Embedded Computing System Coursework Report

ECSCWGroup 53

Joshua Featherstone (2230253) BEng

I. TASK 6

1. *Describe the role of the Floating-Point Unit (FPU) in the C28379D microcontroller and explain why it is essential for real-time control applications.*

The FPU handles complex mathematical operations, being designed to efficiently carry out floating point calculations.

Its key features include high-complexity, high-performance calculations; high precision and dynamic range; and efficient algorithm implementation.

Real-time control applications demand both accurate control and fast response times to changing inputs. This is crucially supported by the FPU which helps provide faster loop execution, reduced calculation error and enables complex real-time operations like filtering and fourier transforms.

2. *Design a dual-core control system using the TMS320F28379D microcontroller. Highlight how the two CPU cores and their corresponding CLAs (Control Law Accelerators) can be used to optimize the performance of a motor control system.*

A dual core motor control system allows for parallel management of motor aspects. Each core's CLA enables it to offload complicated control calculations improving real time response which is crucial for a motor system.

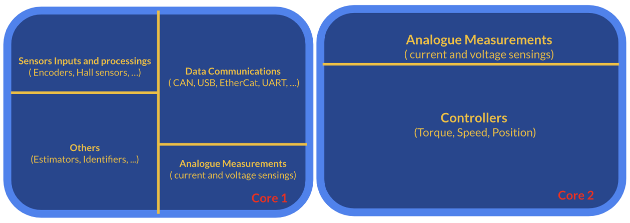

Sensors Inputs and processings
(Encoders, Hall sensors, ...)
Data Communications
(CAN, USB, EtherCat, UART, ...)
Others
(Estimators, Identifiers, ...)
Analogue Measurements
(current and voltage sensings)
Core 1
Analogue Measurements
(current and voltage sensings)
Controllers
(Torque, Speed, Position)
Core 2

Figure 1 - Motor Control CPU split

A possible task division would involve splitting Core 2 to control critical tasks (including analogue measurements and controllers for torque, speed, position) and Core 1 for data acquisition and minor sensors. This design also must ensure there is data communication between cores, but try to maintain isolation where possible.

Core 2's CLA would manage real-time control algorithms allowing high-priority control tasks to execute with minimal latency. Core 1's CLA could handle secondary tasks, ensuring the control loop remains uninterrupted while reducing the computational load on primary.

Having a separate crucial core un-affected by the tasks executing in the other, results in an extremely high bandwidth that can be focused on minimizing latency and maximizing performance in essential computations.

3. *Write a detailed report on how you would automate the process of building, testing, and deploying code to the TMS320F28379D microcontroller using continuous integration tools.*

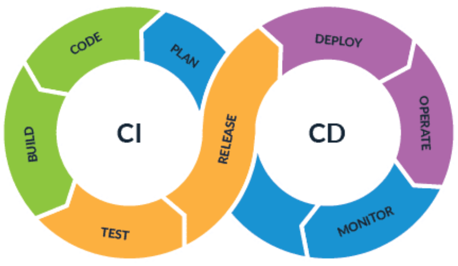
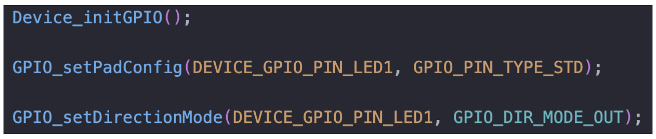


Figure 2 - CI/CD process

Begin by setting up a CI/CD pipeline with GitLab CI tooling. Use a Docker image containing CCS for consistent builds across environments. Configure the pipeline to build on each commit, compiling code via CCS command line. For testing, employ CCS scripting to run unit and integration tests. Finally, deploy to hardware using JTAG tools.

4. *Describe the basic steps to configure a GPIO pin on the TMS320F28379D microcontroller for input or output.*



```
Device_initGPIO();

GPIO_setPadConfig(DEVICE_GPIO_PIN_LED1, GPIO_PIN_TYPE_STD);

GPIO_setDirectionMode(DEVICE_GPIO_PIN_LED1, GPIO_DIR_MODE_OUT);
```

Figure 3 - GPIO config

Call Device_initGPIO() to initialize all GPIO-related configurations and allow access to configure specific pins. Use GPIO_setPadConfig(pin, mode) to set the pin to modes like standard push-pull. Use GPIO_setDirectionMode(pin, direction) to set direction as input or output. Can then write or read for either input or output.

5. *Question 5*

Implementation in Group53_Task5.5. To debounce either ignore subsequent presses for a fixed time after the first press, track button states or use a timer or delay after a debounce interval

6. *Question 6*

Implementation in Group53_Task5.6

7. *Question 7*

Implementation in Group53_Task5.7

8. *Explain the difference between GPxGMUXn and GPxMUXn registers in the TMS320F28379D microcontroller. How do they interact to configure the function of a specific GPIO pin?*

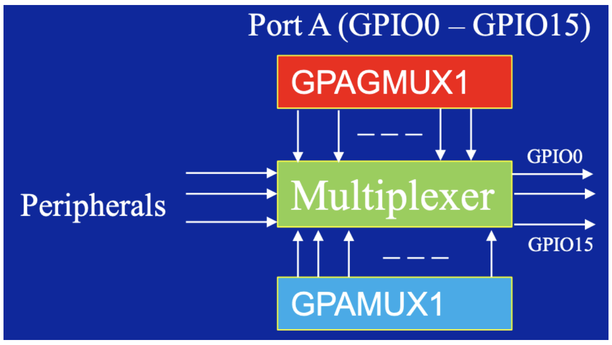
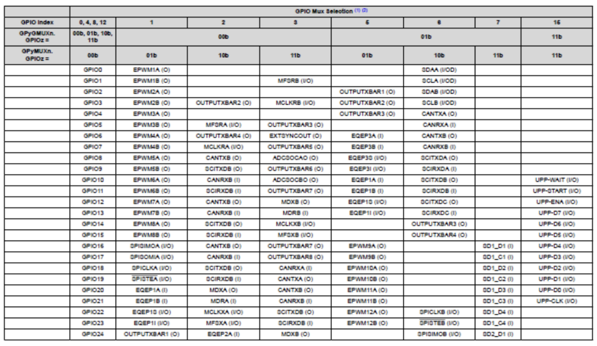


Figure 4 - GPxMUX registers

GPxGMUXn and GPxMUXn together map the various peripheral functions to the GPIO pins. Here x refers to the port number and n the specific pin number. The GPxGMUXn register has multiple bits that control the configuration of each pin. These allow the user to select the function of the pin, when it is set to a specific mode, from a set of predefined options. Whereas the GPxMUXn registers select the mode of the GPIO pin (input, output, or a peripheral function). They multiplex as by the below table



	GPIO Mux Selection (1) (2)							
GPIO Index	0, 4, 8, 12	1	2	3	5	6	7	15
GPyGMUXn. GPIOz =	00b, 01b, 10b, 11b	00b			01b			11b
GPyMUXn. GPIOz =	00b	01b	10b	11b	01b	10b	11b	11b
	GPIO0	EPWM1A (O)				SDAA (I/OD)		
	GPIO1	EPWM1B (O)		MFSRB (I/O)		SCLA (I/OD)		
	GPIO2	EPWM2A (O)			OUTPUTXBAR1 (O)	SDAB (I/OD)		
	GPIO3	EPWM2B (O)	OUTPUTXBAR2 (O)	MCLKRB (I/O)	OUTPUTXBAR2 (O)	SCLB (I/OD)		
	GPIO4	EPWM3A (O)			OUTPUTXBAR3 (O)	CANTXA (O)		
	GPIO5	EPWM3B (O)	MFSRA (I/O)	OUTPUTXBAR3 (O)		CANRXA (I)		
	GPIO6	EPWM4A (O)	OUTPUTXBAR4 (O)	EXTSYNCOUT (O)	EQEP3A (I)	CANTXB (O)		
	GPIO7	EPWM4B (O)	MCLKRA (I/O)	OUTPUTXBAR5 (O)	EQEP3B (I)	CANRXB (I)		
	GPIO8	EPWM5A (O)	CANTXB (O)	ADCSOCAO (O)	EQEP3S (I/O)	SCITXDA (O)		
	GPIO9	EPWM5B (O)	SCITXDB (O)	OUTPUTXBAR6 (O)	EQEP3I (I/O)	SCIRXDA (I)		
	GPIO10	EPWM6A (O)	CANRXB (I)	ADCSOCBO (O)	EQEP1A (I)	SCITXDB (O)		UPP-WAIT (I/O)
	GPIO11	EPWM6B (O)	SCIRXDB (I)	OUTPUTXBAR7 (O)	EQEP1B (I)	SCIRXDB (I)		UPP-START (I/O)
	GPIO12	EPWM7A (O)	CANTXB (O)	MDXB (O)	EQEP1S (I/O)	SCITXDC (O)		UPP-ENA (I/O)
	GPIO13	EPWM7B (O)	CANRXB (I)	MDRB (I)	EQEP1I (I/O)	SCIRXDC (I)		UPP-D7 (I/O)
	GPIO14	EPWM8A (O)	SCITXDB (O)	MCLKXB (I/O)		OUTPUTXBAR3 (O)		UPP-D6 (I/O)
	GPIO15	EPWM8B (O)	SCIRXDB (I)	MFSXB (I/O)		OUTPUTXBAR4 (O)		UPP-D5 (I/O)
	GPIO16	SPISIMOA (I/O)	CANTXB (O)	OUTPUTXBAR7 (O)	EPWM9A (O)		SD1_D1 (I)	UPP-D4 (I/O)
	GPIO17	SPISOMIA (I/O)	CANRXB (I)	OUTPUTXBAR8 (O)	EPWM9B (O)		SD1_C1 (I)	UPP-D3 (I/O)
	GPIO18	SPICLKA (I/O)	SCITXDB (O)	CANRXA (I)	EPWM10A (O)		SD1_D2 (I)	UPP-D2 (I/O)
	GPIO19	$\overline{\text{SPISTEA}}$ (I/O)	SCIRXDB (I)	CANTXA (O)	EPWM10B (O)		SD1_C2 (I)	UPP-D1 (I/O)
	GPIO20	EQEP1A (I)	MDXA (O)	CANTXB (O)	EPWM11A (O)		SD1_D3 (I)	UPP-D0 (I/O)
	GPIO21	EQEP1B (I)	MDRA (I)	CANRXB (I)	EPWM11B (O)		SD1_C3 (I)	UPP-CLK (I/O)
	GPIO22	EQEP1S (I/O)	MCLKXA (I/O)	SCITXDB (O)	EPWM12A (O)	SPICLKB (I/O)	SD1_D4 (I)	
	GPIO23	EQEP1I (I/O)	MFSXA (I/O)	SCIRXDB (I)	EPWM12B (O)	$\overline{\text{SPISTEB}}$ (I/O)	SD1_C4 (I)	
	GPIO24	OUTPUTXBAR1 (O)	EQEP2A (I)	MDXB (O)		SPISIMOB (I/O)	SD2_D1 (I)	

Figure 5 - GPxMUX multiplexing

9. *How are peripheral interrupts managed and prioritized in the TMS320F28379D, and what role does the PIE (Peripheral Interrupt Expansion) module play in this process?*

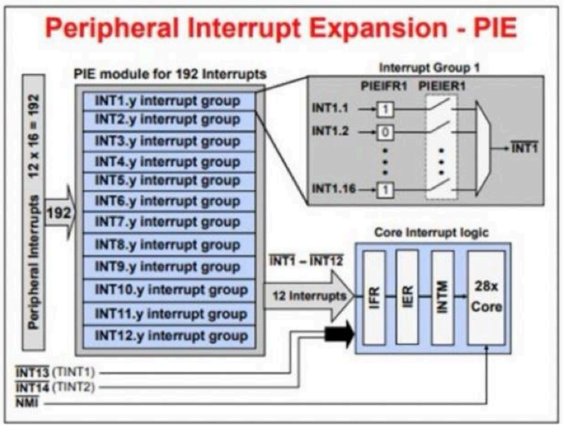


Figure 6 - PIE diagram

The TMS320F28379D has many peripherals that can generate interrupts, including timers, ADCs and peripherals. Each interrupt generates an interrupt request sent to the CPU (flag registers). The interrupt vector table stores the addresses of interrupt service routines, called as the CPU is paused.

The CPU has 14 interrupt lines, 2 from timers and 12 that pass through the expansion module PIE. The PIE combines up to sixteen peripheral interrupts into each of the 12 lines and gives each interrupt its own ISR. The mapping of peripherals to PIE groups is fixed and cannot be changed, ensuring correct handling and prioritization of interrupts. The output groups are then passed into core interrupt logic.

10. *Describe the process of configuring and handling non-maskable interrupts (NMIs) in the TMS320F28379D microcontroller. What are the unique challenges associated with handling NMIs compared to maskable interrupts, and how does the system ensure NMI sources are managed correctly?*

Non-Maskable Interrupts (NMIs) are a class of interrupts that cannot be disabled or ignored by the CPU, enabling critical events to always be handled promptly. NMIs are used for events requiring immediate attention, such as system faults, clock failures, or watchdog timer resets.

NMIs are configured with the NMICFG and NMIFLG flags, and the NMI interrupt vector for ISRs.. Each NMIFLG bit corresponds to a particular NMI source. Setting a flag in NMICFG enables the detection of that source, while writing to NMIFLG clears the flag once the source has been handled. NMIs have a dedicated interrupt vector that points to the NMI service routine, jumped to immediately by the CPU. This dedicated system allows them to be correctly managed

Uniquely, as NMIs cannot be ignored the CPU must be prepared with methods to immediately suspend a task, even a critical one. This can disrupt ongoing code or interrupts. They are also for serious faults so are tricky to debug or emulate.

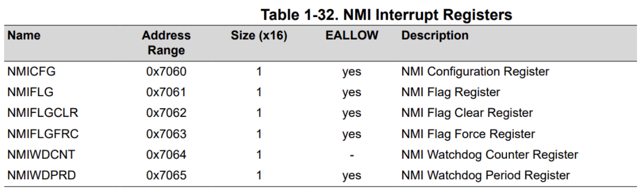


Table 1-32. NMI Interrupt Registers

Name	Address Range	Size (x16)	EALLOW	Description
NMICFG	0x7060	1	yes	NMI Configuration Register
NMIFLG	0x7061	1	yes	NMI Flag Register
NMIFLGCLR	0x7062	1	yes	NMI Flag Clear Register
NMIFLGFRC	0x7063	1	yes	NMI Flag Force Register
NMIWDCNT	0x7064	1	-	NMI Watchdog Counter Register
NMIWDPRD	0x7065	1	yes	NMI Watchdog Period Register

Figure 7 - Non Maskable Interrupt Registers

II. Task 2

Write a brief explanation of how the FIR filter works and why the filtering is done in the filter1() function.

FIR filters process discrete signals by removing unwanted frequencies. This is done by applying a set of filter coefficients to a sequence of recent input samples through convolution. The filter coefficients define the convolution weightings and can act to provide a low/high pass filter. In this case we have selected a low pass.

In filter1(), for each new input sample, it stores the sample in a circular buffer; it multiplies each coefficient by the corresponding sample in the buffer; it sums these products to compute the filtered output.

By encapsulating filtering in filter1(), the function can be called repeatedly within the ISR routine to process each new input sample independently. filter1() is efficient enough to process each sample in real-time. By applying convolution within the ISR, each new sample can be filtered and stored in the output buffer without delay. Finally, placing the filter in a dedicated function allows easy modification of filter logic if different coefficients or filter types are needed.

Ensure that the program runs correctly and that the filtered signal matches the expected result as shown in Figure 2, plot your results

- signal_FixedP variable showing input signal

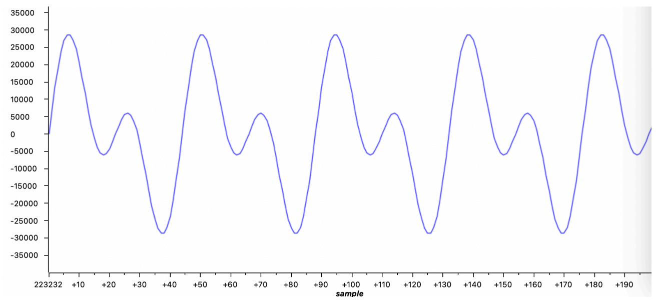


Figure 8 - Filter input signal (CCS)

- buffer1Fixed variable showing filtered signal

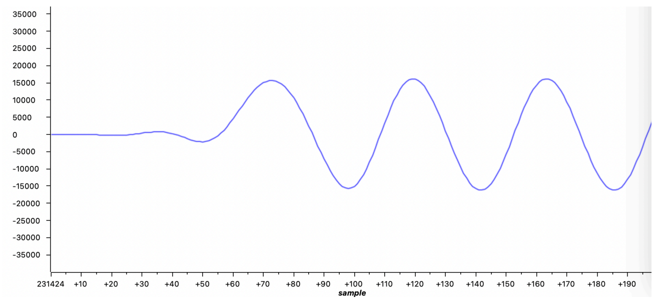


Figure 9 - Filtered signal (CCS)

As a note the filtered frequency matches that in figure 2 and becomes active once the filter reaches its transition band

III. Task 3

Produce a report that illustrates the implementation of the Goertzel algorithm

The Goertzel algorithm is a DSP technique that efficiently computes the power (magnitude) of specific target frequencies in a data sequence. This is effectively calculating specific terms from the Discrete Fourier Transform. I chose to use the optimized version of the Goertzel algorithm where you don't calculate real and imaginary components but instead the relative magnitude.

You precompute constants omega, cosine, sine and coeff used during processing

$$k = (\text{int})\left(0.5 + \frac{N * \text{target\_freq}}{\text{sample\_rate}}\right)$$

$$w = (2 * \pi / N) * k$$

$$cosine = \cos w$$

$$sine = \sin w$$

$$coeff = 2 * cosine$$

Figure 10 - Goertzel preprocessing

The per-sample processing variables each have a specific meaning. Q1 is the value of Q0 last time and Q2 is the value of Q0 two times ago.

$$Q_0 = coeff * Q_1 - Q_2 + sample$$

$$Q_2 = Q_1$$

$$Q_1 = Q_0$$

Figure 11 - Goertzel per sample processing

After the per-sample equations run N times, we measure the magnitudes using the optimized version to see if the tone is present or not.

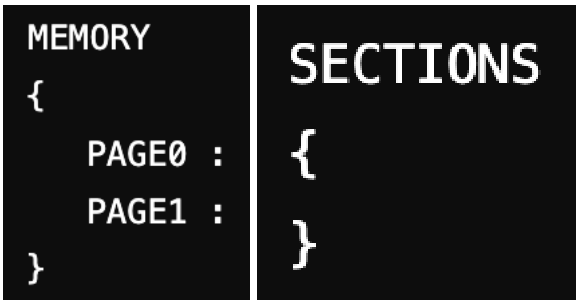
$$magnitude^2 = Q_1{}^2 + Q_2{}^2 - Q_1 * Q_2 * coeff$$

Figure 12 - Goertzel optimized magnitude

This is repeated for each of the target frequencies storing each of their magnitudes. These magnitudes can then be later compared in two groups to see the most prolific in each.

Produce a report that illustrates the implementation of the command file

The linker command file defines the memory layout and allocation of code, data, and other sections in an embedded system. I've specified the memory regions and the locations where the program code, initialized data, and other memory sections will reside.



```
MEMORY
{
    PAGE0 :
    PAGE1 :
}
```

```
SECTIONS
{
}
```

Figure 13 - Command file structure

PAGE 0 is reserved for program memory. This includes RAM and Flash regions intended to hold program instructions. The RAM Regions (RAMM0, RAMD0..) are used for temporary storage and runtime data. The Flash Sectors (FLASHA to FLASHG) are the main memory locations where the program code is stored. Each block has a specific range to allocate different code sections.

PAGE 1 is designated for data memory. We allocate memory for runtime data and data buffers, keeping them separate from the program code. The CPU RAM Regions (RAMM1, RAMD1, RAMLS5..) include general purpose storage areas. Shared RAM regions are used for inter-CPU communication when two processors are sharing memory.

The SECTIONS part maps specific code and data sections to the designated memory regions. The program sections .cinit, .pinit, and .text are allocated in the Flash sectors FLASHB, FLASHC, FLASHD, and FLASHE for storing program code and constant data. The data sections that store uninitialized data (.stack, .ebss, .esysmem) are assigned to RAM (RAMM1, RAMLS5, RAMGS0, RAMGS1..) for dynamic data storage.

To optimize stack memory allocation, the program needed a specific adjustment to the default command file. Particularly, the magnitude printing statements within the goertzel_multi function required the stack to be located in a continuous memory block around 0x001000 in size, so I assigned RAMGS2 for this purpose.

```
.stack                  : > RAMGS2          PAGE = 1
```

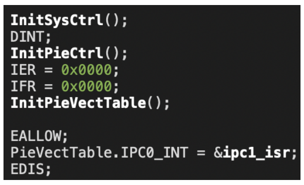
Figure 14 - Stack allocation modification

IV. Task 4

Documentation explaining the implementation, steps taken to modify the code, and how the communication between the two CPUs is established.

Each CPU is part of a separated CCS project and executed in parallel as part of a grouped debug session. CPU1 initializes its system, sends data to shared memory, and signals CPU2 to read the data. CPU2 initializes its system, waits for the data-ready signal from CPU1, reads and stores the data from shared memory

Setup:



```
InitSysCtrl();
DINT;
InitPieCtrl();
IER = 0x0000;
IFR = 0x0000;
InitPieVectTable();

EALLOW;
PieVectTable.IPC0_INT = &ipc1_isr;
EDIS;
```

Figure 15 - Interrupt and system setup

For both CPU1/2 the initialisation for IPC is identical. The code initializes system control, disables interrupts during setup, configures the interrupt controller, and assigns the interrupt vector to ipc1/2_isr method. This enables inter-processor communication between CPU1 and CPU2.

IpcRegs:

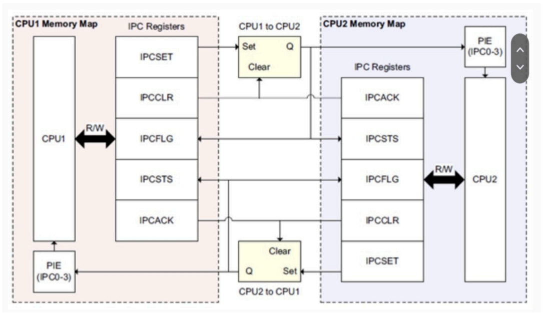


Figure 16 - IPC registers

For communication between CPU1 and CPU2 we require use of IPCregisters, which we can see status of with IPCSTS and operate on with IPCSET, IPCCLR, IPCACK. After initial setup, IPC17 is set to 1 from CPU2 to signal its readiness, which CPU1 waits for in a blocking loop. After completing data transfer to shared memory, CPU1 sets IPC1 to 1 telling CPU2 that data is ready. CPU2s IPC method catches this, reads the shared data and acknowledges IPC1.

Shared Memory:

```
CPU1TOCPU2RAM   : origin = 0x03FC00, length = 0x000400
```

Figure 17 - Shared memory

The main modification to the skeleton code involved shared memory. Both CPU1 and 2 have reference to a RAM section in their .cmd files called CPU1TOCPU2RAM, which originates at 0x03FC00. Data is transferred by setting a pointer to this memory address and safely writing and reading from this point, within the IPC reg sequence.

V. Task 5

The DMA is a feature of the TMS320F2837xD microcontroller, allowing for peripheral or memory-to-memory data transfer without direct CPU usage. This offloads the load from the CPU, freeing it up to operate elsewhere and improving overall efficiency.

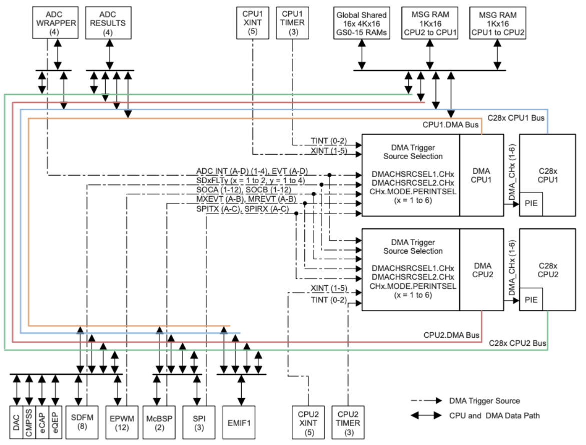
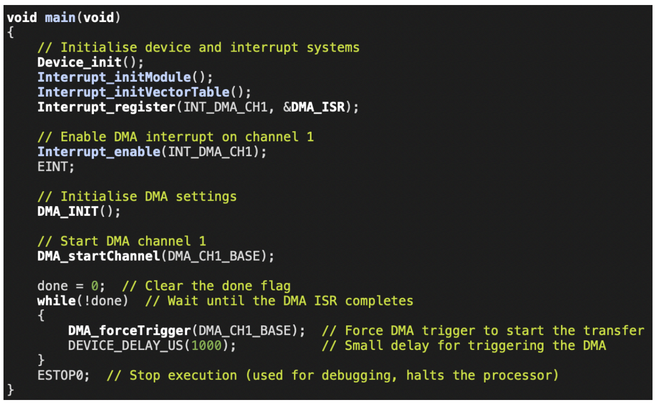


Figure 18 - DMA block diagram

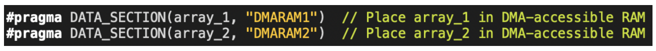
There is a specific DMA assigned to each of our dual core CPUs, providing independence. The DMA has up to 6 independent channels and transfer can be initiated by either hardware or software triggers. A transfer must have a defined source and destination (either in memory or a peripheral) and will often occur in a number bursts of a specific size. Once transfer is complete there is often an ISR setup



```
void main(void)
{
    // Initialise device and interrupt systems
    Device_init();
    Interrupt_initModule();
    Interrupt_initVectorTable();
    Interrupt_register(INT_DMA_CH1, &DMA_ISR);

    // Enable DMA interrupt on channel 1
    Interrupt_enable(INT_DMA_CH1);
    EINT;

    // Initialise DMA settings
    DMA_INIT();

    // Start DMA channel 1
    DMA_startChannel(DMA_CH1_BASE);

    done = 0;  // Clear the done flag
    while(!done)  // Wait until the DMA ISR completes
    {
        DMA_forceTrigger(DMA_CH1_BASE);  // Force DMA trigger to start the transfer
        DEVICE_DELAY_US(1000);           // Small delay for triggering the DMA
    }
    ESTOP0;  // Stop execution (used for debugging, halts the processor)
}
```

Figure 19 - DMA transfer main execution process

Practically my code implementation follows this process. Once the peripheral management and interrupts are set up I initialize the DMA. This includes defining the source/destination addresses (starts of their corresponding arrays) and defining the burst/transfer sizes. I then start transfer down channel one and trigger the channel repeatedly, separated by a small delay period. Once complete an ISR runs acknowledging the DMA interrupt from group 7 and checking for successful transfer.



```
#pragma DATA_SECTION(array_1, "DMARAM1")  // Place array_1 in DMA-accessible RAM
#pragma DATA_SECTION(array_2, "DMARAM2")  // Place array_2 in DMA-accessible RAM
```

Figure 20 - DMA RAM

For the DMA to have access to the memory spaces we place both arrays in DMA accessible RAM as highlighted above.

References

[1] "The Goertzel algorithm," *Embedded.com*, Feb. 2000. Available: https://www.embedded.com/the-goertzel-algorithm/

[2] "Multi-Core Motor Controlling and How It Is Shaping the Future," **Solo Motor Controllers**, Available: https://www.solomotorcontrollers.com/blog/multi-core-motor-controlling-and-how-it-is-shaping-the-future/?srsltid=AfmBOoqireAJUN_bi9OrfPt10OG3aFg8HHIRh--93WHbacW8uAnZN6D6.

[3] "Using CPU and CLA of TMS320F2837xD with dual core," Newbie Developer, Available: https://newbie-developer.tistory.com/353.

[4] "JTAG Hardware Debugger," JTAG Technologies. Available: https://www.jtag.com/jtag-hw-debugger/#:~:text=Debugging%3A%20JTAG%20is%20commonly%20used,of%20code%20step%20by%20step.

[5] "GitLab Continuous Integration Documentation," GitLab. Available: https://docs.gitlab.com/ee/ci/

[6] "CI/CD: What It Is and Why It Matters," HubSpot Blog. Available: https://blog.hubspot.com/website/cicd

[7] Texas Instruments. (2022). *TMS320F2837xD Dual-Core Microcontrollers Technical Reference Manual* (SPRuh18i). Retrieved from https://www.ti.co