 1250. This corresponds to a 50% modulation on the 20kHz PWM. This has been set up to examine the operation and timing of the GCB driving Infinion gate drive board.

```
EPwm4Regs.CMPA.bit.CMPA = 780;          //Modulate leg T7/T8 (boost leg)
```

The above statement applies a fixed modulation to the a boost converter used in the test rig for testing the GCB.

The code for the remainder of the CLA Tasks (i.e. Task2 – Task8) follows. Code in Task2, Task3, and Task4 is for the examination of various CLA intrinsic functions and does not perform any useful purpose at present. The presence of GPIO SET and CLEAR statements within CLA Tasks was to examine timing of the different Tasks. As mentioned earlier, Task8 is used to initialise CLA1 global variables. At present only the statement `angle_f=0` is doing anything useful as it is used in Task1 for the angle generator.

```
void svm(float vds, float vqs, uint16_t *mia, uint16_t *mib, uint16_t *mic)
```

There follows the SVM function body (function header above). Apart from a few minor changes this function is the same as that used in F28335 example code.