Combat Vehicles Research and Development Establishment
Defence Research and Development Organization

Research Internship Project Report
On

Miniaturized Wheel Dynamics Simulator – Model based design using TMS320 32-bit microcontroller

Submitted By

Ayush Gupta
Tharun Kumar J
Aiswarya S
B.Tech, (IV Year)
Department of Electronic and Communication Engineering
SRM Institute of Science and Technology
S.R.M Nagar, Kattankulathur-603203

Under the Guidance of

Ms. S Mohana Sundari
Scientist 'F'

Combat Vehicles Research & Development Estt. (CVRDE)
Defence Research and Development Organization (DRDO)
Ministry of Defence, Govt. of India
Avadi, Chennai - 600054

ACKNOWLEDGEMENTS

Combat Vehicles Research and Development Establishment [CVRDE] is one of the premier establishment labs of the Defence Research and Development Organization [DRDO] under the Ministry of Defence, Govt. of India. This establishment is committed to the development of the manned and unmanned variants of Armoured Fighting Vehicles (AFVs) and combat aircraft line-replaceable units and we feel extremely grateful for being granted such a great opportunity to pursue an Internship at CVRDE-DRDO, Avadi, Chennai.

We are highly obliged to **Shri V. Balamurugan, Director-CVRDE, Avadi** for allowing us to associate with this esteemed establishment as Interns to carry out Project R&D work in 'Gun Control Systems [GCS]' Division during the tenure 08th June to 29th July, 2022.

Further, we extend our heartfelt gratitude to the Head of GCS Division, **Dr. V Ramesh Kumar,** Scientist 'F', CVRDE for granting us permission to GCS, and allowing us to undertake our project without hassle.

Most importantly, we express our sincere thanks to **Ms. S Mohana Sundari, Scientist 'F'** and **S. Raja Vidhya Saharan, Scientist 'D'** for their unflagging guidance throughout the progress of our internship project as well as valuable contribution in the preparation and compilation of the text.

We are thankful to the Scientists and other staff at CVRDE, DRDO who made the things go easy by paying their time and effort in the successful completion of our internship.

(Ayush Gupta)
(Tharun Kumar J)
(Aiswarya S)
SRM Institute of Science and Technology
S.R.M Nagar, Kattankulathur-603203

SYNOPSIS

In the accomplishment of our internship R&D work, we are submitting a project report entitled "Miniaturized Wheel Dynamics Simulator -- Model based design using TMS320 32-bit microcontroller". Subject to the limitation of time, efforts and resources every possible attempt has been made to study the problems deeply.

The whole project is divided into various sub-sections based on different work learned and performed at CVRDE-DRDO.

The design featured in this report presents a Model-Based Development (MBD) methodology as a promising approach for rapid, reliant and reprogrammable development of Embedded Hardware applications, including those involving digital controllers. Based on existing hardware and software application-oriented tools by Texas Instruments, a new modeling technique has been implemented to move from a conventional design workflow to an MBD one by using MathWorks MATLAB software. Looking from a practical standpoint of the proposed design, the model is deployed on a TMS320F28335 MCU Control card plugged onto its corresponding TI Docking station which is interfaced directly with a 5V logic interface board which in turn is connected directly with the landing gear controller [LGC] and the aircraft telemetry display to view the wheel speed data gathered from the model computations.

The project serves as a means to simulate and test complex embedded hardware applications using re-programmability offered by microcontrollers designed for such applications as opposed to testing these applications using much bigger systems for the same purpose. Once the functionality as performed by the MCU is witnessed to be what was desired, the model can then be deployed onto the flash memory of the MCU connected to the respective hardware to serve as an embedded solution.

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Objective of Study

The study presents the model for a Miniature Wheel Dynamics Brake Simulator using Texas Instruments' TI C2000 Microcontroller and MATLAB-Simulink Based Embedded Coder Plugin. It is expected to contribute and promote further work via the means of Simulink-Embedded Coder serving as a test use-case of industry relevant problem statements pertaining to embedded systems while keeping the following objectives in mind:

- This research seeks to construct a test bench device for the performance of an Aircraft Wheels and their dynamics in a real-world setting.
- To comprehend and make the appropriate modifications for activating the registers present in TMS320F28335 in order to create the necessary components for the Final Miniaturized Dynamic Braking Model.
- The purpose of the project is to replicate the wheel braking system in real-time by applying pulse width modulation of frequency, taking the voltage and the initial velocity as input which can be accomplished by designing a model that takes into account all of the actual factors or forces that have an impact on the aircraft, such as kinematics, drag, thrust, axes, etc. Once this model has been designed, it is then deployed onto the microcontroller, and testing is performed by employing a variable input velocity through a serial communication interface transmitting it.
- This can simulate the model's operation at a variety of speeds. The finished product of the project is an automatic braking system that can be employed. It is then upscaled to a 5V logic and then combined with a Landing Gear Controller.
- To read, collect, and evaluate the data that was obtained, the model was integrated into a real workspace.

1.2 Background

When we take a step back and consider how far technology has come over the course of history, we can see that we no longer perform even the most basic of jobs in the same way that we once did. Everything advances through time, and this project would make the simulation and testing of the wheel-braking system far less difficult than it would otherwise be. There is no longer a requirement to use cumbersome motors and appliances in order to recreate the conditions of the braking environment. This project demonstrates a method for modeling a real-world Miniaturized Wheel Brake Dynamics Simulator by making use of a MCU.

Since the application includes testing on a Real Landing Gear Controller, the Texas Instruments TMS320 series control card was taken which contains the TI c2000 series digital signal controller or DSC on it. The control card is attached onto its compatible TI docking station for expanding the MCU pinouts thus making the GPIOs and peripherals easily accessible for interfacing.

The MCU acts as the mind of the model controlling every single aspect both hardware and software as proposed in the design. The functionality begins by sending an initial velocity through the SCI (Serial Communication Interface), which is then added with the positive factors and negative factors that act on it. For instance, the variable voltage that was given as input using the ADC would then be converted to get a corresponding proportional Brake pressure that acts as a negative factor on the entire model. In this way, the aim of achieving the objectives mentioned in previous section were accomplished. The value obtained for the Final Velocity is used to indicate the effect that the brakes have had on the whole system. This is accomplished by converting the value into the proper frequency for each wheel and putting it into an ePWM. When it comes to managing many of the power electrical systems that can be found in commercial and industrial machinery, the enhanced pulse width modulator (ePWM) peripheral is an essential component. This allows for the frequency to be varied in real-time according to the amount of brake pressure that is being applied to the system. The model was designed and debugged with the use of a hardware set-up that included a TMS320F28335, code composer studio, and MATLAB R2022a.

MATLAB Embedded Coder was chosen as the Plugin of Choice for this project because it creates C and C++ code that is understandable, concise, and quick for embedded processors that are utilized in mass manufacturing. It expands the capabilities of MATLAB Coder and Simulink Coder by adding sophisticated optimizations that allow for more accurate control of the functions, files, and data that are generated. The efficiency of the code is improved, and interaction with legacy code, data types, and calibration parameters is made easier as a result of these enhancements. In order to see and debug the code that was created from the model, Code Composer Studio was used. Code Composer Studio is an integrated development environment that enables users to construct applications for Texas Instruments' embedded processors. Code Composer Studio provides access to a wide variety of functions, many of which are useful for writing code for a Texas Instruments Controller.

2. TMS320F28335 MCU

2.1 Architecture

The Texas Instruments C2000™ 32-bit microcontrollers are optimized for processing, sensing, and actuation to improve closed-loop performance in real-time control applications.

The microcontroller that was chosen for this project, that is, the TMS320F28335 is a member of the TMS320C28x/Delfino™ DSC/MCU generation which are highly integrated, high-performance solutions for demanding control applications such as industrial motor drives; electrical vehicles and transportation; sensing and signal processing. The TMS320F2833x Digital Signal Controller is capable of executing six basic operations in a single instruction cycle, and therefore the architecture of the device must reflect this feature in some way.

The TMS320F2833x Block Diagram can be divided into the following functional units:

- Internal and external Bus System
- Central Processing Unit (CPU)
- Internal Memory Sections
- Control Peripherals
- Communication Channels
- Direct Memory Access Controller (DMA)
- Interrupt Management Unit (PIE) and Core Time Unit
- Real - Time Emulation Interface

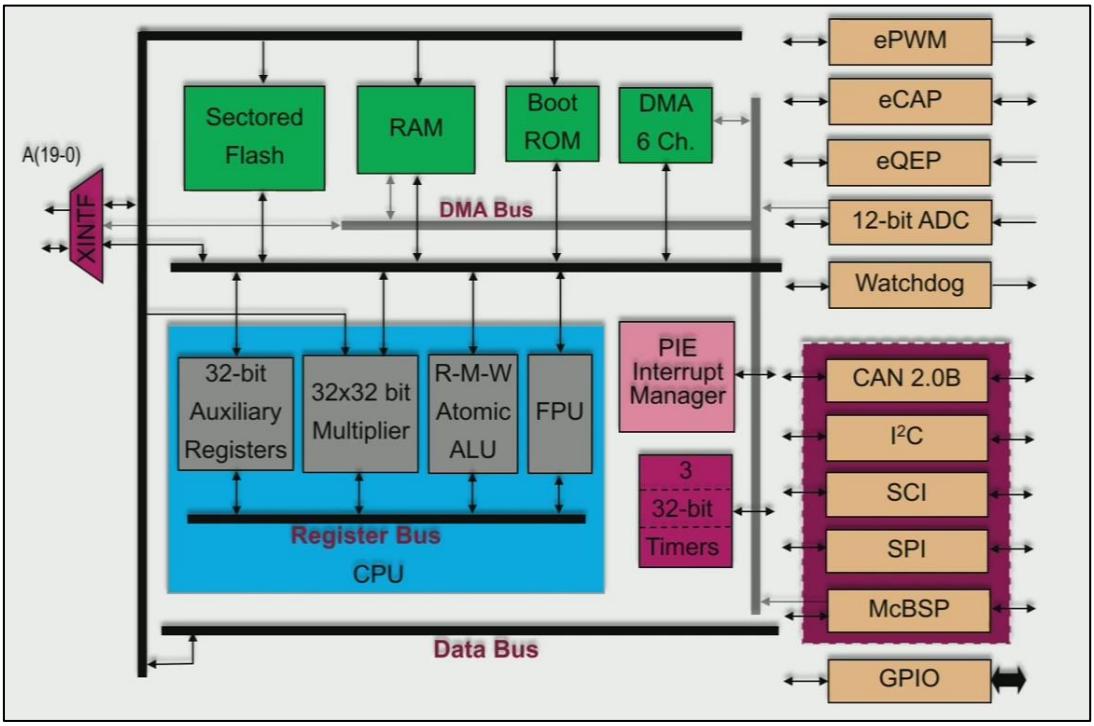


Fig. 1. F2833x MCU Block Diagram

2.2 TMS320C28x Core

The TMS320C28x is a 32-bit fixed point DSP that specializes in high performance control applications such as, robotics, industrial automation, mass storage devices, lighting, optical networking, power supplies, and other control applications needing a single processor to solve a high-performance application. The C28x-CPU is able to execute most of the instructions to perform register-to-register operations and a range of instructions that are commonly used by microcontrollers, for instance, byte packing and unpacking and bit manipulation in a single cycle. The architecture is also supported by powerful addressing modes, which allow the compiler as well as the assembly programmer to generate compact code that corresponds almost one-to-one with the C code. The C28x is as efficient in typical math tasks for Digital Signal Processing as it is in the system control tasks that are typically handled by microcontroller devices. This efficiency removes the need for a second processor in many systems. The C28x architecture can be divided into 3 functional blocks:

- CPU and busing
- Memory
- Peripherals

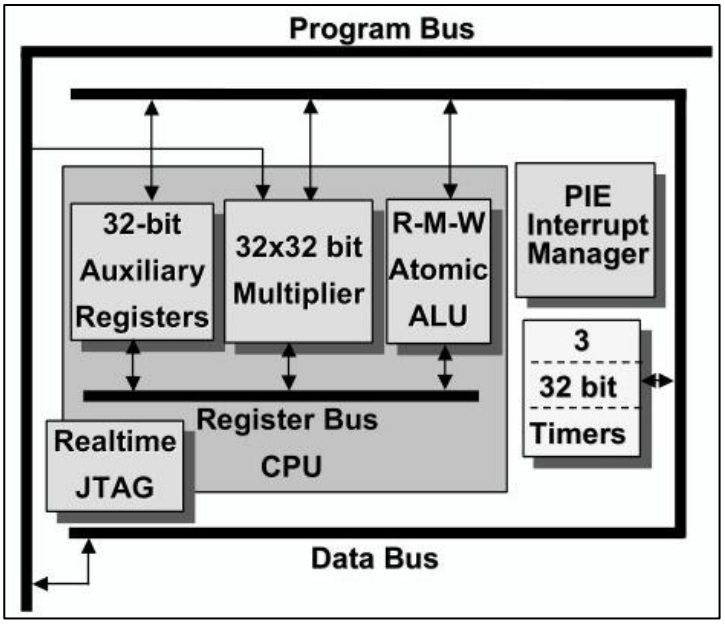


Fig. 2. C28x CPU

The C28x design supports an efficient C engine with hardware that allows the C compiler to generate compact code. Multiple busses and an internal register bus allow an efficient and flexible way to operate on the data. The architecture is also supported by powerful addressing modes, which allow the compiler as well as the assembly programmer to generate compact code that is almost one to one corresponded to the C code.

2.2.1 Multiplier, ALU and Shifters

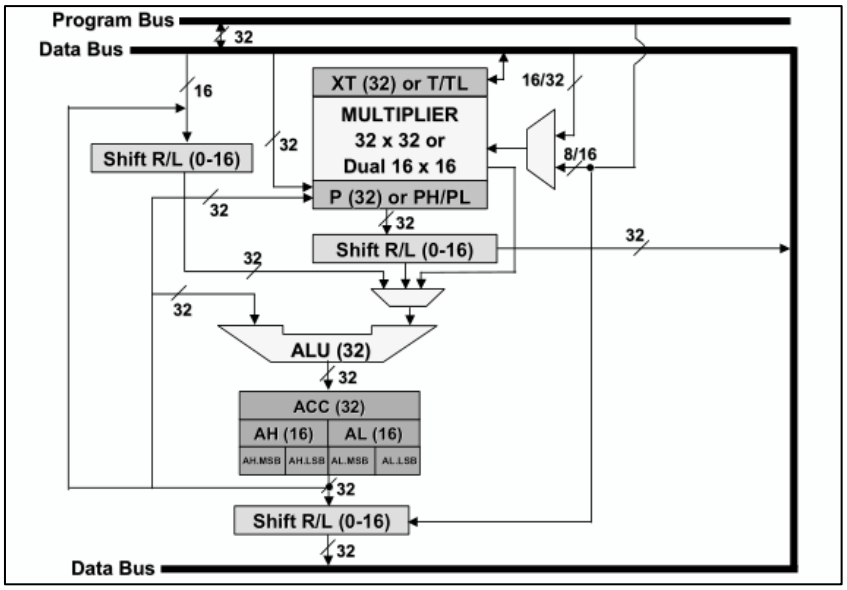


Fig. 3. C28x Multiplier and ALU/Shifters

The 32 x 32-bit MAC capabilities of the C28x and its 64-bit processing capabilities, enable the C28x to efficiently handle higher numerical resolution problems that would otherwise demand a more expensive floating-point processor solution. Along with this is the capability to perform two 16 x 16-bit multiply accumulate instructions simultaneously or Dual MACs (DMAC).

2.2.2 TMS320C28x Internal Bussing

As with many DSP type devices, multiple busses are used to move data between the memories and peripherals and the CPU. The C28x memory bus architecture contains:

- A program read bus (22-bit address line and 32-bit data line)
- A data read bus (32-bit address line and 32-bit data line)
- A data write bus (32-bit address line and 32-bit data line)

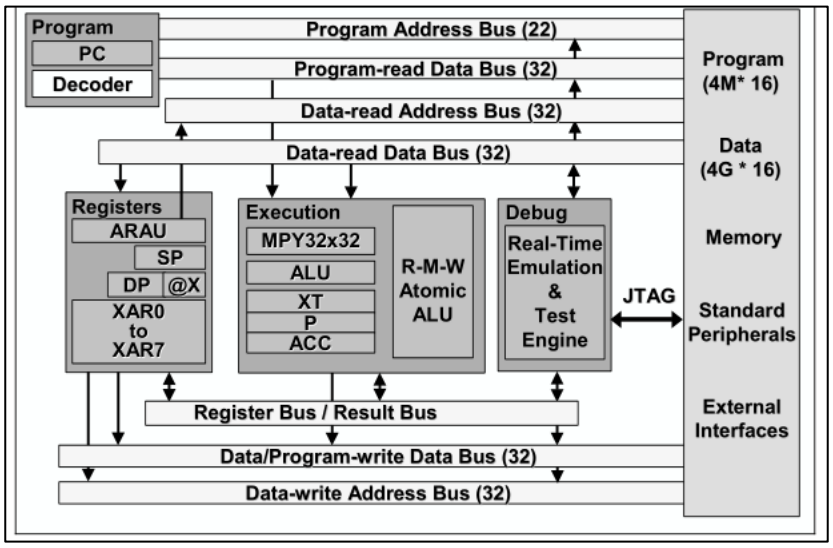


Fig. 4. C28x Multiplier and ALU/Shifters

- The 32-bit-wide data busses enable single cycle 32-bit operations.
- This multiple bus architecture, known as a Harvard Bus Architecture enables the C28x to fetch an instruction, read a data value and write a data value in a single cycle.
- All peripherals and memories are attached to the memory bus and will prioritize memory accesses.

2.2.3 DMA-Direct Memory Access Controller

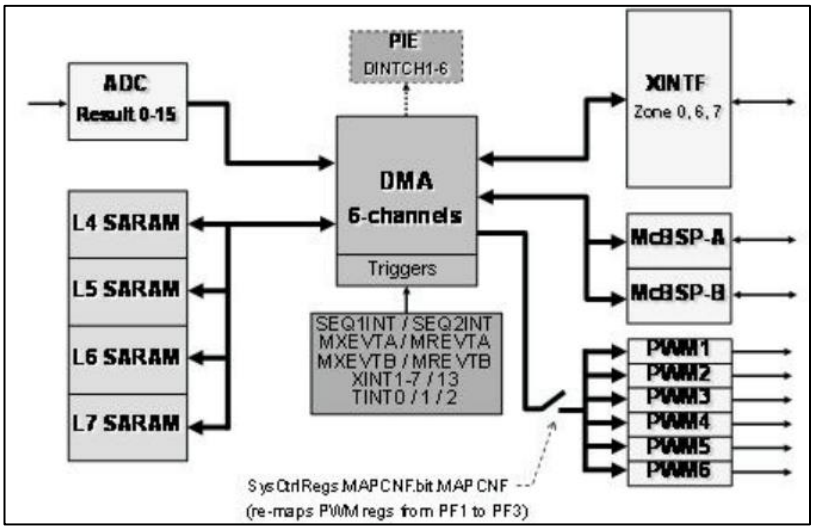


Fig. 5. F2833x DMA Interface

A Direct Memory Access Controller has been introduced in the F2833x family. A DMA unit allows a data transfer from a source to a destination unit without the need of an interaction of the CPU. The strength of a digital signal controller (DSC) is measured not only in processor speed, but also in total system capability. As a part of the equation, whenever the CPU bandwidth for a given function can be reduced, the greater the system capability will be. Very often applications spend a significant amount of their bandwidth moving data, whether it is from of-chip memory to on-chip memory or from a peripheral such as an analog-to-digital converter (ADC) to RAM, or even from one peripheral to another. Furthermore, there are times when this data comes in a format that is not compatible with the optimum processing power of the CPU. The DMA module has the ability to free up CPU bandwidth and rearrange the data into a more streamlined processing pattern. The DMA module is an event-based machine, meaning it requires a peripheral interrupt trigger to start a DMA transfer, such as:

- Analogue to Digital Converter Sequencer 1 (SEQ1INT) or Sequencer 2 (SEQ2INT)
- Multichannel Buffered Serial Port A and B (McBSP-A, McBSP-B) transmit/receive
- External Interrupt Input Signals XINT1-7 and XINT13
- CPU Timers 0, 1 and 2
- Pulse Width Module (PWM) signals and Software Interrupts

2.2.4 Pipelining

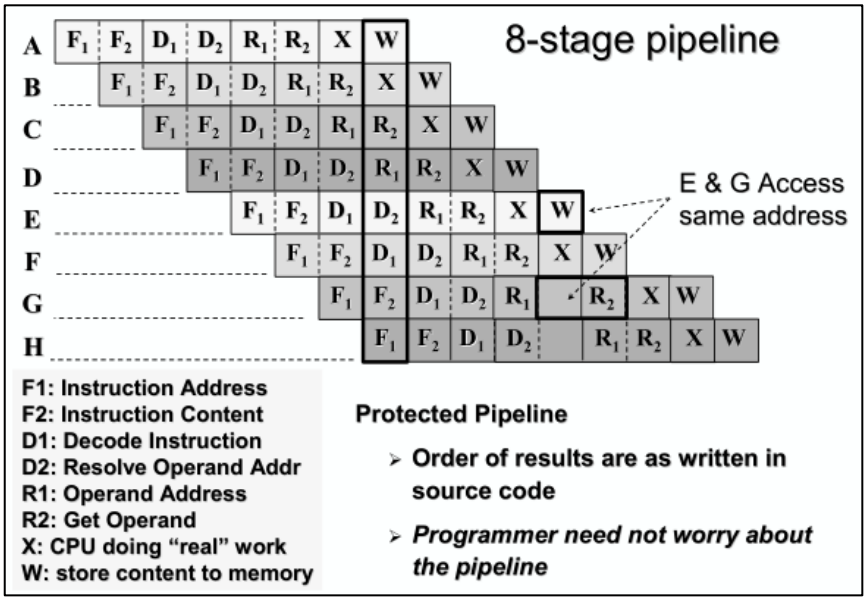


Fig. 6. C28x Pipeline

Like almost all today's microprocessors that operate in speed regions above 50 MHz the F2833x also uses a pipeline technique to maximize the code throughput. The F2833x features an 8-stage protected pipeline. The adjective "protected" means that the pipeline unit itself automatically prevents a "write to" and a "read from" from occurring out of sequence at the same location.

The C28x uses a special 8-stage protected pipeline to maximize the throughput. This protected pipeline prevents a write to and a read from the same location from occurring out of order. It also enables the C28x to execute at high speeds without resorting to expensive high-speed memories. Special branch-look-ahead hardware minimizes the latency for conditional discontinuities. Special store conditional operations further improve performance. Each instruction passes through 8 stages until fnal completion. Once the pipeline is flled with instructions, one instruction is executed per clock cycle. For a 150MHz device, this equates to 6.67ns per instruction. The stages are:

- F1: Generate Instruction Address at program bus address lines.
- F2: Read the instruction from program bus data lines.
- D1: Decode Instruction
- D2: Calculate Address information for operand(s) of the instruction
- R1: Load operand(s) address to data and/or program bus address lines
- R2: Read Operand
- X: Execute the instruction
- W: Write back result to data memory

2.2.5 Memory Mapping

The memory space on the C28x is divided into program and data space. There are several different types of memory available that can be used as both program or data space. They include the flash memory, single access RAM (SARAM), expanded SARAM, and Boot ROM which is factory programmed with boot software routines and trigonometric lookup tables used in math-based algorithms. Memory space width is always 16 bits.

The C28x CPU contains no memory, but can access memory both on and off the chip. The C28x uses 32-bit data addresses and 22-bit program addresses. This allows for a total address reach of 4G words (1 word = 16 bits) in data space and 4M words in program space. Memory blocks on all C28x designs are uniformly mapped to both program and data space. This memory map shows the different blocks of memory available to the program and data space.

To load information into FLASH and OTP, a dedicated download program is needed, which is also part of the Texas Instruments Code Composer Studio integrated design environment. Volatile Memory is split into 10 areas called M0, M1 and L0 - L7 that can be used both as code memory and data memory. PF0, PF1 and PF2 are Peripheral Frames that cover control and status registers of all peripheral units ("Memory Mapped Registers").

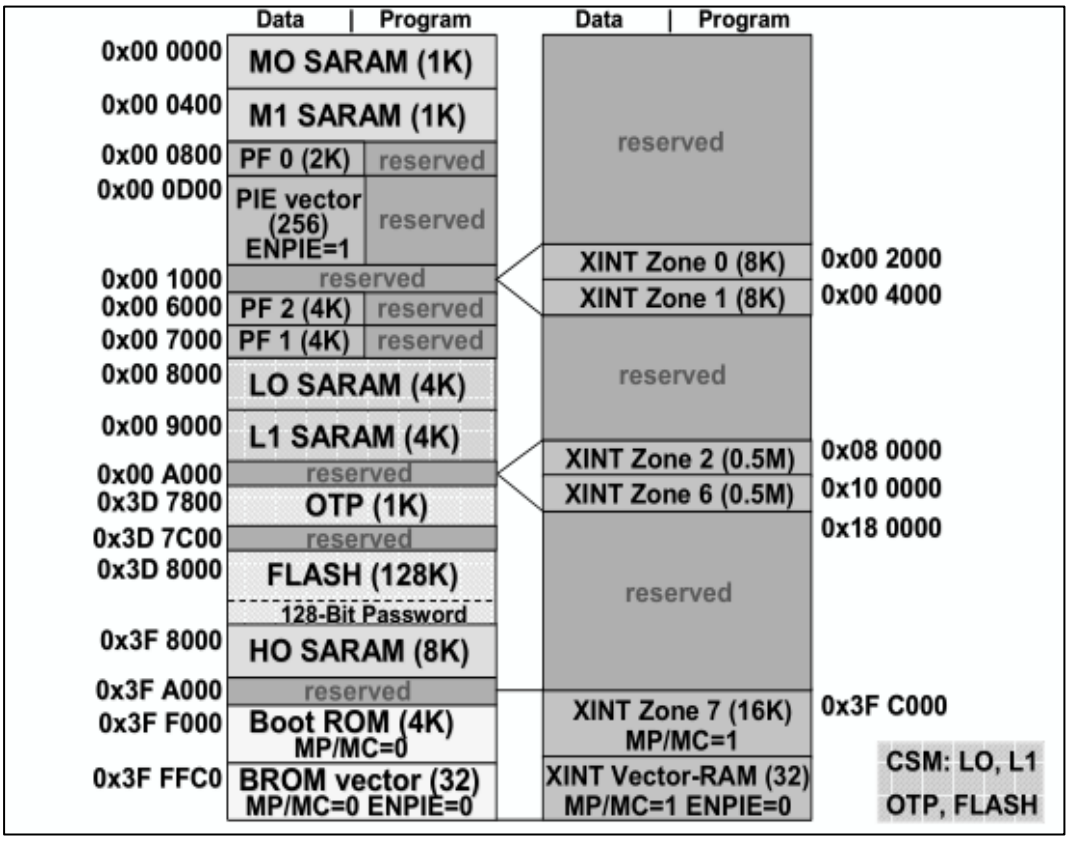


Fig. 7. TMS320F2833x Memory Map

2.2.6 Code Security Module

There is an internal security module available in all F2833x family members. It is based on a 128-bit password that is written by the software developer into the last 8 memory spaces of the internal FLASH (0x3F 7FF8 to 0x3F 7FFF). Once a pattern is written into this area, all further accesses to any of the memory areas covered by this Code Security Module (CSM) are denied, as long as the user does not write an identical pattern into password registers of frame PF0.

2.2.7 Interrupt Response

A key feature of a control system is its ability to respond to asynchronous external hardware events as quickly as possible. The F2833x combines such fast interrupt responses with automatic context save of critical registers, which allows for the service of many asynchronous events with minimal latency. F2833x devices implement a zero-cycle penalty to save and restore the 14 registers during an interrupt. This feature helps to reduce the interrupt service routine overheads.

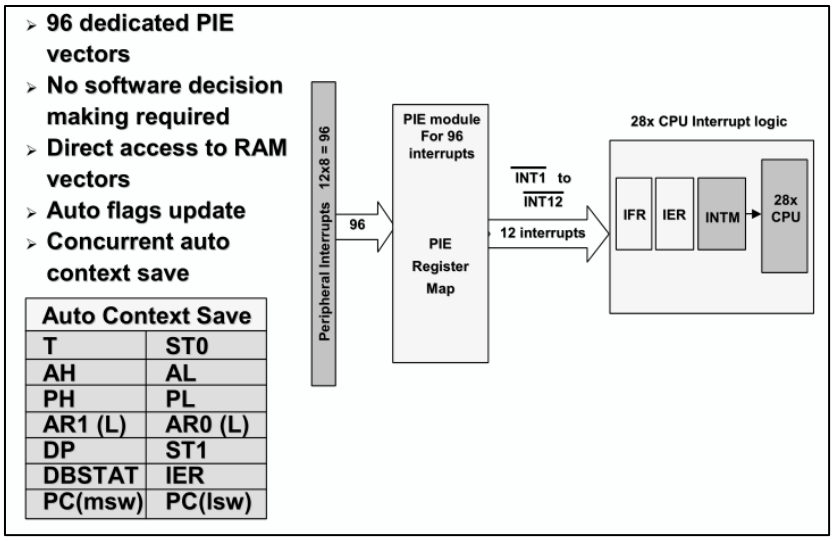


Fig. 9. F2833x Fast Interrupt Response Manager

2.2.8 Operating Modes

The F2833x silicon is able to operate in three different modes:

- C28x - Mode: takes advantage of all 32-bit features of the F2833x device
- C24x - Mode: source code compatibility to the 16-bit family members
- Test - Mode: intermediate operating mode, test purposes only.

After RESET, the device is set into test mode. To take advantage of the full computing power of an F2833x device, set the control flag “OBJMODE” to 1 which will switch the device into F2833x native mode.

2.3 Peripherals & Memory Access

2.3.1 Peripherals

The F2833x comes with many built in peripherals optimized to support control applications. The table shown below lists all the available peripherals that can be programmed in the target.

Peripheral	Lit. No.	Type(1)
System Control and Interrupts	SPRUI07	–
Boot ROM		–
Analog-to-Digital Converter (ADC)		2
Analog-to-Digital Converter Wrapper		0
Multichannel Buffered Serial Port (McBSP)		1
Serial Communications Interface (SCI)		0
Serial Peripheral Interface (SPI)		0
Enhanced Controller Area Network (eCAN)		0
Enhanced Quadrature Encoder Pulse (eQEP)		0
Enhanced Pulse Width Modulator Module (ePWM)		0
Enhanced Capture Module (eCAP)		0
Inter-Integrated Circuit (I2C)		0
High-Resolution Pulse-Width Modulator (HRPWM)		0
Direct Memory Access (DMA)		0
External Interface (XINTF)		1
Floating-Point Unit (FPU)	SPRUHS1	–

NOTE: Chapter 4: Conventional Programming discusses a detailed overview of the more common and widely used peripherals and how to go on about programming them.

2.3.2 Data Memory Access

Two basic methods are available to access data memory locations:

1. Direct Addressing Mode: It generates the 22-bit address for a memory access from two sources - a 16-bit register "Data Page (DP)" for the highest 16 bits plus another 6 bits taken from the instruction.
   - Advantage: Once DP is set, it can access any location of the selected page, in any order.
   - Disadvantage: If the code needs to access another page, DP must be changed first.
2. Indirect Addressing Mode: It uses one of eight 32-bit XARn registers to hold the 32-bit address of the operand.
   - Advantage: Using the ARAU (auxiliary register arithmetic unit) pointer arithmetic is available in same cycle in which access to a data memory location is made.
   - Disadvantage: A random access to data memory needs the pointer register to be setup with a new value.

2.4 Control Card and Docking Station

2.4.1 Control Card

TMDSCNCD28335 is a DIMM100 control card-based evaluation and development tool for the C2000™ Delfino™ F2833x series. Control cards are ideal to use for initial evaluation and system prototyping. Control cards are complete board-level modules that utilize one of two standard form factors (100-pin DIMM or 180-pin HSEC) to provide a low-profile single-board controller solution. For first evaluation, control cards are typically purchased bundled with a baseboard, a Peripheral Explorer Kit, or an application kit. The TMDSCNCD28335 control card requires a baseboard to function.

Fig. 10. TMS320F28335 Control Card

2.4.2 Docking Station

TMDSDOCK28335 is a DIMM100 control card-based evaluation and development tool for the C2000™ Delfino™ F2833x series of microcontroller products. The docking station provides power to the control card and has a bread-board area for prototyping. Access to the key device signals are available using a series of header pins.

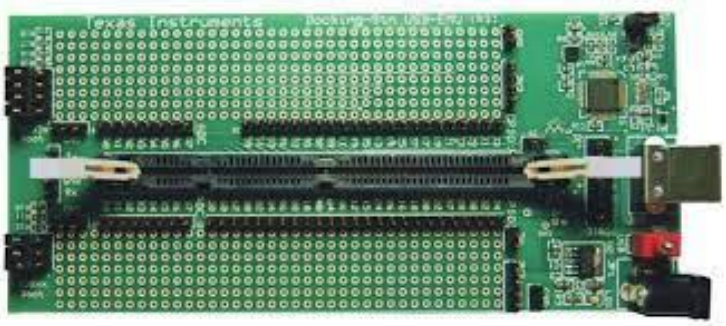


Fig. 11. TMDSDOCK28335 Docking Station

Fig. 12. Control Card Plugged Onto the Docking Station

3. DEVELOPMENT ENVIRONMENT

3.1 Code Composer Studio [CCS] IDE

Code Composer Studio is an Integrated Development Environment used for programming Texas Instruments [TI] based Embedded microcontrollers. The software includes an optimizing C and C++ compiler, source code editor, project build environment, debugger, profiler and many other features. In an effort to standardize the software development process, TI uses the Common Object File Format (COFF) that has several features which make it a powerful software development system. Each file of code, called a module, may be written independently, including the specification of all resources necessary for module operation. Modules can be written in CCS editor or any text editor capable of providing a simple ASCII file. The expected extension of a source file is .ASM for assembly and .C for C programs.

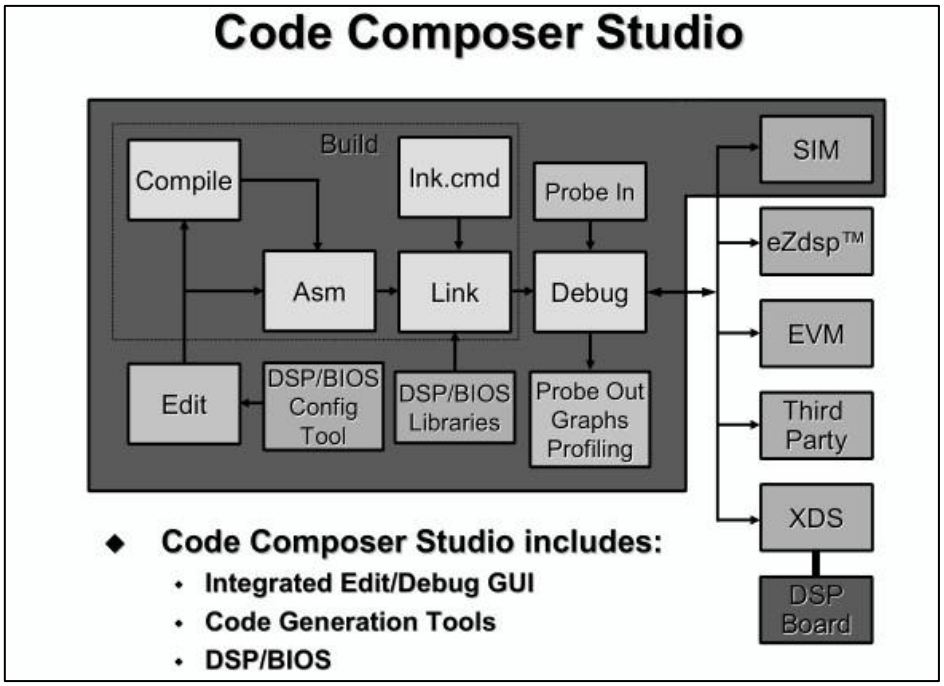


Fig. 1. CCS Design Flow

Code Composer Studio additionally has tools to connect file input and output, as well as built-in graph displays for output. Other features can be added using the plug-ins capability Numerous modules are joined to form a complete program by using the linker. The linker efficiently allocates the resources available on the device to each module in the system. The linker uses a command (.CMD) file to identify the memory resources and placement of where the various sections within each module are to go. Outputs of the linking process includes the linked object file (.OUT), which runs on the DSP, and can include a .MAP file which identifies where each linked section is located. The high level of modularity and portability resulting from this system simplifies the processes of verification, debug and maintenance.

The concept of COFF (Common Object File Format) tools is to allow modular development of software independent of hardware concerns. An individual assembly language file is written to perform a single task and may be linked with several other tasks to achieve a more complex total system. Writing code in modular form permits code to be developed by several people working in parallel so the development cycle is shortened. Debugging and upgrading code is faster, since components of the system, rather than the entire system, is being operated upon.

Also, new systems may be developed more rapidly if previously developed modules can be used in them. Code developed independently of hardware concerns increases the benefits of modularity by allowing the programmer to focus on the code and not waste time managing memory and moving code as other code components grow or shrink. A linker is invoked to allocate systems hardware to the modules desired to build a system. Changes in any or all modules, when re-linked, create a new hardware allocation, avoiding the possibility of memory resource conflicts.

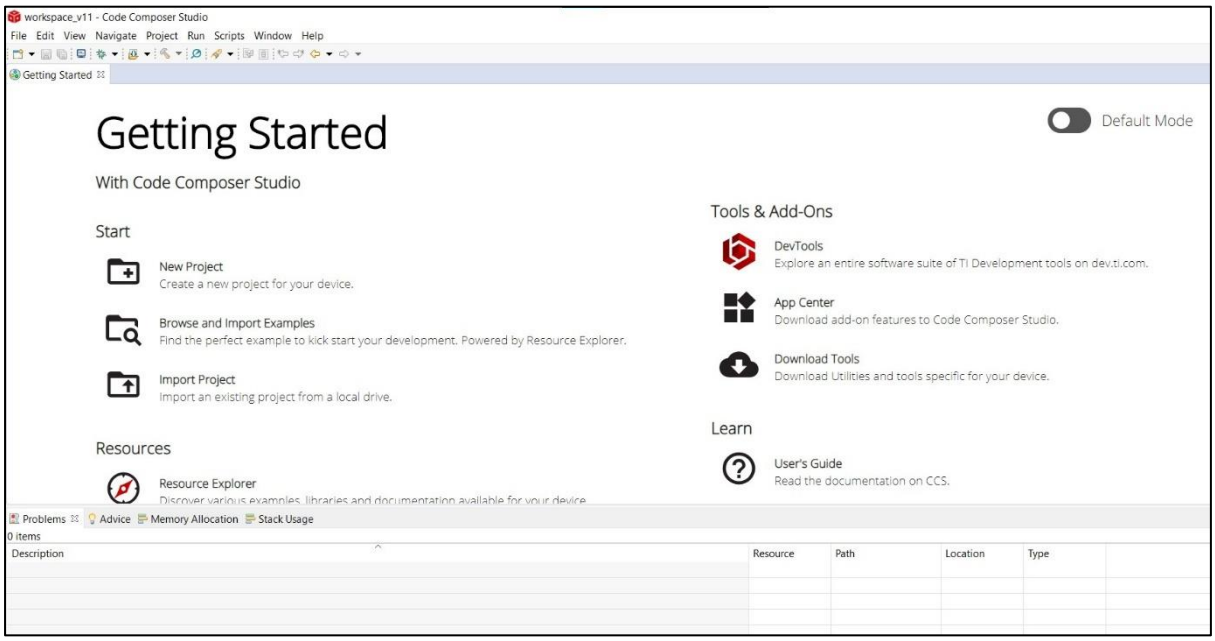


Fig. 2. Code Composer Studio User Interface

Code Composer works with a project paradigm. Essentially, within CCS a project is created for each executable program. Projects store all the information required to build the executable. For example, it lists things like: the source files, the header files, the target system's memory-map, and program build options.

Project options direct the code generation tools (i.e. compiler, assembler, linker) to create code according to the system's needs. When a new project is created, CCS creates build options – called Configurations: which further expand to Build & Debug source code on the target.

3.2 Simulink-Embedded Coder

3.2.1 Introduction

Mathworks Inc., along with various micro-controller manufacturers has developed a Simulink toolbox in order to generate processor specific embedded code. Matlab provides processor specific input output block set's which along with the blocks from Simulink can be used to generate code. Simulink allows testing the built model through regular simulation before it could be deployed on to the hardware. The model can be built and deployed to hardware through processor specific software like Code Composer Studio, Cross Core Embedded Studio, Arduino IDE etc. The model can also be deployed directly from Simulink on to the hardware. Both the procedures require that the processor specific integrated development tool be installed. Embedded Coder enables one to incorporate generated code into the code execution environment. With MATLAB, the code generated from Embedded Coder executes using the same execution framework as provided by MATLAB Coder. With Simulink, Embedded Coder significantly extends the real-time execution framework provided by Simulink Coder. By default, the code can be executed with or without a real-time operating system (RTOS) and in single-tasking, multitasking, multicore, or asynchronous mode.

3.2.1 Code Generation & Development

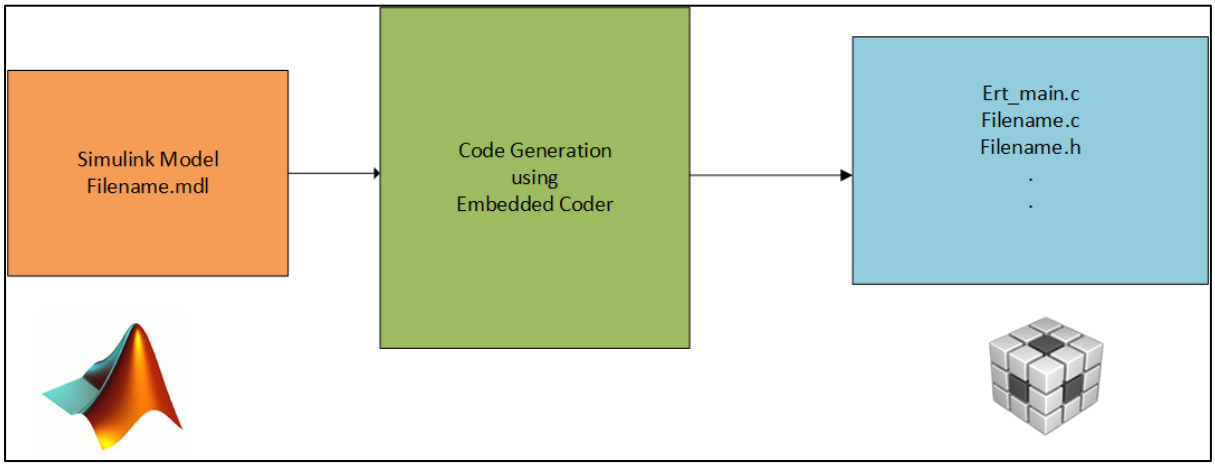


Fig. 3. Block Diagram of Code Generation

Discussed below is a brief introduction to setup and run embedded coder on TI-F28335 processor. The user can load the program on to the processor in two ways- one is through MATLAB directly and the other way is by using the generated files in Code Composer Studio, the official Texas Instruments IDE for C2000 class processors. If a user builds a model in Simulink for testing. Once, testing is done, the same model can be used to generate code by placing the required processor specific blocks and making necessary changes in the

configuration menu. Embedded Coder provides seamless integration capabilities over multiple processor's not only from the same manufacturer but from different manufacturers as well. This can be achieved by just replacing the processor specific block sets. Hence, the designer does not have to worry about rewriting the whole code again to ensure compatibility.

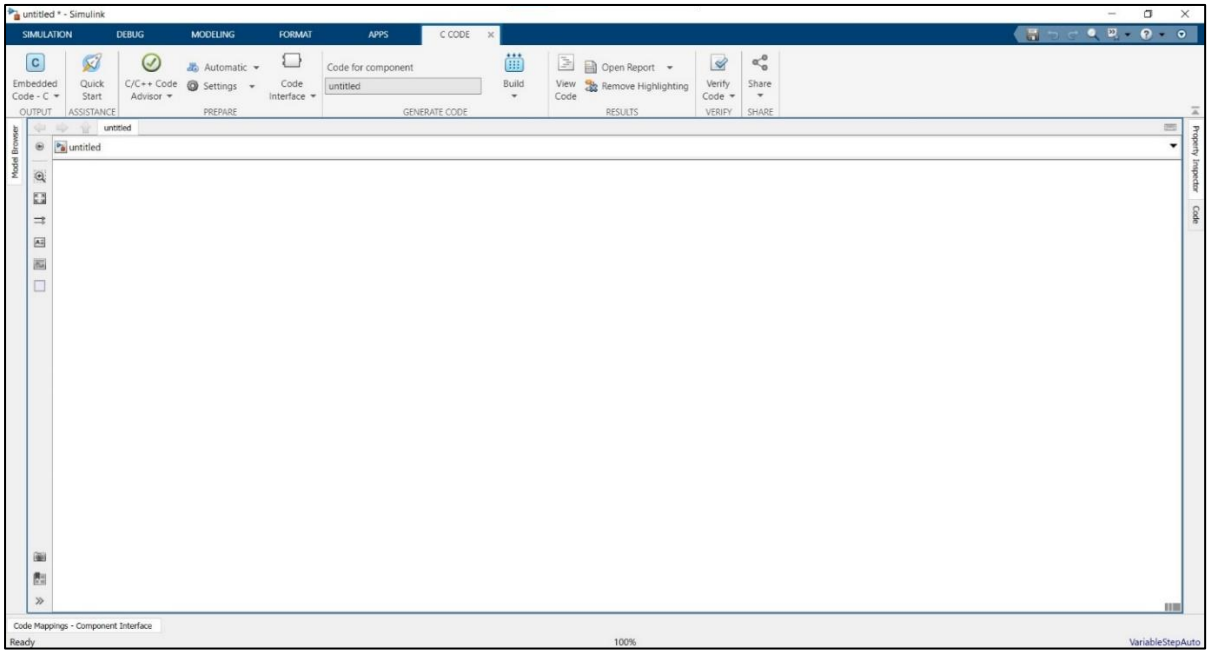


Fig. 4. Simulink-Embedded Coder User Interface

Embedded Coder not only makes the job of an embedded software engineer easy, but also that of a hardware engineer, with a freedom to change the processor at any stage of the project as is discussed in the section ahead. These points make Embedded Coder a viable tool to be used for research and development in academic field as well as in industries. Processor specific integrated development tool needs to be selected in the configuration menu. Once that is done Simulink generates a specific set of files and folders. The description of the same is given below.

- A main file to initialize the system and check for errors.
- A c file to initialize the board peripherals along with interrupts.
- A c file containing the algorithm designed in the Simulink model.
- A c file containing the constants and variables used in the Simulink model.
- A c file containing the data types of the various constants and variables used in the model.
- A c file to initialize the hardware as per the requirement of the program. This c file is supported by additional c files and header files to initialize the hardware.
- Additional header files are generated to support the code generation are generated or pulled in from the Texas Instruments Control Suite.

3.3 Embedded Coder-TI c2000 Support Package

MathWorks provides additional toolbox along for Texas Instruments microcontrollers. These blocks can be used in Simulink like the regular blocks. These blocks represent the functionality of the specified modules. Hence, a knowledge of the configurations of the processor under use is essential. Sophisticated algorithms can be built using available Simulink blocks and the TI Embedded Coder blocks can be used to generate the codes.

The TI C2000 Embedded Coder tool set provided by MathWorks is shown in fig. 5 below. As we can see from the figure, the support package consists of processor specific modules, optimization tools for fixed point operations, RTDX instrumentation for real time operation, Target Communication and scheduling blocks. The processor specific modules include all the peripheral blocks available in that particular module.

The fig. 5 shows the modules available for C2833x family processors. These block sets were used and configured which come as a part of the TI c2000 Support Package that can be directly installed and added to the Simulink-Embedded Coder Library Browser.

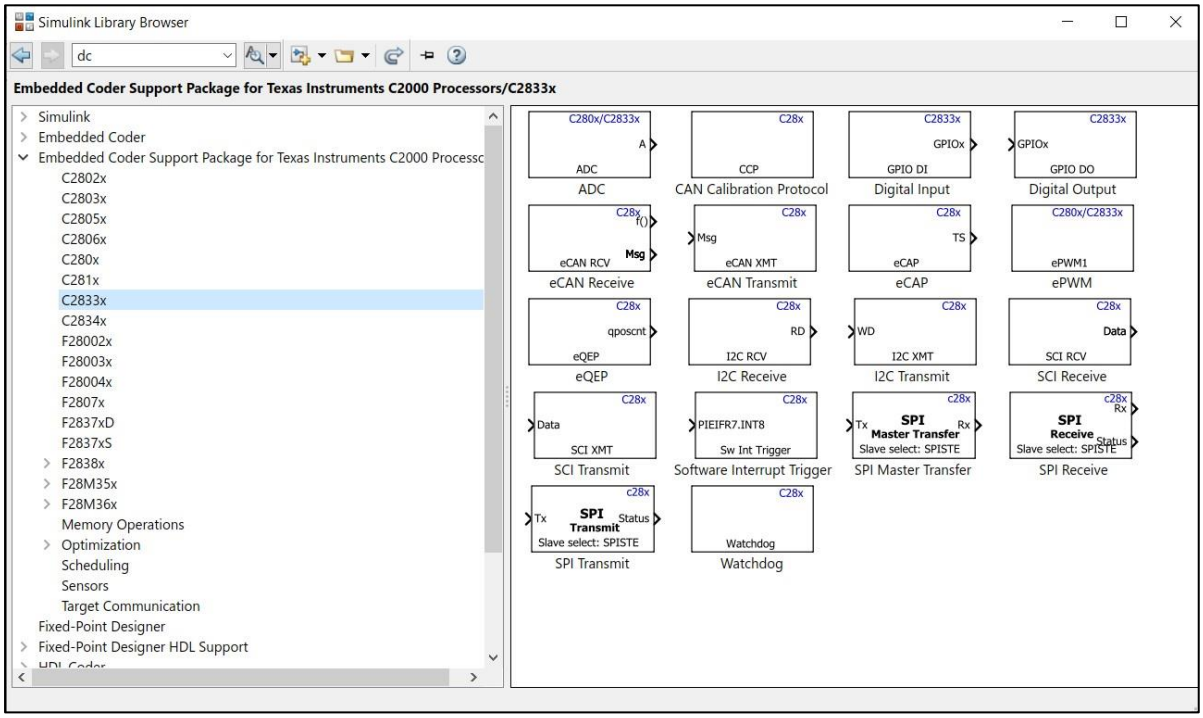


Fig. 5. Embedded Coder Support Package for TI C2000

Now, suppose if the user tends to use a different processor, then he just needs to switch those blocks for a different desirable processor belonging to the c2000 family and make the necessary changes in the configuration window and the model is good to go. This helps users avoid unnecessarily rewriting or rebuilding the whole model again and moreover provides a whole lot of flexibility to the user.

4. CONVENTIONAL PROGRAMMING

Conventional Programming serves as means to provide the pre-requisite work needed to get acquainted with the MCU target, its peripherals, GPIOs and programming flow before moving on to build the actual model of the brake dynamics simulator on Simulink-Embedded Coder. Hence, this chapter will be centered around getting familiar with programming the MCU registers inside the controller before proceeding forward with the Model Based Development. Keeping that in mind, the language used for the purpose is Embedded C and programming would be followed as per the general design flow of the Code Composer Studio Integrated Development Environment [CCS-IDE], that is, with all the necessary libraries and header files of the F28335 MCU being imported into the same CCS Project directory as the main source code file.

Bare-metal programming is a practice aimed to provide a good hands-on knowledge to test, build and debug embedded code and get proper familiarity with the various modules, peripherals, GPIOs etc. associated with any MCU. Programming bare-metal refers to programming an embedded system without the use of an underlying Operating System (OS). The conventional pattern of trying to learn any microcontroller goes in accordance with the level of complexity that each of its features/peripherals has. Figure below shows the pinout for the F28335 DSC [digital signal controller] listing all the pinouts for numerous peripherals provided by the controller.

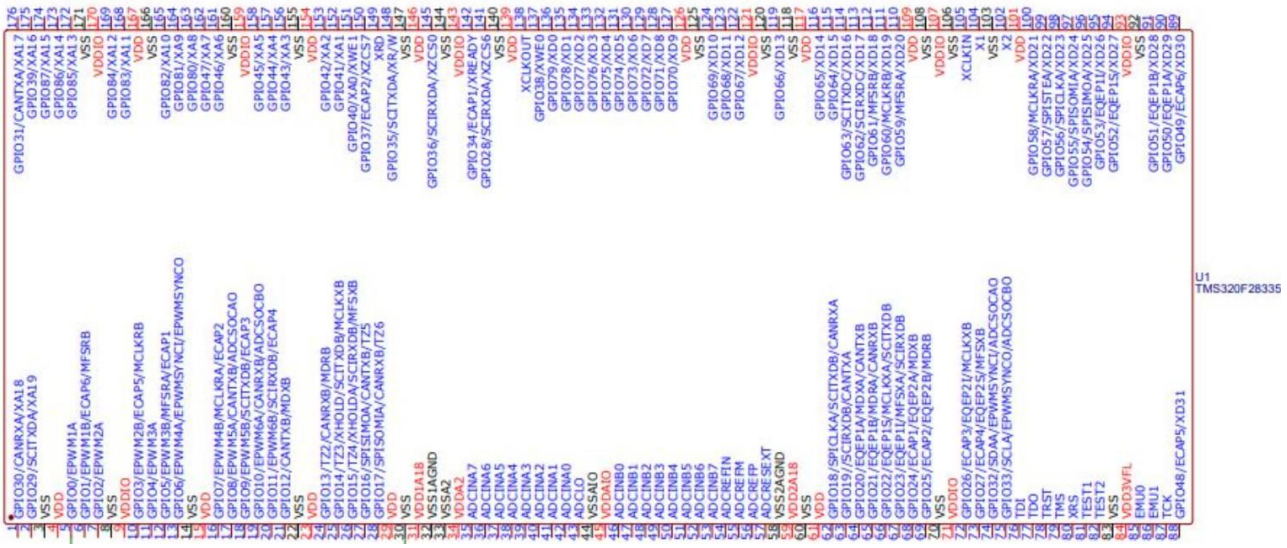


Fig. 1. F28335 GPIOs/Peripherals Pinout

There are several peripherals provided in the F28335 MCU that can be programmed directly. Let us begin discussing some of these peripherals starting with the relatively simple GPIO.

4.1 GPIO-General Purpose Input/Output

General Purpose Input/Output is the one of the first basic peripherals that can be programmed to get acquainted with the embedded programming flow of the F28335 MCU.
The GPIO peripheral provides special general-purpose contacts that can be configured as inputs or outputs. When configured as an input, one can determine the status of the input by reading the state of the internal register. When configured as an output, we can write to the internal register to monitor the managed state on the output contact.

But as simple as it sounds, there are several key factors [or so-called Registers] to programming the GPIO peripheral for even a relatively simple LED blinking program. These factors determine which GPIO is to be selected, whether it has to be programmed as input or output and so on. Hence, it is extremely important to note which bits of which register have to be kept high/low while writing the code depending upon the desired logic. Likewise, the TMS320x2833x datasheet and technical reference manual provided by Texas Instruments comes in handy and from an embedded programmer's perspective, it is highly important to know how to read them as well.

4.1.1 GPIO/Peripheral Multiplexing

F28335 features what is known as GPIOMUX or General-Purpose Input/Output Multiplexer Registers. Most of the peripherals in the target are multiplexed with GPIO signals. This enables the user to use a pin as GPIO if the peripheral signal or function is not used. On reset, GPIO pins are configured as inputs. The user can individually program each pin for GPIO mode or peripheral signal mode. For specific inputs, the user can also select the number of input qualification cycles. This is to filter unwanted noise glitches. The GPIO signals can also be used to bring the device out of specific low-power modes.

It is important to note that up to three independent peripheral signals are multiplexed on a single GPIO-enabled pin in the F28335 MCU in addition to the individual pin bit-I/O capability. There are three 32-bit I/O ports. Port A consists of GPIO0-GPIO31, port B consists of GPIO32-GPIO63, and port C consists of GPIO64-87. Figure 1-40 shows the basic modes of operation for the GPIO module.

Several tables shown ahead give an insight and a general idea about the Registers associated with the discussion above.

Additionaly, shown below is an overview of the possible multiplexing combinations sorted under GPIOA MUX which comprises of GPAMUX1 and GPAMUX2 registers, each 32-bit wide and made up of several peripherals multiplexed onto the same set of bits as shown. For instance, the multiplexing for the GPIO7 is controlled by writing to the GPAMUX1[15:14] bits. The pin is then configured as either GPIO7, or one of the other three peripheral functions. If no pin is configured as an input to a peripheral, or if more than one pin is configured as an input for the same peripheral, then the input to the peripheral will either default to a 0 or a 1.

	Default at Reset Primary I/O Function	Peripheral Selection	Peripheral Selection 2	Peripheral Selection 3
GPAMUX1 Register Bits	**(GPAMUX1 bits = 00)**	**(GPAMUX1 bits = 01)**	**(GPAMUX1 bits = 10)**	**(GPAMUX1 bits = 11)**
1-0	GPIO0	EPWM1A (O)	Reserved(1)	Reserved(1)
3-2	GPIO1	EPWM1B (O)	ECAP6 (I/O)	MFSRB (I/O)(1)
5-4	GPIO2	EPWM2A (O)	Reserved(1)	Reserved(1)
7-6	GPIO3	EPWM2B (O)	ECAP5 (I/O)	MCLKRB (I/O)(1)
9-8	GPIO4	EPWM3A (O)	Reserved(1)	Reserved(1)
11-10	GPIO5	EPWM3B (O)	MFSRA (I/O)	ECAP1 (I/O)
13-12	GPIO6	EPWM4A (O)	EPWMSYNCI (I)	EPWMSYNCO (O)
15-14	GPIO7	EPWM4B (O)	MCLKRA (I/O)	ECAP2 (I/O)
17-16	GPIO8	EPWM5A (O)	CANTXB (O)	ADCSOCAO (O)
19-18	GPIO9	EPWM5B (O)	SCITXDB (O)	ECAP3 (I/O)
21-20	GPIO10	EPWM6A (O)	CANRXB (I)	ADCSOCBO (O)
23-22	GPIO11	EPWM6B (O)	SCIRXDB (I)	ECAP4 (I/O)
25-24	GPIO12	TZ1 (I)	CANTXB (O)	MDXB (O)
27-26	GPIO13	TZ2 (I)	CANRXB (I)	MDRB (I)
29-28	GPIO14	TZ3/XHOLD (I)	SCITXDB (O)	MCLKXB (I/O)
31-30	GPIO15	TZ4/XHOLDA (O)	SCIRXDB (I)	MFSXB (I/O)
GPAMUX2 Register Bits	**(GPAMUX2 bits = 00)**	**(GPAMUX2 bits = 01)**	**(GPAMUX2 bits = 10)**	**(GPAMUX2 bits = 11)**
1-0	GPIO16	SPISIMOA (I/O)	CANTXB (O)	TZ5 (I)
3-2	GPIO17	SPISOMIA (I/O)	CANRXB (I)	TZ6 (I)
5-4	GPIO18	SPICLKA (I/O)	SCITXDB (O)	CANRXA (I)
7-6	GPIO19	SPISTEA (I/O)	SCIRXDB (I)	CANTXA (O)
9-8	GPIO20	EQEP1A (I)	MDXA (O)	CANTXB (O)
11-10	GPIO21	EQEP1B (I)	MDRA (I)	CANRXB (I)
13-12	GPIO22	EQEP1S (I/O)	MCLKXA (I/O)	SCITXDB (O)
15-14	GPIO23	EQEP1I (I/O)	MFSXA (I/O)	SCIRXDB (I)
17-16	GPIO24	ECAP1 (I/O)	EQEP2A (I)	MDXB (O)
19-18	GPIO25	ECAP2 (I/O)	EQEP2B (I)	MDRB (I)
21-20	GPIO26	ECAP3 (I/O)	EQEP2I (I/O)	MCLKXB (I/O)
23-22	GPIO27	ECAP4 (I/O)	EQEP2S (I/O)	MFSXB (I/O)
25-24	GPIO28	SCIRXDA (I)	XZCS6 (O)	XZCS6 (O)
27-26	GPIO29	SCITXDA (O)	XA19 (O)	XA19 (O)
29-28	GPIO30	CANRXA (I)	XA18 (O)	XA18 (O)
31-30	GPIO31	CANTXA (O)	XA17 (O)	XA17 (O)

Fig. 2. GPIOA-MUX Peripherals Table

NOTE: Apart from GPIOA MUX, the target also features GPIOB MUX and GPIOC MUX each of which are also subdivided into two 32-bit registers similar to GPIOAMUX. The GPAMUX registers were sufficient enough to help accomplish most of the conventional coding tasks successfully. The methodology to programming the other two is exactly the same. More can be found about GPBMUX and GPCMUX registers in the target's reference manual.

4.1.2 Configuration Overview of General-Purpose Registers

The pin function assignments, input qualification, and the external interrupts sources are all controlled by the GPIO configuration control registers. These registers [as listed below] are key to programming and configuring the GPIO pins to match the programmer's requirements.

Name [1]	Address	Size (x16)	Register Description
GPACTRL	0x6F80	2	GPIO A Control Register (GPIO0-GPIO31)
GPAQSEL1	0x6F82	2	GPIO A Qualifier Select 1 Register (GPIO0-GPIO15)
GPAQSEL2	0x6F84	2	GPIO A Qualifier Select 2 Register (GPIO16-GPIO31)
GPAMUX1	0x6F86	2	GPIO A MUX 1 Register (GPIO0-GPIO15)
GPAMUX2	0x6F88	2	GPIO A MUX 2 Register (GPIO16-GPIO31)
GPADIR	0x6F8A	2	GPIO A Direction Register (GPIO0-GPIO31)
GPAPUD	0x6F8C	2	GPIO A Pull Up Disable Register (GPIO0-GPIO31)
GPBCTRL	0x6F90	2	GPIO B Control Register (GPIO32-GPIO63)
GPBQSEL1	0x6F92	2	GPIO B Qualifier Select 1 Register (GPIO32-GPIO47)
GPBQSEL2	0x6F94	2	GPIO B Qualifier Select 2 Register (GPIO48 - GPIO63)
GPBMUX1	0x6F96	2	GPIO B MUX 1 Register (GPIO32-GPIO47)
GPBMUX2	0x6F98	2	GPIO B MUX 2 Register (GPIO48-GPIO63)
GPBDIR	0x6F9A	2	GPIO B Direction Register (GPIO32-GPIO63)
GPBPUD	0x6F9C	2	GPIO B Pull Up Disable Register (GPIO32-GPIO63)
GPCMUX1	0x6FA6	2	GPIO C MUX 1 Register (GPIO64-GPIO79)
GPCMUX2	0x6FA8	2	GPIO C MUX 2 Register (GPIO80-GPIO87)
GPCDIR	0x6FAA	2	GPIO C Direction Register (GPIO64-GPIO87)
GPCPUD	0x6FAC	2	GPIO C Pull Up Disable Register (GPIO64-GPIO87)

Fig. 3. GPIO Control Registers Table

Name	Address	Size (x16)	Register Description
GPADAT	0x6FC0	2	GPIO A Data Register (GPIO0-GPIO31)
GPASET	0x6FC2	2	GPIO A Set Register (GPIO0-GPIO31)
GPACLEAR	0x6FC4	2	GPIO A Clear Register (GPIO0-GPIO31)
GPATOGGLE	0x6FC6	2	GPIO A Toggle Register (GPIO0-GPIO31)
GPBDAT	0x6FC8	2	GPIO B Data Register (GPIO32-GPIO63)
GPBSET	0x6FCA	2	GPIO B Set Register (GPIO32-GPIO63)
GPBCLEAR	0x6FCC	2	GPIO B Clear Register (GPIO32-GPIO63)

Name	Address	Size (x16)	Register Description
GPBTOGGLE	0x6FCE	2	GPIO B Toggle Register (GPIO32-GPIO63)
GPCDAT	0x6FD0	2	GPIO C Data Register (GPIO64 - GPIO87)
GPCSET	0x6FD2	2	GPIO C Set Register (GPIO64 - GPIO87)
GPCCLEAR	0x6FD4	2	GPIO C Clear Register (GPIO64 - GPIO87)
GPCTOGGLE	0x6FD6	2	GPIO C Toggle Register (GPIO64 - GPIO87)

Fig. 4. GPIO Data Registers Table

- **GPxDAT Register**

Each I/O port has one data register. Each bit in the data register corresponds to one GPIO pin. No matter how the pin is configured (GPIO or peripheral function), the corresponding bit in the data register reflects the current state of the pin after qualification. Writing to the GPxDAT register clears or sets the corresponding output latch and if the pin is enabled as a general-purpose output (GPIO output) the pin will also be driven either low or high. For example, changing the output latch level of GPIOA0 by writing to the GPADAT register bit 0, using a read-modify-write instruction. The problem can occur if another I/O port A signal changes level between the read and the write stage of the instruction. To tackle this, comes the next set of registers, that is, GPxSET, GPxCLEAR, and GPxTOGGLE to lead O/P latch.

- **GPxSET Register**

The set registers are used to drive specified GPIO pins high without disturbing other pins. Each I/O port has one set register and each bit corresponds to one GPIO pin. If the corresponding pin is configured as an output, then writing a 1 to that bit in the set register will set the output latch high and the corresponding pin will be driven high. Writing a 0 to any bit in the set registers has no effect.

- **GPxCLEAR Register**

The clear registers are used to drive specified GPIO pins low without disturbing other pins. Each I/O port has one clear register. If the corresponding pin is configured as a general-purpose output, then writing a 1 to the corresponding bit in the clear register will clear the output latch and the pin will be driven low. Writing a 0 to any bit in the clear registers has no effect.

- **GPxTOGGLE Register**

The toggle registers are used to drive specified GPIO pins to the opposite level without disturbing other pins. Each I/O port has one toggle register. The toggle registers always read back 0. If the corresponding pin is configured as an output, then writing a 1 to that bit in the toggle register flips the output latch and pulls the corresponding pin in the opposite direction. That is, if the output pin is driven low, then writing a 1 to the corresponding bit in the toggle register will pull the pin high. Likewise, if the output pin is high, then writing a 1 to the corresponding bit in the toggle register will pull the pin low. If the pin is not configured as a GPIO output, then the value will be latched but the pin will not be driven. Writing a 0 to any bit in the toggle registers has no effect.

Sections ahead feature several coding tasks wherein all these registers come handy while programming the F28335 target. Snippets and algorithms are discussed wherever necessary.

4.1.3 LED Toggling Program

Task 1: The TMS320F28335 control card has an in-built LED connected to the GPIO31 peripheral and the purpose is to program this LED through its corresponding GPIO pin to make it toggle or flicker. In order to do achieve that, it important to select the correct peripheral from the GPAMUX register and also set its direction to Input or Output as desired. The programming flow and the respective Embedded C code corresponding to the logic are discussed.

Code Flow

- Start
- Include Necessary Header files and declare DELAY function prototype
- Enter Main Function
- Initialize system control and PIE control registers to known state
- Configure MUX and Direction register to set peripheral to GPIO31 and direction as output
- Reset Interrupt Enable and Interrupt Flag Registers
- While (condition is true) do

  Set GPIO31 bit in Toggle Register to 1.

  Wait for delay
- Exit Main Function
- Stop

Code Composer Studio Code Snippet

```
#include "DSP28x_Project.h"

void delay_loop(void);

void main(void)
{
    InitSysCtrl();

    EALLOW;
    GpioCtrlRegs.GPAMUX2.bit.GPIO31 = 0;
    GpioCtrlRegs.GPADIR.bit.GPIO31 = 1;
    EDIS;
    IER = 0x0000;
    IFR = 0x0000;
    while(1)
    {
        GpioDataRegs.GPATOGGLE.bit.GPIO31 = 1;
        delay_loop();
    }
}

void delay_loop()
{
    volatile long i;
    for (i = 0; i < 1000000; i++)
    {}
}
```

4.2 Interrupt Handling

An interrupt is an external or internal event to get the attention of the CPU. Once the controller detects the interrupt, it suspends the current task and performs a special service procedure known as Interrupt Service Routine (ISR).

Interrupts are useful in many cases where a process simply wants to continue performing its main function and other units (timers or external events) seek its attention when needed. In other words, the microcontroller does not need to monitor timers, serial communication, or external pins.

Every time an event related to these units occurs, the microcontroller is notified with the help of interrupts.

4.2.1 Peripheral Interrupt Expansion

The peripheral interrupt expansion (PIE) block multiplexes numerous interrupt sources into a smaller set of interrupt inputs. The PIE block can support 96 individual interrupts that are grouped into blocks of eight. Each group is fed into one of 12 core interrupt lines (INT1 to INT12). Each of the 96 interrupts is supported by its own vector stored in a dedicated RAM block that can be modified. The CPU, upon servicing the interrupt, automatically fetches the appropriate interrupt vector. It takes nine CPU clock cycles to fetch the vector and save critical CPU registers. Therefore, the CPU can respond quickly to interrupt events. Prioritization of interrupts is controlled in hardware and software. Each individual interrupt can be enabled/disabled within the PIE block.

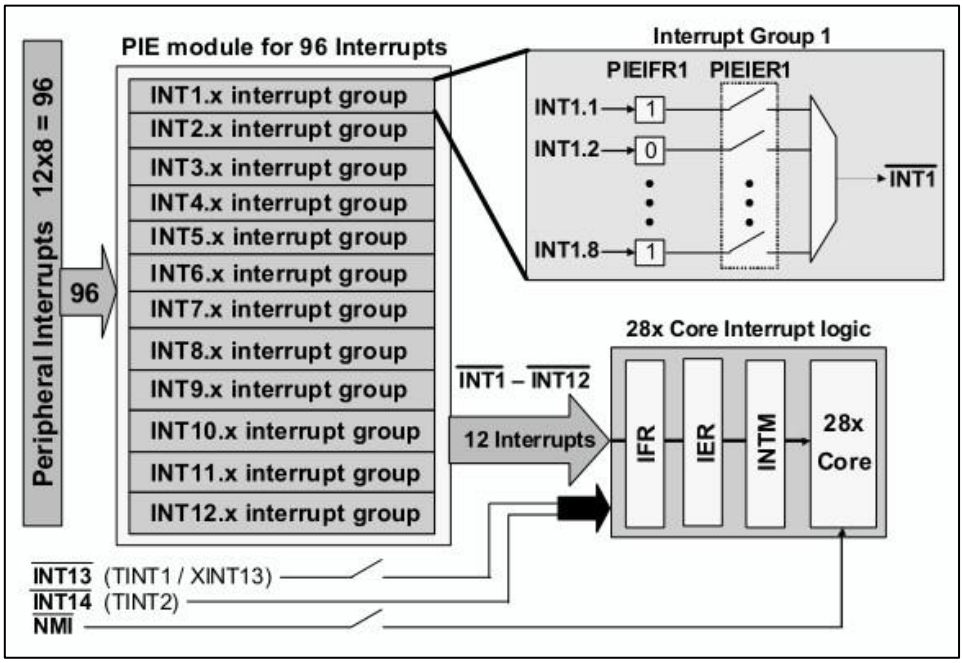


Fig. 5. Peripheral Interrupt Expansion

- **Peripheral Level Handling**

An interrupt-generating event occurs in a peripheral. The interrupt flag (IF) bit corresponding to that event is set in a register for that particular peripheral. If the corresponding interrupt enable (IE) bit is set, the peripheral generates an interrupt request to the PIE controller. If the interrupt is not enabled at the peripheral level, then the IF remains set until cleared by software. If the interrupt is enabled at a later time, and the interrupt flag is still set, the interrupt request is asserted to the PIE. Interrupt flags within the peripheral registers must be manually cleared. See the peripheral reference guide for a specific peripheral for more information.

- **PIE Level Handling**

The PIE block multiplexes eight peripheral and external pin interrupts into one CPU interrupt. These interrupts are divided into 12 groups: PIE group 1 - PIE group 12. The interrupts within a group are multiplexed into one CPU interrupt. For example, PIE group 1 is multiplexed into CPU interrupt 1 (INT1) and PIE group 12 is multiplexed into CPU interrupt 12 (INT12). For the nonmultiplexed interrupts, the PIE passes the request directly to the CPU. For multiplexed interrupts, each interrupt group in the PIE block has an associated flag register (PIEIFRx) and enable (PIEIERx) register (x = PIE group 1 - PIE group 12). Each bit, referred to as y, corresponds to one of the 8 MUXed interrupts within a group. Thus PIEIFRx.y and PIEIERx.y correspond to interrupt y (y = 1-8) in PIE group x (x = 1-12). Also, there is one acknowledge bit (PIEACK) for every PIE interrupt group referred to as PIEACKx (x = 1-12). Once the request is made to the PIE controller, the corresponding PIE interrupt flag (PIEIFRx.y) bit is set. If the PIE interrupt enable (PIEIERx.y) bit is also set for the given interrupt then PIE checks the corresponding PIEACKx bit to determine if the CPU is ready for an interrupt from that group. If the PIEACKx bit is clear, then PIE sends the interrupt request to the CPU. If PIEACKx is set, then the PIE waits until it's cleared to send the request for INTx.

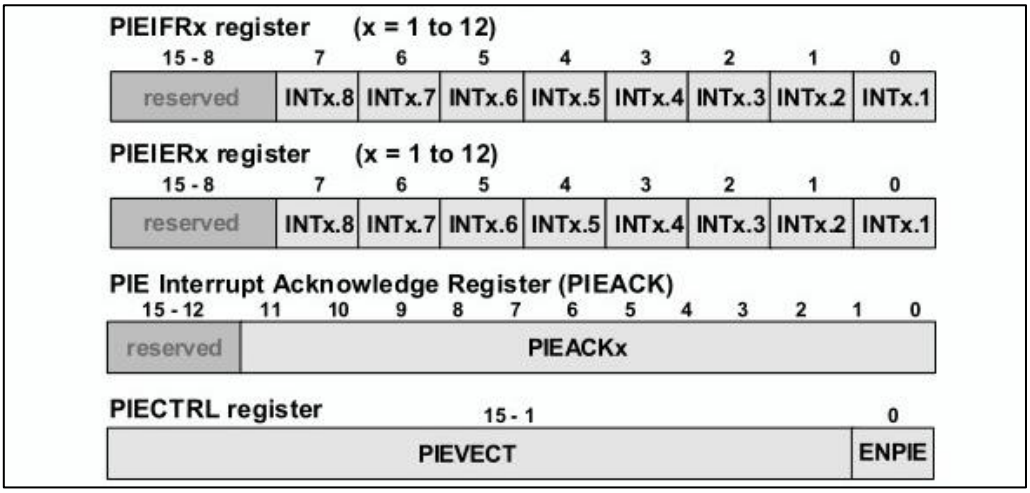


Fig. 6. PIE Registers

	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1
INT1	WAKEINT	TINT0	ADCINT	XINT2	XINT1		SEQ2INT	SEQ1INT
INT2			EPWM6_TZINT	EPWM5_TZINT	EPWM4_TZINT	EPWM3_TZINT	EPWM2_TZINT	EPWM1_TZINT
INT3			EPWM6_INT	EPWM5_INT	EPWM4_INT	EPWM3_INT	EPWM2_INT	EPWM1_INT
INT4			ECAP6_INT	ECAP5_INT	ECAP4_INT	ECAP3_INT	ECAP2_INT	ECAP1_INT
INT5							EQEP2_INT	EQEP1_INT
INT6			MXINTA	MRINTA	MXINTB	MRINTB	SPITXINTA	SPIRXINTA
INT7			DINTCH6	DINTCH5	DINTCH4	DINTCH3	DINTCH2	DINTCH1
INT8			SCITXINTC	SCIRXINTC			I2CINT2A	I2CINT1A
INT9	ECAN1_INTB	ECAN0_INTB	ECAN1_INTA	ECAN0_INTA	SCITXINTB	SCIRXINTB	SCITXINTA	SCIRXINTA
INT10								
INT11								
INT12	LUF	LVF		XINT7	XINT6	XINT5	XINT4	XINT3

Fig. 7. PIE Interrupts Assignment Table

- **CPU Level Handling**

Once the request is sent to the CPU, the CPU level interrupt flag (IFR) bit corresponding to INTx is set. After a flag has been latched in the IFR, the corresponding interrupt is not serviced until it is appropriately enabled in the CPU interrupt enable (IER) register or the debug interrupt enable register (DBGIER) and the global interrupt mask (INTM) bit.

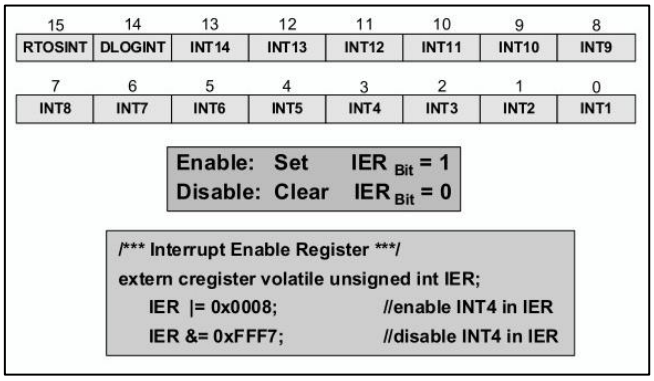
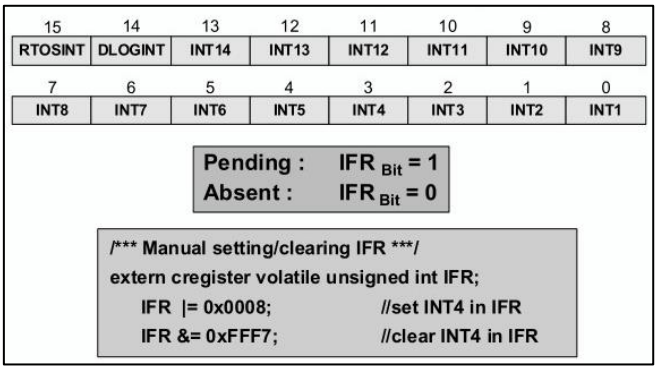


Fig. 8(a). Interrupt Enable Register [IER]



8(b). Interrupt Flag Register [IFR]

The requirements for enabling the maskable interrupt at the CPU level depends on the interrupt handling process being used. In the standard process, which happens most of the time, the DBGIER register is not used. When the 28x is in real-time emulation mode and the CPU is halted, a different process is used. In this special case, the DBGIER is used and the INTM bit is ignored. If the DSP is in real-time mode and the CPU is running, the standard interrupt-handling process applies. The CPU then prepares to service the interrupt. This preparation process is described in detail in TMS320C28x DSP CPU and Instruction Set Reference Guide (literature number SPRU430). In preparation, the corresponding CPU IFR and IER bits are cleared, EALLOW and LOOP are cleared, INTM and DBGM are set, the pipeline is flushed and the return address is stored, and the automatic context save is performed.

The vector of the ISR is then fetched from the PIE module. If the interrupt request comes from a multiplexed interrupt, the PIE module uses the group PIEIERx and PIEIFRx registers to decode which interrupt needs to be serviced. The address for the interrupt service routine that is executed is fetched directly from the PIE interrupt vector table.

- **PIE Vector Table**

The PIE vector table consists of a 256 x 16 SARAM block that can also be used as RAM if the PIE block is unused. PIE vector table contents are undefined on reset. The CPU fixes interrupt priority for INT1 to INT12. The PIE controls priority for each group of eight interrupts. Out of the 96 possible MUXed interrupts present in the table, 43 interrupts are currently used while the remaining interrupts are reserved for future devices.

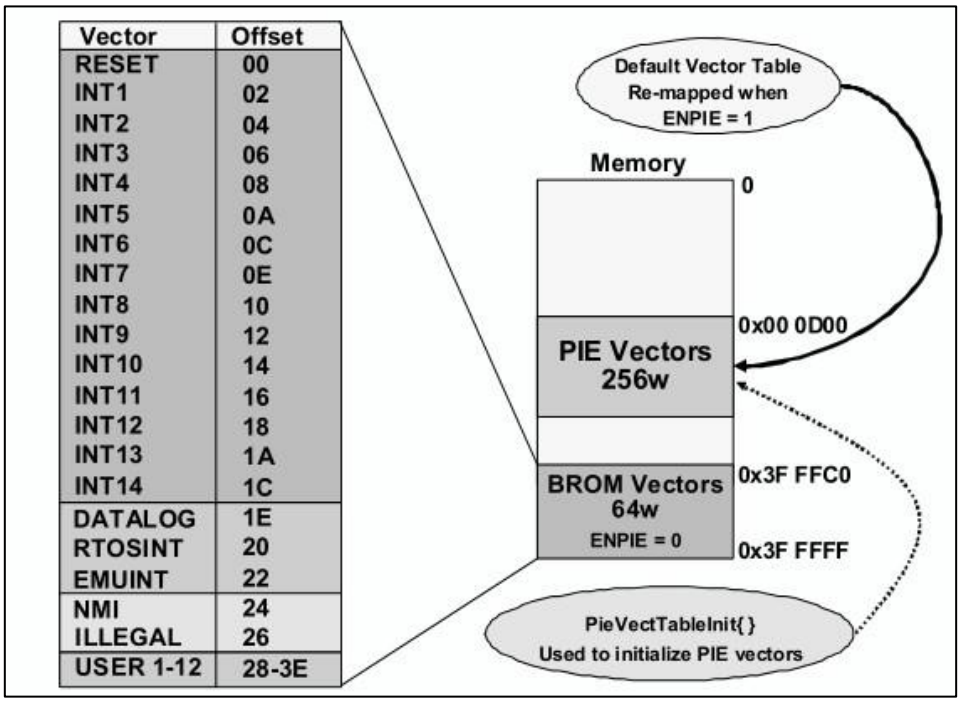


Fig. 9. PIE Vector Table

4.2.2 Multiplexed Interrupt Request Flow [Peripheral to CPU]

Step 1. Any peripheral or external interrupt within the PIE group generates an interrupt. If interrupts are enabled within the peripheral module, then the interrupt request is sent to the PIE module.

Step 2. The PIE module recognizes that interrupt y within PIE group x (INTx.y) has asserted an interrupt and the appropriate PIE interrupt flag bit is latched: PIEIFRx.y = 1.

Step 3. For the interrupt request to be sent from the PIE to the CPU, both of the following conditions must be true:

a) The proper enable bit must be set (PIEIERx.y = 1) and

b) The PIEACKx bit for the group must be clear.

Step 4. If both conditions in 3a and 3b are true, then an interrupt request is sent to the CPU and the acknowledge bit is again set (PIEACKx = 1). The PIEACKx bit will remain set until you clear it to indicate that additional interrupts from the group can be sent from the PIE to the CPU.

Step 5. The CPU interrupt flag bit is set (CPU IFRx = 1) to indicate a pending interrupt x at the CPU level.

Step 6. If the CPU interrupt is enabled (CPU IER bit x = 1, or DBGIER bit x = 1) AND the global interrupt mask is clear (INTM = 0) then the CPU will service the INTx.

Step 7. The CPU recognizes the interrupt and performs the automatic context save, clears the IER bit, sets INTM, and clears EALLOW. All of the steps that the CPU takes in order to prepare to service the interrupt are documented in the TMS320C28x CPU and Instruction Set Guide.

Step 8. The CPU will then request the appropriate vector from the PIE.

Step 9. For multiplexed interrupts, the PIE module uses the current value in the PIEIERx and PIEIFRx registers to decode which vector address should be used. There are two possible cases:

a. The vector for the highest priority interrupt within the group that is both enabled in the PIEIERx register, and flagged as pending in the PIEIFRx is fetched and used as the branch address. In this manner if an even higher priority enabled interrupt was flagged after Step 7, it will be serviced first.

b. If no flagged interrupts within the group are enabled, then the PIE will respond with the vector for the highest priority interrupt within that group. That is the branch address used for INTx.1. This behavior corresponds to the 28x TRAP or INT instructions.

NOTE: Because the PIEIERx register is used to determine which vector will be used for the branch, care must be taken when clearing the bits within the PIEIERx register.

4.2.3 LED Toggling using Timer Interrupt

Task 2: The end result is a little similar to the one seen earlier in 4.1.3 LED Toggling but this time the toggling is controlled by the Interrupt Service Routine or the ISR. The Timer Interrupt 0 is used for the purpose wherein every time the that the ISR is called, the control is passed from the current program and passed to the ISR to toggle the LEDs connected to GPIO31 and GPIO34 peripherals. Simply put, an interrupt service routine (ISR) is a software routine that hardware invokes in response to an interrupt. ISR examines the type of interrupt and its condition, determines how to handle it, executes it, and returns a logical interrupt value.

Unlike the previous case where the toggling was happening directly inside the infinite loop, this time, it is being manipulated via the in-between mechanism of interrupt handling where the Interrupt comes as a condition that makes the CPU suspend the current program instruction and pass the flow to execute an ISR which again is a specially written code segment to service the condition that caused the interrupt, which in this case, would be to toggle the LEDs.

Code Flow

- Start
- Include Necessary Header files and declare function prototypes
- Enter Main Function
- Initialize system control registers, PIE control registers and PIE Vector Table
- Point Timer Interrupt 0 present in PIE Vector Table to the ISR Base Address
- Initialize CPU Timers
- Configure MUX registers and Direction Registers to set peripherals to the required GPIO31 and GPIO34 and direction as output
- Initialize Interrupt Enable Register to 1
- To enable Timer Interrupt 0, set bit 7 of PIE INT1 Interrupt Enable Register to 1
- Enable Global Interrupt and Global Real-time Interrupt
- Loop Forever
- Exit Main Function
- Interrupt Service routine to be invoked as per Timer Interrupt 0
    - Increment Interrupt Count
    - Set GPIO31 and GPIO34 bits in Toggle Registers to 1
    - Set PIE Acknowledge Bit for INT1 Group to 1
- Exit Interrupt Service Routine
- Stop

Code Composer Studio Code Snippet

```
#include "DSP28x_Project.h"
__interrupt void cpu_timer0_isr(void);

void main(void)
{
    InitSysCtrl();
    DINT;
    InitPieCtrl();
    IER = 0x0000;
    IFR = 0x0000;
    InitPieVectTable();

    EALLOW;         // Enable writing to protected registers
    PieVectTable.TINT0 = &cpu_timer0_isr;
    EDIS;           // Disable writing to protected registers

    InitCpuTimers();
    #if (CPU_FRQ_150MHZ)
    ConfigCpuTimer(&CpuTimer0, 150, 500000);
    #endif
    CpuTimer0Regs.TCR.all = 0x4000;

    EALLOW;
    GpioCtrlRegs.GPBMUX1.bit.GPIO34 = 0;
    GpioCtrlRegs.GPBDIR.bit.GPIO34 = 1;
    EDIS;
    EALLOW;
    GpioCtrlRegs.GPAMUX2.bit.GPIO31 = 0;
    GpioCtrlRegs.GPADIR.bit.GPIO31 = 1;
    EDIS;

    IER |= M_INT1;
    PieCtrlRegs.PIEIER1.bit.INTx7 = 1;

    EINT;   // Enable Global Interrupt INTM
    ERTM;   // Enable Global Real-time interrupt DBGM
    for(;;);
}

__interrupt void cpu_timer0_isr(void)
{
    CpuTimer0.InterruptCount++;
    GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1;
    GpioDataRegs.GPATOGGLE.bit.GPIO31 = 1;

    PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
}
```

4.2 ePWM-Enhanced Pulse Width Modulator

Pulse width modulation (PWM) is a method for representing an analog signal with a digital Approximation, that is, it represents a given signal as a sequence of pulses. Since, the output of PWM is a digital signal, it makes it easy for a DSP such as F28335 to process and send it to the output. The enhanced pulse width modulator (ePWM) peripheral is a key element in controlling many of the power electronic systems found in both commercial and industrial equipment such as digital motor control, switch mode power supply control, UPS and so on.

An effective PWM peripheral must be able to generate complex pulse width waveforms with minimal CPU overhead or intervention. It needs to be highly programmable and very flexible while being easy to understand and use. The ePWM unit described here addresses these requirements by allocating all needed timing and control resources on a per PWM channel basis.

Multiple ePWM modules are instanced within the target where one single ePWM module represents one complete PWM channel composed of two PWM outputs ePWMxA and ePWMxB [x ranges from 1 to 6]. The 2833x devices contain up to six enhanced PWM (ePWM) modules (ePWM1 to ePWM6) meaning a total of 12 PWM outputs. The PWM output signals are made available external to the device through the GPIO peripherals multiplexed altogether as is described in the datasheet and reference guide for the device taken.

The ePWM modules are chained together via a clock synchronization scheme that allows them to operate as a single system when required. All events pertaining to ePWM can trigger both CPU interrupts and ADC start of conversion (SOC) as discussed briefly below.

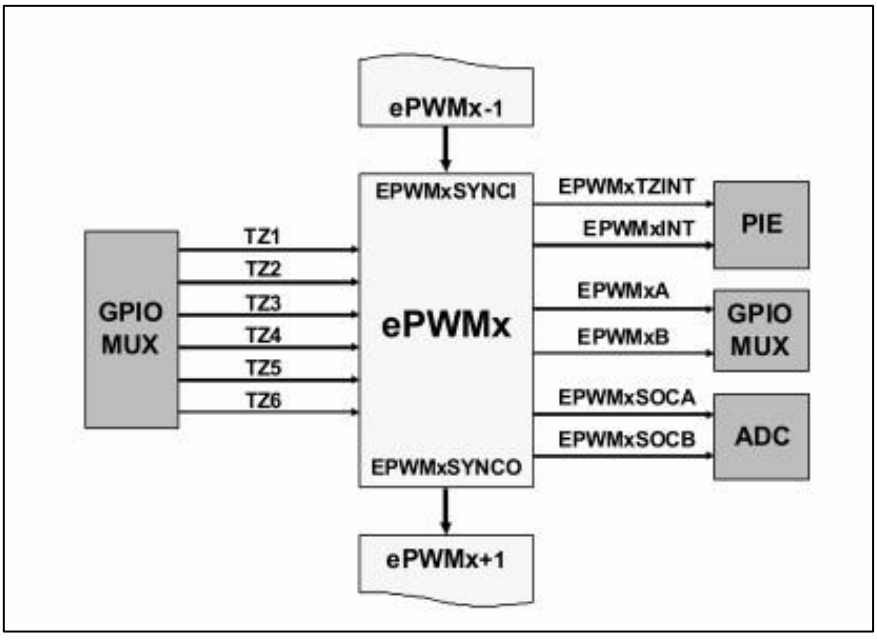


Fig. 10. ePWM Signal Connections

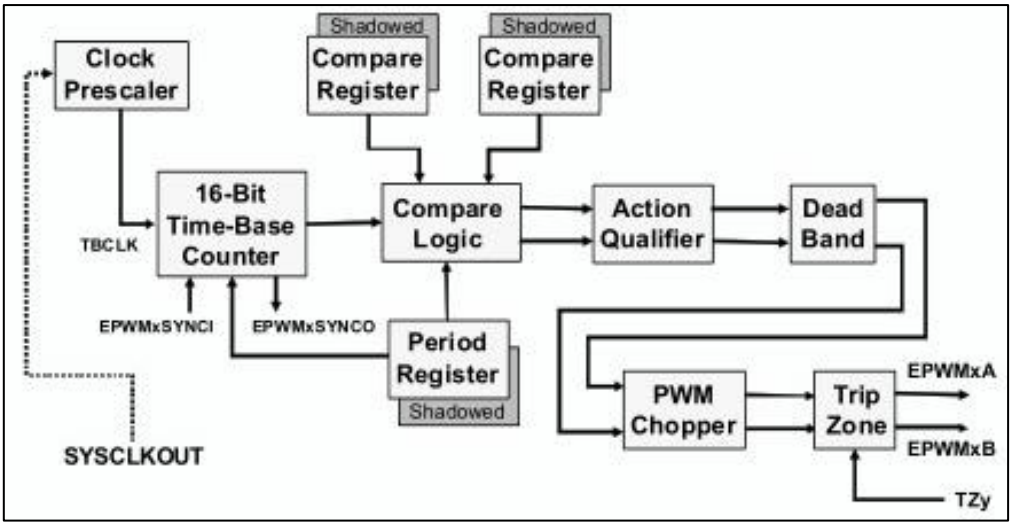


Fig. 11. ePWM Block Diagram

4.2.1 ADC start-of-conversion signals (EPWMxSOCA and EPWMxSOCB)

Each ePWM module has two ADC start of conversion signals (one for each sequencer). Any ePWM module can trigger a start of conversion for either sequencer. Which event triggers the start of conversion is configured in the Event-Trigger submodule of the ePWM.

4.2.2 Time-Base (TB) Submodule

Each ePWM module has its own time-base submodule that determines all of the event timing for the ePWM module. Built-in synchronization logic allows the time-base of multiple ePWM modules to work together as a single system. The Time-based submodule provides a lot of additional features that are configurable as per the programmer's requirements such as:

- Specify the ePWM time-base counter (TBCTR) frequency or period to control how often events occur.
- Manage time-base synchronization and phase relationship with other ePWM modules
- Configure 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.

Name	Description	Structure
TBCTL	Time-Base Control	EPwmxRegs.TBCTL.all =
TBSTS	Time-Base Status	EPwmxRegs.TBSTS.all =
TBPHS	Time-Base Phase	EPwmxRegs.TBPHS =
TBCTR	Time-Base Counter	EPwmxRegs.TBCTR =
TBPRD	Time-Base Period	EPwmxRegs.TBPRD =

Fig. 12. ePWM Time Based Sub-Module Registers

- **Time-Base Control Register [TBCTL]**

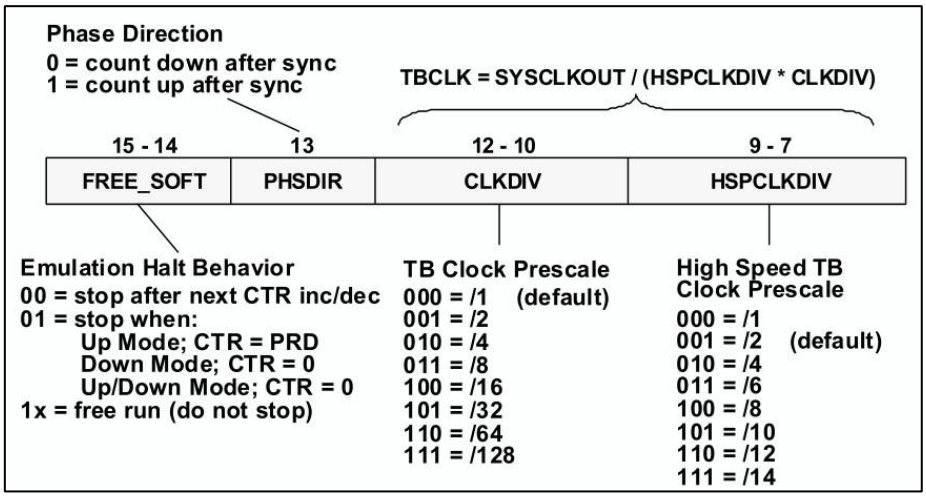


Fig. 13(a). TBCTL [Higher order bits]

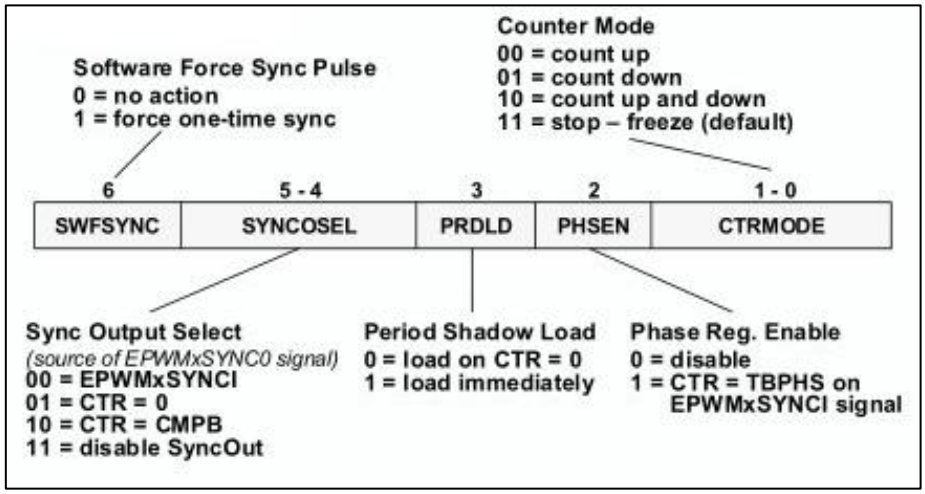


Fig. 13(b). TBCTL [Lower order bits]

- **Time-Base Period Register [TBPRD]**

Time-base counter [CTR] in ePWM equals to the specified period that is desired out of the PWM output signal and this value is affected directly by the highly significant TBPRD register. This is because the signal is generated whenever the counter value is equal to the active period register value, that is, when TBCTR = TBPRD.

Bits	Name	Value	Description
15-0	TBPRD	0000-FFFFh	These bits determine the period of the time-base counter. This sets the PWM frequency. Shadowing of this register is enabled and disabled by the TBCTL[PRDLD] bit. By default this register is shadowed. • If TBCTL[PRDLD] = 0, then the shadow is enabled and any write or read will automatically go to the shadow register. In this case, the active register will be loaded from the shadow register when the time-base counter equals zero. • If TBCTL[PRDLD] = 1, then the shadow is disabled and any write or read will go directly to the active register, that is the register actively controlling the hardware. • The active and shadow registers share the same memory map address.

Fig. 14. TBCTL [Lower order bits]

4.2.3 ePWM Period and Frequency

The frequency of PWM events is controlled by the time-base period (TBPRD) register and the mode of the time-base counter. Figure 3-6 shows the period (Tpwm) and frequency (Fpwm) relationships for the up-count, down-count, and up-down-count time-base counter modes when the period is set to 4 (TBPRD = 4). The time increment for each step is defined by the time-base clock (TBCLK) which is a pre-scaled version of the system clock (SYSCLKOUT). Time-base counter has 3 modes of operation selected by the time-base control register (TBCTL):

- Up-Down-Count Module: In up-down-count mode, the time-base counter starts from zero and increments until the period (TBPRD) value is reached. When the period value is reached, the time-base counter then decrements until it reaches zero. At this point the counter repeats the pattern and begins to increment.
- Up-Count Mode: In this mode, the time-base counter starts from zero and increments until it reaches the value in the period register (TBPRD). When the period value is reached, the time-base counter resets to zero and begins to increment once again.
- Down-Count Mode: In down-count mode, the time-base counter starts from the period (TBPRD) value and decrements until it reaches zero. When it reaches zero, the time-base counter is reset to the period value and it begins to decrement once again.

NOTE: In the section ahead, while discussing the conventional coding task related to ePWM, Up-Down Count Mode has been kept in focus while configuring the registers.

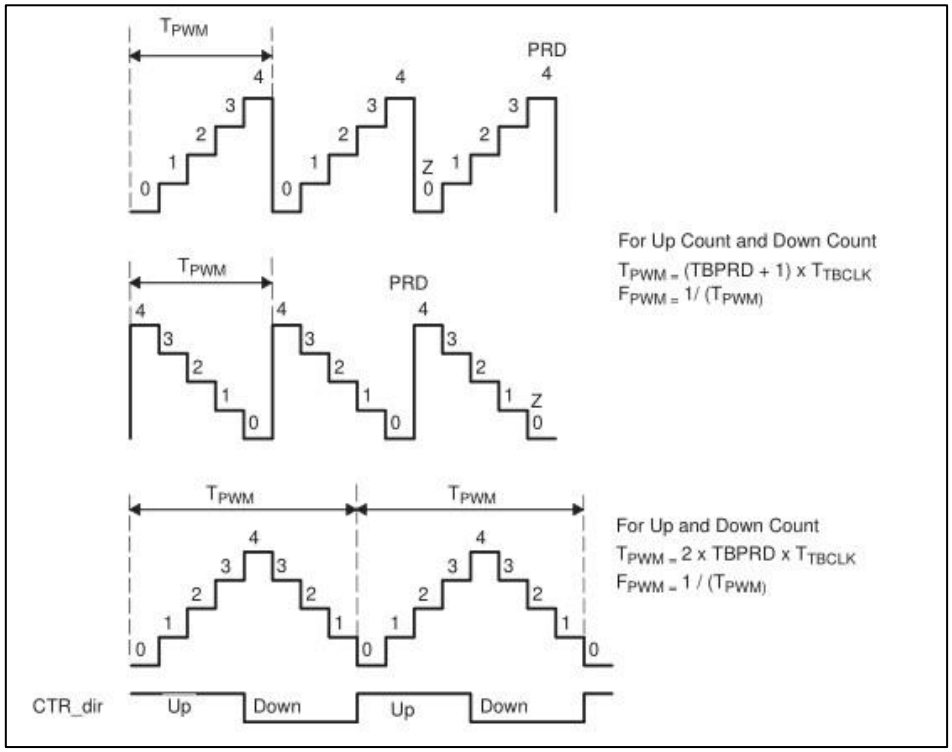


Fig. 15. Calculation of ePWM Frequency and Period

4.2.4 Counter-Compare Submodule

The counter-compare submodule takes as input the time-base counter value. This value is continuously compared to the counter-compare A (CMPA) and counter-compare B (CMPB) registers. When the time-base counter is equal to one of the compare registers, the counter compare unit generates an appropriate event. The counter-compare:

- Generates events based on programmable time stamps using the CMPA and CMPB registers
- Controls the PWM duty cycle if the action-qualifier submodule is configured appropriately

- **Counter Compare A Register [CMPA]**

  Time-base counter equals counter-compare A register (TBCTR = CMPA).

- **Counter Compare B Register [CMPB]**

  Time-base counter equals counter-compare B register (TBCTR = CMPB)

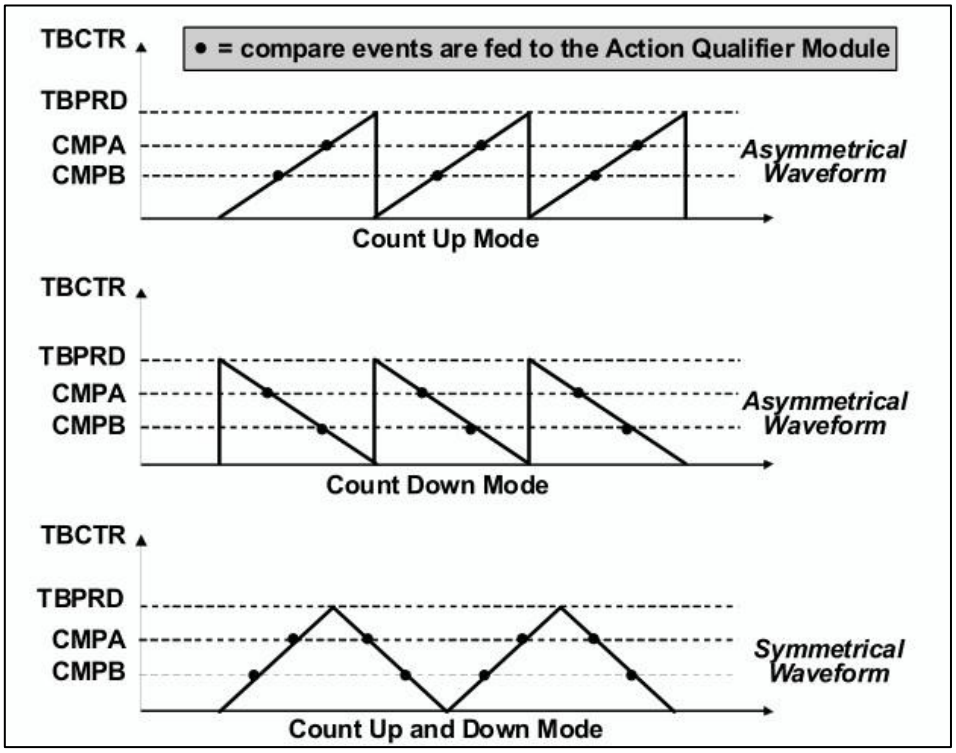


Fig. 16. ePWM Compare Event Waveforms

NOTE: Some controllers such as the one used here include a hardware extension that allows more precise control of the PWM outputs. This extension is the high-resolution pulse width modulator (HRPWM). It provides PWM resolution (time granularity) which is significantly better than what can be achieved using conventionally derived digital PWM methods. There are several other ePWM related sub-modules that have been discussed in great depth in the target reference manual.

4.2.5 Generation of ePWM Output

Task 3: Herewith, the task is focused around generating three different ePWM signals of the same frequency and amplitude and observe the them on a digital oscilloscope. In this case, ePWM1A, ePWM2A and ePWM3A are taken as the output peripherals. Programming ePWM module in this manner is analogous to better understanding the Aircraft Wheel Dynamics Simulator. For pins that can function as ePWM output pins the internal pullup resistors are disabled by default, so they are enabled by setting the GPIO bits to 1 followed by setting up the ePWM peripherals. As per program requirements, values of timer-based period register and timer-based counter register are set for the desirable freq., count mode, phase etc.

Code Flow

- Start
- Include Necessary Header files and declare all function prototypes
- Enter Main Function
- Initialize system control registers, PIE control registers and PIE Vector Table
- Function call to select ePWM Peripherals
  - Set GPIO0, GPIO02 and GPIO04 bits in Pull-up disable Register to 1
  - Set GPIO0, GPIO02 and GPIO04 bits in MUX register to 1 for ePWM1A, ePWM2A and ePWM3A
- Exit function
- Function call to setup ePWM1A
  - Set timer period
  - Set Phase to 0
  - Clear counter
  - Set Mode to Up_down
  - Disable Phase loading
  - Set High Speed time pre-scalar and time-based clock pre-scalar to 2
  - Set Compare A register value to half the timer period value
  - Set Action bits in Action Qualifier Control Register
- Exit Function
- Repeat above steps for two more function calls of ePWM2A and ePWM3A respectively
- Enable Global Interrupt and Global Real-time interrupt
- Exit Main Function
- Stop

Code Composer Studio Code Snippet

```
#include "DSP28x_Project.h"
void Gpio_select(void);
void Setup_ePWM1(void);
void Setup_ePWM2(void);
void Setup_ePWM3(void);

void main(void)
{
    InitSysCtrl();
    DINT;
    InitPieCtrl();
    InitPieVectTable();
    Gpio_select();
    Setup_ePWM1();
    Setup_ePWM2();
    Setup_ePWM3();
    EINT;       // Enable Global interrupt INTM
    ERTM;       // Enable Global realtime interrupt DBGM
    while(1);
}

void Gpio_select(void)
{
    EALLOW;
    GpioCtrlRegs.GPAPUD.bit.GPIO0=1;
    GpioCtrlRegs.GPAPUD.bit.GPIO2=1;
    GpioCtrlRegs.GPAPUD.bit.GPIO4=1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO0=1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO2=1;
    GpioCtrlRegs.GPAMUX1.bit.GPIO4=1;
    EDIS;
}

void Setup_ePWM1(void)
{
    EPwm1Regs.TBPRD = 930;                          // Set timer period
    EPwm1Regs.TBPHS.half.TBPHS = 0x0000;            // Phase is 0
    EPwm1Regs.TBCTR = 0x0000;                       // Clear counter

    EPwm1Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN;  // Count up_down
    EPwm1Regs.TBCTL.bit.PHSEN = TB_DISABLE;         // Disable phase loading
    EPwm1Regs.TBCTL.bit.HSPCLKDIV = TB_DIV4;        // Clock ratio to SYSCLKOUT
    EPwm1Regs.TBCTL.bit.CLKDIV = TB_DIV4;
    EPwm1Regs.CMPA.half.CMPA = 465;

    // Set actions
    EPwm1Regs.AQCTLA.bit.CAU = AQ_SET;
    EPwm1Regs.AQCTLA.bit.CAD = AQ_CLEAR;
}

void Setup_ePWM2(void)
{
    EPwm2Regs.TBPRD = 930;
    EPwm2Regs.TBPHS.half.TBPHS = 0x0000;
    EPwm2Regs.TBCTR = 0x0000;

    EPwm2Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN;
    EPwm2Regs.TBCTL.bit.PHSEN = TB_DISABLE;
    EPwm2Regs.TBCTL.bit.HSPCLKDIV = TB_DIV4;
    EPwm2Regs.TBCTL.bit.CLKDIV = TB_DIV4;
    EPwm2Regs.CMPA.half.CMPA = 465;

    // Set actions
    EPwm2Regs.AQCTLA.bit.CAU = AQ_SET;
    EPwm2Regs.AQCTLA.bit.CAD = AQ_CLEAR;
}

void Setup_ePWM3(void)
{
    EPwm3Regs.TBPRD = 930;
    EPwm3Regs.TBPHS.half.TBPHS = 0x0000;
    EPwm3Regs.TBCTR = 0x0000;

    EPwm3Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN;
    EPwm3Regs.TBCTL.bit.PHSEN = TB_DISABLE;
    EPwm3Regs.TBCTL.bit.HSPCLKDIV = TB_DIV4;
    EPwm3Regs.TBCTL.bit.CLKDIV = TB_DIV4;
    EPwm3Regs.CMPA.half.CMPA = 310;

    // Set actions
    EPwm3Regs.AQCTLA.bit.CAU = AQ_SET;
    EPwm3Regs.AQCTLA.bit.CAD = AQ_CLEAR;
}
```

4.2.6 ePWM Signal Generation with LED flickering using Interrupts

Task 4: The task is similar to the one discussed in section 4.2.5 above with the exception of interrupt handling which also has been discussed previously in section 4.2.2 in detail. The flow of the program goes as such that the LEDs connected to GPIO31 and GPIO34 have to be programmed so as to flicker them at a time period of 250µs but using the CPU Timer 0 Interrput. For the purpose, CPU config timer function is utilized which initializes the selected timer to the period specified by its parameters. Meanwhile, the ePWM output signals should be undisturbed and be observable on the scope.

Code Flow

- Start
- Include Necessary Header files and declare all function prototypes and ISRs
- Enter Main Function
- Initialize system control registers, PIE control registers and PIE Vector Table
- Reset Interrupt Enable and Interrupt Flag Registers
- Function call to select ePWM Peripherals
    - Set GPIO0, GPIO02 and GPIO04 bits in Pull-up disable Register to 1
    - Set GPIO0, GPIO02 and GPIO04 bits in MUX register to 1 for ePWM1A, ePWM2A and ePWM3A
- Exit function
- Function call to setup ePWM1A
    - Set timer period
    - Set Phase to 0
    - Clear counter
    - Set Mode to Up_down
    - Disable Phase loading
    - Set High Speed time pre-scalar and time-based clock pre-scalar to 2
    - Set Compare A register value to half the timer period value
    - Set Action bits in Action Qualifier Control Register
- Exit Function
- Repeat above steps for two more function calls of ePWM2A and ePWM3A respectively
- Point Timer Interrupt 0 present in PIE Vector Table to the ISR Base Address
- Initialize CPU Timers

- Configure CPU timer for 250µs period and 150Mhz frequency
- Set the Interrupt Flag bit in Timer Control Register
- Configure MUX registers and Direction Registers to set peripherals to the required GPIO31 and GPIO34 and direction as output
- Initialize Interrupt Enable Register to 1
- To enable Timer Interrupt 0, set bit 7 of PIE INT1 Interrupt Enable Register to 1
- Enable Global Interrupt and Global Real-time Interrupt
- Loop Forever
- Exit Main Function
- Stop

Code Composer Studio Code Snippet

```
#include "DSP28x_Project.h"

void Gpio_PWM_select(void);
void Setup_ePWM1(void);
void Setup_ePWM2(void);
void Setup_ePWM3(void);
void Gpio_TimedLED_select(void);
__interrupt void cpu_timer0_isr(void);

void main(void)
{
    InitSysCtrl();
    DINT;
    InitPieCtrl();
    IER = 0x0000;
    IFR = 0x0000;
    InitPieVectTable();

    Gpio_PWM_select();
    Setup_ePWM1();
    Setup_ePWM2();
    Setup_ePWM3();
    EALLOW;
    PieVectTable.TINT0 = &cpu_timer0_isr;
    EDIS;

    InitCpuTimers();
    ConfigCpuTimer(&CpuTimer0, 150, 250000);
    CpuTimer0Regs.TCR.all = 0x4000;

    Gpio_TimedLED_select();

    IER |= M_INT1;
    // Enable TINT0 in the PIE: Group 1 interrupt 7
    PieCtrlRegs.PIEIER1.bit.INTx7 = 1;
    EINT;       // Enable Global interrupt INTM
    ERTM;       // Enable Global realtime interrupt DBGM

    while(1);
}
```

```
41
42 void Gpio_TimedLED_select(void)
43 {
44     EALLOW;
45     GpioCtrlRegs.GPBMUX1.bit.GPIO34 = 0;
46     GpioCtrlRegs.GPBDIR.bit.GPIO34 = 1;
47     EDIS;
48     EALLOW;
49     GpioCtrlRegs.GPAMUX2.bit.GPIO31 = 0;
50     GpioCtrlRegs.GPADIR.bit.GPIO31 = 1;
51     EDIS;
52 }
53
54 void Gpio_PWM_select(void)
55 {
56     EALLOW;
57     GpioCtrlRegs.GPAPUD.bit.GPIO0=1;
58     GpioCtrlRegs.GPAPUD.bit.GPIO2=1;
59     GpioCtrlRegs.GPAPUD.bit.GPIO4=1;
60     GpioCtrlRegs.GPAMUX1.bit.GPIO0=1;
61     GpioCtrlRegs.GPAMUX1.bit.GPIO2=1;
62     GpioCtrlRegs.GPAMUX1.bit.GPIO4=1;
63     EDIS;
64 }
65
66 void Setup_ePWM1(void)
67 {
68     EPwm1Regs.TBPRD = 930;                          // Set timer period
69     EPwm1Regs.TBPHS.half.TBPHS = 0x0000;            // Phase is 0
70     EPwm1Regs.TBCTR = 0x0000;                       // Clear counter
71
72     EPwm1Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // Count up
73     EPwm1Regs.TBCTL.bit.PHSEN = TB_DISABLE;         // Disable phase loading
74     EPwm1Regs.TBCTL.bit.HSPCLKDIV = TB_DIV4;        // Clock ratio to SYSCLKOUT
75     EPwm1Regs.TBCTL.bit.CLKDIV = TB_DIV4;
76     EPwm1Regs.CMPA.half.CMPA = 465;
77
78     EPwm1Regs.AQCTLA.bit.CAU = AQ_SET;              // Set PWM1A on Zero
79     EPwm1Regs.AQCTLA.bit.CAD = AQ_CLEAR;
80 }
81
82 void Setup_ePWM2(void)
83 {
84     EPwm2Regs.TBPRD = 930;
85     EPwm2Regs.TBPHS.half.TBPHS = 0x0000;
86     EPwm2Regs.TBCTR = 0x0000;
87
88     EPwm2Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN;
89     EPwm2Regs.TBCTL.bit.PHSEN = TB_DISABLE;
90     EPwm2Regs.TBCTL.bit.HSPCLKDIV = TB_DIV4;
91     EPwm2Regs.TBCTL.bit.CLKDIV = TB_DIV4;
92     EPwm2Regs.CMPA.half.CMPA = 465;
93
94     EPwm2Regs.AQCTLA.bit.CAU = AQ_SET;
95     EPwm2Regs.AQCTLA.bit.CAD = AQ_CLEAR;
96 }
97
98 void Setup_ePWM3(void)
99 {
100     EPwm3Regs.TBPRD = 930;
101     EPwm3Regs.TBPHS.half.TBPHS = 0x0000;
102     EPwm3Regs.TBCTR = 0x0000;
103
104     EPwm3Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN;
105     EPwm3Regs.TBCTL.bit.PHSEN = TB_DISABLE;
106     EPwm3Regs.TBCTL.bit.HSPCLKDIV = TB_DIV4;
107     EPwm3Regs.TBCTL.bit.CLKDIV = TB_DIV4;
108     EPwm3Regs.CMPA.half.CMPA = 310;
109
110     EPwm3Regs.AQCTLA.bit.CAU = AQ_SET;
111     EPwm3Regs.AQCTLA.bit.CAD = AQ_CLEAR;
112 }
113
114 __interrupt void cpu_timer0_isr(void)
115 {
116     CpuTimer0.InterruptCount++;
117     GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1;
118     GpioDataRegs.GPATOGGLE.bit.GPIO31 = 1;
119
120     PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
121 }
```

4.3 Analog to Digital Converter [ADC] Module

The world around us is analog in nature. Almost every environmental measurable parameter is in analog form like temperature, sound, pressure, etc. and any system needs an intermediate device to convert the analog temperature data into digital data in order to communicate with digital processors like microcontrollers and microprocessors. Analog to Digital Converter (ADC) is that very electronic integrated circuit needed to convert the analog signals such as voltages to digital or binary form consisting of 1s and 0s.

The TMS320x2833x ADC module is a 12-bit pipelined analog-to-digital converter (ADC). This section explains the operation of the analog-to-digital converter. The system consists of a 12-bit analog-to-digital converter with 16 analog input channels. The analog input channels have a range from 0 to 3 volts. Two input analog multiplexers are used, each supporting 8 analog input channels. Each multiplexer has its own dedicated sample and hold circuit. Therefore, sequential, as well as simultaneous sampling is supported. Also, the ADC system features programmable auto sequence conversions with 16 results registers. Start of conversion (SOC) can be performed by an external trigger, software, or an ePWM event.

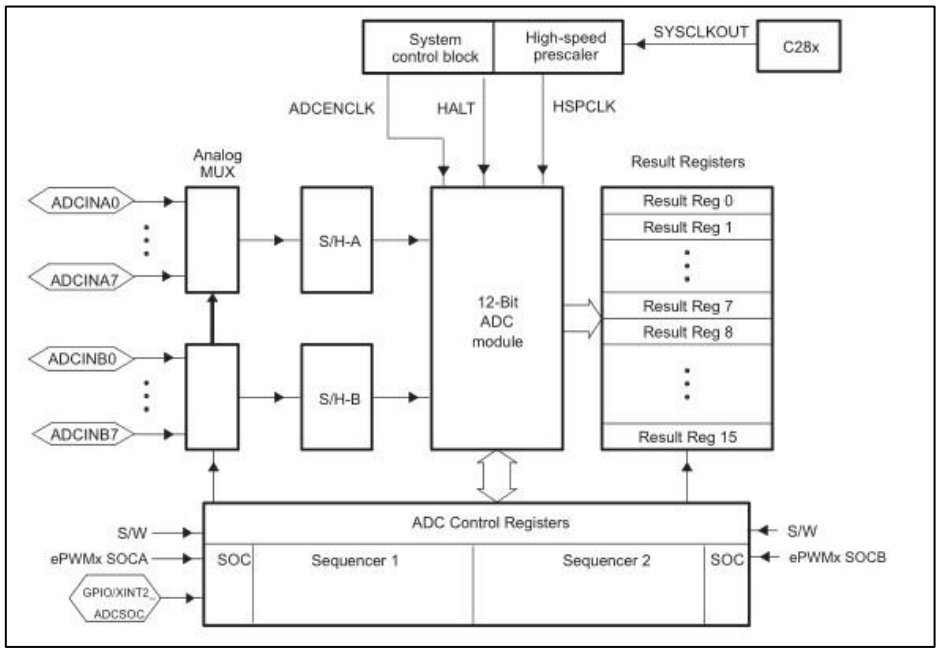


Fig. 17. ADC Block Diagram

The following sub-sections form together the physical implementation of C28x ADC circuit:

- Clocking
- Sample and Hold Circuitry
- Reference Selection
- Power Modes and Sequencing
- Calibration and Offset Correction

4.3.1 Operation & Functions

As mentioned earlier, the ADC module has 16 channels, configurable as two independent 8-channel modules. The two independent 8-channel modules can be cascaded to form a 16-channel module. Although there are multiple input channels and two sequencers, there is only one converter in the ADC module. The two 8-channel modules can autosequence a series of conversions; each module has the choice of selecting any one of the respective eight channels available through an analog MUX. In the cascaded mode, the autosequencer functions as a single 16-channel sequencer. On each sequencer, once the conversion is completed, the selected channel value is stored in its respective ADCRESULT register. Autosequencing allows the system to convert the same channel multiple times, allowing the user to perform oversampling algorithms. This oversampling gives increased resolution over traditional single-sampled conversion results.

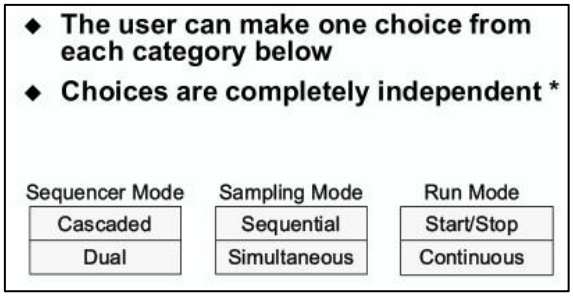


Fig. 18. ADC Operating Modes

Functions of the ADC module include:

- 12-bit ADC core with built-in dual sample-and-hold (S/H) and input from 0 V to 3 V
- Simultaneous sampling or sequential sampling modes and 16-channel, multiplexed inputs
- Fast conversion time runs at 25 MHz, ADC clock, or 12.5 MSPS
- Autosequencing capability provides up to 16 "autoconversions" in a single session. Each conversion can be programmed to select any 1 of 16 input channels.
- Sequencer can be operated as two independent 8-state sequencers or as one large 16-state
- sequencer (that is, two cascaded 8-state sequencers).
- Sixteen result registers (individually addressable) to store conversion [digital] values.
- Multiple triggers as sources for the start-of-conversion (SOC) sequence
  - S/W - software immediate start
  - ePWM 1-6
  - GPIO XINT2
- Flexible interrupt control allows interrupt request on every end-of-sequence (EOS) or every other EOS

- Sequencer can operate in "start/stop" mode, allowing multiple "time-sequenced triggers" to synchronize conversions.
- ePWM triggers can operate independently in dual-sequencer mode.
- Sample-and-hold (S/H) acquisition time window has separate prescale control.

The formulation of calculating the digital value corresponding to the value given as the input analog voltage is shown in the figure below:

$$\text{Digital Value} = 0, \quad \text{when input} \le 0\ \text{V}$$

$$\text{Digital Value} = 4096 \times \frac{\text{Input Analog Voltage} - \text{ADCLO}}{3}, \quad \text{when } 0\ \text{V} < \text{input} < 3\ \text{V}$$

$$\text{Digital Value} = 4095, \quad \text{when input} \ge 3\ \text{V}$$

Fig. 19. ADC Value Calc.

4.3.2 ADC Clocking

The peripheral clock HSPCLK is divided down by the ADCCLKPS[3:0] bits of the ADCTRL3 register. An extra divide-by-two is provided via the CPS bit of the ADCTRL1 register. In addition, the ADC can be tailored to accommodate variations in source impedances by widening the sampling/acquisition period.

This is controlled by the ACQ_PS[3:0] bits in the ADCTRL1 register. These bits do not affect the conversion portion of the S/H and conversion process, but do extend the length of time in which the sampling portion takes by extending the start of the conversion pulse. Moreover, The ADC module has several prescaler stages to generate any desired ADC operating clock speed.

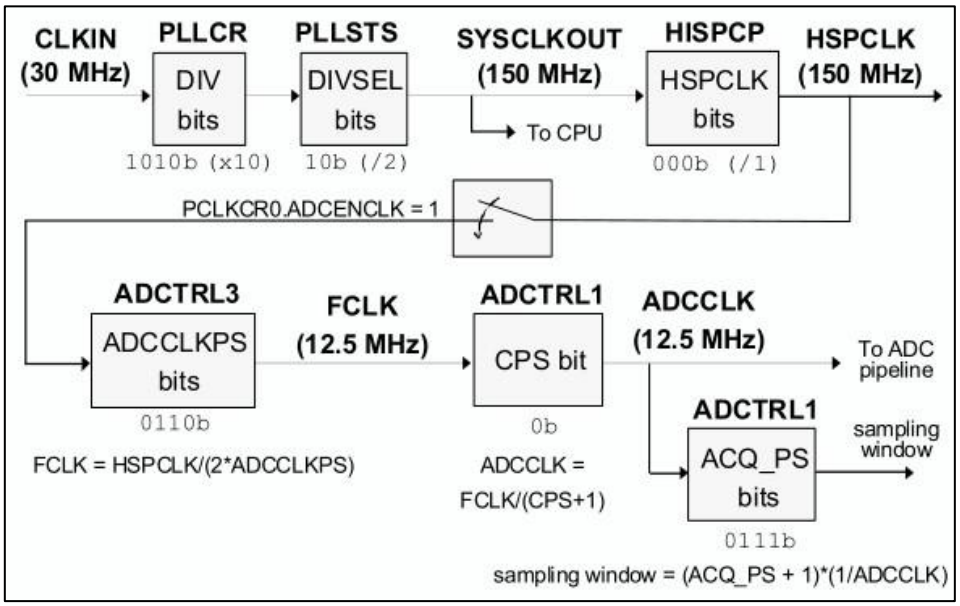


Fig. 20. ADC Clocking Flow

4.3.3 ADC Register Configuration

Table shown belows lists the memory-mapped registers for the ADC.

Register	Description
ADCTRL1	ADC Control Register 1
ADCTRL2	ADC Control Register 2
ADCTRL3	ADC Control Register 3
ADCMAXCONV	ADC Maximum Conversion Channels Register
ADCCHSELSEQ1	ADC Channel Select Sequencing Control Register 1
ADCCHSELSEQ2	ADC Channel Select Sequencing Control Register 2
ADCCHSELSEQ3	ADC Channel Select Sequencing Control Register 3
ADCCHSELSEQ4	ADC Channel Select Sequencing Control Register 4
ADCASEQSR	ADC Autosequence Status Register
ADCRESULT0	ADC Conversion Result Buffer Register 0
ADCRESULT1	ADC Conversion Result Buffer Register 1
ADCRESULT2	ADC Conversion Result Buffer Register 2
⋮	⋮
ADCRESULT14	ADC Conversion Result Buffer Register 14
ADCRESULT15	ADC Conversion Result Buffer Register 15
ADCREFSEL	ADC Reference Select Register
ADCOFFTRIM	ADC Offset Trim Register
ADCST	ADC Status and Flag Register

Fig. 21. ADC Registers Table

There are many registers associated with ADC in C28x but some of the important ones that are key and vital to the ADC Programming are discussed in the sections below.

- **ADC Control Register 1 [ADCTRL1]**

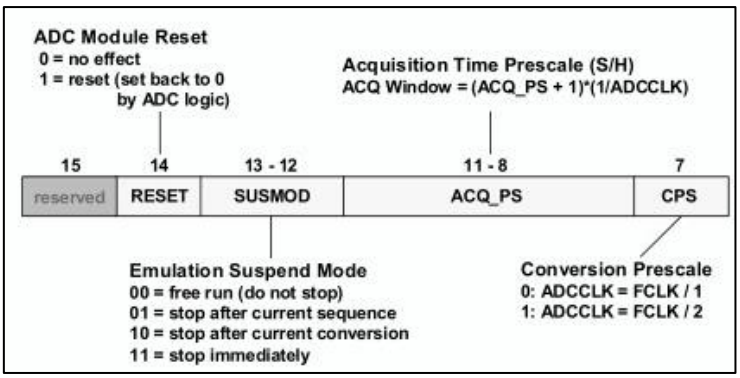


Fig. 22(a). ADCTRL1 [Higher Order Bits]

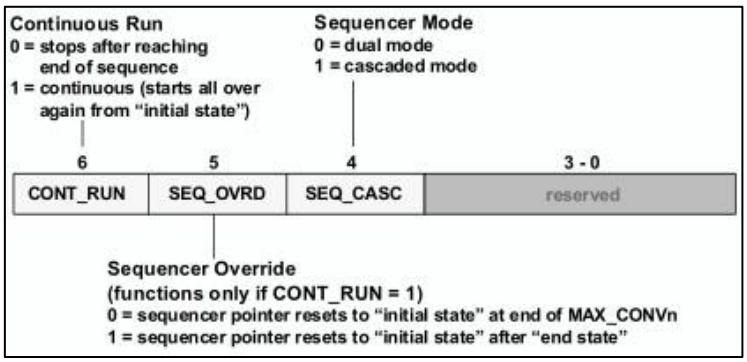


Fig. 22(b). ADCTRL1 [Lower Order Bits]

- **ADC Control Register 2 [ADCTRL2]**

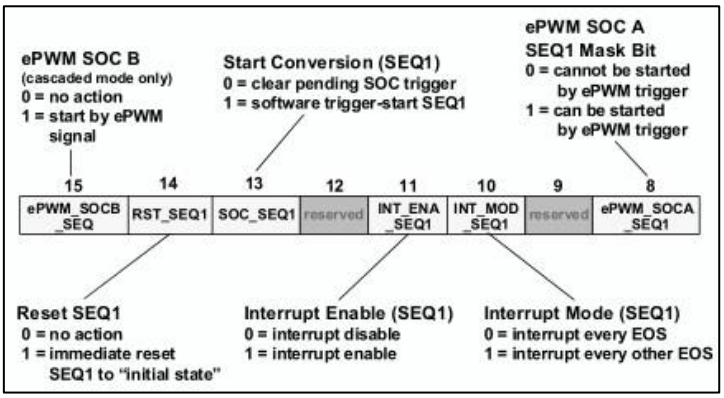


Fig. 23(a). ADCTRL2 [Higher Order Bits]

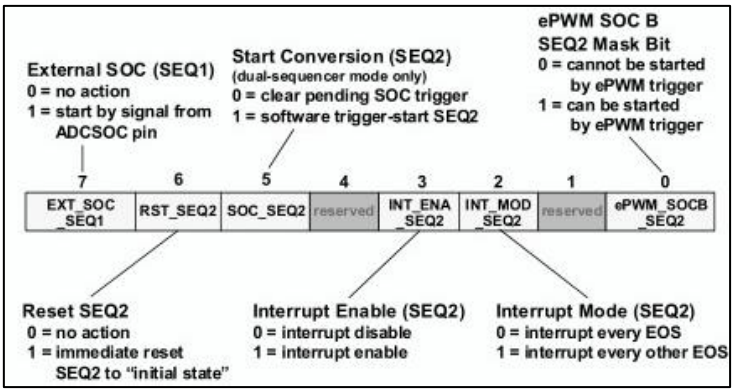


Fig. 23(b). ADCTRL2 [Lower Order Bits]

NOTE: More information can be found out about the ADCTRL3 Register in the F28335 target reference manual.

- **Max Conversion Channel Register [ADCMAXCONV]**

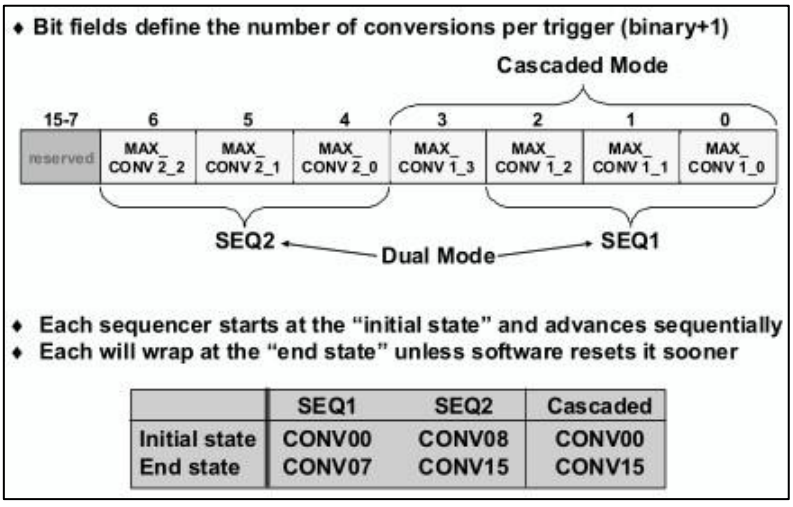


Fig. 24. ADCMAXCONV

- **Input Channel Select Sequencing Control Register [ADCCHSELSEQx]**

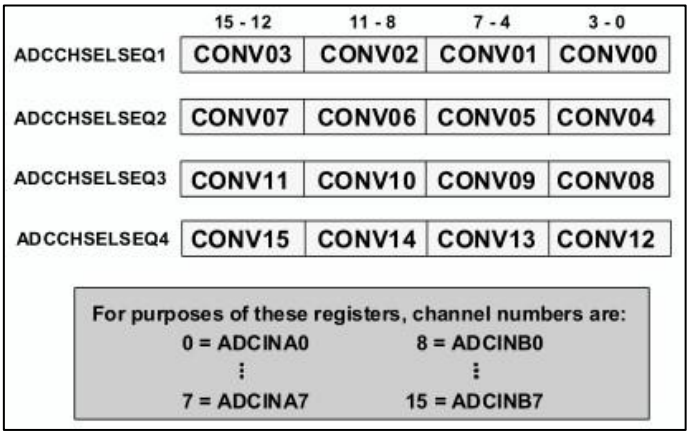
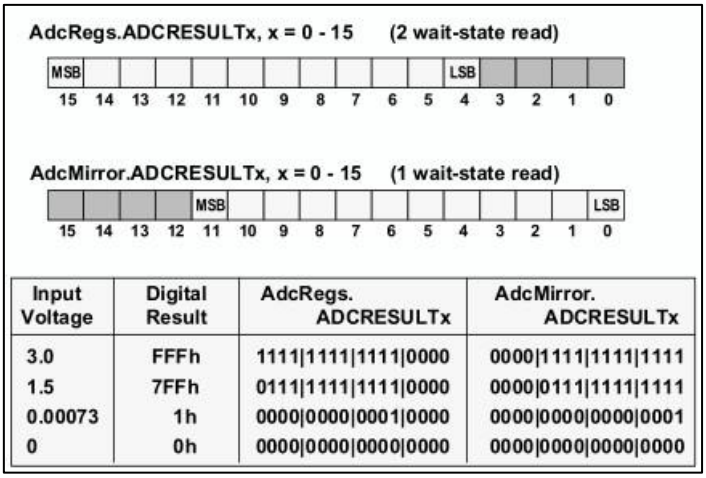


Fig. 25. ADCCHSELSEQx

- **ADC Conversion Result Registers [ADCRESULTx]**



Input Voltage	Digital Result	AdcRegs. ADCRESULTx	AdcMirror. ADCRESULTx
3.0	FFFh	1111|1111|1111|0000	0000|1111|1111|1111
1.5	7FFh	0111|1111|1111|0000	0000|0111|1111|1111
0.00073	1h	0000|0000|0001|0000	0000|0000|0000|0001
0	0h	0000|0000|0000|0000	0000|0000|0000|0000

Fig. 26. ADCRESULT1 and ADCRESULT2

4.3.4 ADC Interrupt Operation during SEQ Conversion

There are two ADC interrupt signals that correspond to each sequencer, SEQ1 INT and SEQ2 INT. Each interrupt can be selectively enabled/disabled by its corresponding INT_ENA_SEQ# bit in the ADCTRL2 register. The status(active = 1) of each interrupt is presented in the ADCST register as INT_SEQ2 and INT_SEQ1 bits respectively.

Once an interrupt is latched in either INT_SEQ1 or INT_SEQ2, it must be cleared in order for another interrupt to be generated to the C28x CPU/PIE module. The interrupt clear bit is also located in ADCST as INT_SEQ1_CLR and INT_SEQ2_CLR. Finally, cascaded sequencer mode only uses SEQ1 INT, as this mode treats SEQ1 as the lone 16-state sequencer for the ADC.

The sequencer can generate interrupts under two operating modes. These modes are determined by the Interrupt-Mode-Enable control bits in ADCTRL2.

- Case 1: Number of samples in the first and second sequences are not equal

  Mode 1 Interrupt operation (that is, Interrupt request occurs at every EOS)

  1. Sequencer is initialized with MAX_CONVn = 1 for converting I1 and I2
  2. At ISR "a", MAX_CONVn is changed to 2 (by software) for converting V1, V2, and V3
  3. At ISR "b", the following events take place :
     a. MAX_CONVn is changed to 1 again for converting I1 and I2.
     b. Values I1, I2, V1, V2, and V3 are read from ADC result registers.
     c. The sequencer is reset.
  4. Steps 2 and 3 are repeated. Note that the interrupt flag is set every time SEQ_CNTR reaches zero and both interrupts are recognized.

- **Case 2**: Number of samples in the first and second sequences are equal

  Mode 2 Interrupt operation (that is, Interrupt request occurs at every other EOS)

  1. Sequencer is initialized with MAX_CONVn = 2 for converting I1, I2, and I3 (or V1, V2, and V3).
  2. At ISR "b" and "d", the following events take place :
     a. Values I1, I2, I3,V1, V2, and V3 are read from ADC result registers.
     b. The sequencer is reset.
  3. Step 2 is repeated.

- **Case 3**: Number of samples in first and second sequences are equal (with dummy read)

  Mode 2 Interrupt operation (that is, Interrupt request occurs at every other EOS)

  1. Sequencer is initialized with MAX_CONVn = 2 for I1, I2, and x(dummy sample).
  2. At ISR "b" and "d", the following events take place:
     a. Values I1, I2, x,V1, V2, and V3 are read from ADC result registers.
     b. The sequencer is reset.
  3. Step 2 is repeated. Note that the third I-sample (x) is a dummy sample, and is not really required.

However, to minimize ISR overhead and CPU intervention, advantage is taken of the "every other" Interrupt request feature of Mode 2.

4.3.5 Generate periodic ADC SOC on SEQ1 using ePWM

Task 4: The objective of this task is to become familiar with the programming and operation of the on-chip analog-to-digital converter. This will help in knowing how ADC SOC [start of conversion] and ePWM module are interconnected in the target. The purpose here is to vary the duty cycle of the ePWM signal depending upon the input voltage being read from a potentiometer. A potentiometer is an analog electronic device that can act as a variable resistor in which the voltage can change within a range of values depending upon the direction that the variable knob is rotated in.

The MCU will be setup to sample a single ADC input channel at a prescribed sampling rate and store the conversion result in a memory buffer. This buffer will operate in a circular fashion, such that new conversion data continuously overwrites older results in the buffer.
The key thing is that two different ADC channels are to be setup as input for 1st and 2nd SEQ1 conversion respectively. The potentiometer can be given to any of these channels to get the converted digital value and then that value is to determine how the ePWM duty cycle would vary. The most important thing is to enable SOCA [start of conversion A] from ePWM to start SEQ1. Interrupt service routine would be invoked during sequence conversion [more on this discussed in the 4.3.4 section above] and inside the ISR, there would be an array of maximum size 10 to store sampled values from ADC Result register.

As mentioned, one of the ePWMs (ePWM1) will be configured to automatically trigger the SOCA signal at the desired sampling rate. The ADC end-of-conversion interrupt will be used to prompt the CPU to copy the results of the ADC conversion into a results buffer in memory. This buffer pointer will be managed in a circular fashion, such that new conversion results will continuously overwrite older conversion results in the buffer. Additionally, the LED connected to the GPIO31 on the controlCARD will glow as a visual indication to when the knob of the potentiometer is at maximum voltage. It is to be noted that the second ePWM [ePWM2] would be used and configured to observe the changes in the duty cycle of its corresponding output signal depending upon the voltage coming from the potentiometer.

One aspect that is common in the evaluation of any ADC, discrete or integrated, is its conversion speed. This is how long it takes the ADC to convert the sampled analog signal into a digital representation. This is especially important in high bandwidth systems as it is beneficial to update the system intracycle vs waiting for the next full cycle of the control loop

to happen. The ADC on the C2000 MCU is capable of generating a conversion in as little as 260ns after the sample completes. This allows enough time for the C28x or CLA core to calculate the new PWM frequencies well ahead of the next control loop period allowing for the intra-cycle update to happen. The ePWM module plays a role here as well, allowing the PWM to be updated by writing to the active registers vs the shadow register which would pend the update until the next full period.

Code Flow

- Start
- Include Necessary Header files and declare all function prototypes, ISRs
- Initialize Global Variables LoopCount, ConversionCount, AdcResult0 AdcResult1 and global arrays Voltage1, Voltage2 each of size 10
- Enter Main Function
- Initialize system control registers and CPU clock frequency
- Define ADC Clock Frequency (less than or equal to 25 Mhz)
- Initialize PIE control registers and PIE Vector Table
- Reset Interrupt Enable and Interrupt Flag Registers
- Function call to select ePWM Peripheral
  - Set GPIO02 bit in Pull-up disable Register to 1
  - Set GPIO02 bit in MUX register to 1 for ePWM2A
- Exit function
- Function call to setup ePWM2A
  - Set timer period to AdcResult0 value
  - Set Phase to 0 and clear counter
  - Set Count Mode to Up_down
  - Disable Phase loading
  - Set High Speed time pre-scalar and time-based clock pre-scalar to 2 [Clock Ratio to SYSCLKOUT]
  - Set Compare A register value to half the AdcResult0 value
  - Set Action bits in Action Qualifier Control Register to put ePWM2A to 0
- Exit Function
- Point ADC Interrupt present in PIE Vector Table to ADC ISR Base Address
- Initialize ADC using in-built function
- To enable ADCINT in PIE, set bit 7 of PIE INT1 Interrupt Enable Register to 1

- Initialize Interrupt Enable Register to 1
- Enable Global Interrupt and Global Real-time Interrupt
- Declare LoopCount and ConversionCount to 0
- Configure ADC Module
    Setup two conversions on SEQ1
    Setup ADC Input Channel 3A and 2A as 1st and 2nd SEQ1 conversions respectively
- Enable SOCA from ePWM to start SEQ1
    Enable SEQ1 Interrupt (every EOS)
    Enable SOCA using ePWM1
    Select SOC from CMPA on Up_Count
    Generate Pulse on 1st Event
    Set Compare A value
    Set period for ePWM1
    Count up and Start
- Configure MUX register and Direction Register to set peripheral to the required GPIO31 and direction as output
- Loop while condition is true
    Store Conversion Results in ADC Result registers
    if AdcResult0 <= 65500 then
        LED at GPIO31 turned ON
    else
        LED at GPIO31 remains OFF
    Set ePWM2 timer period to AdcResult0 value
- Exit Main Function
- Interrupt Service routine to be invoked as per ADC Interrupt
- Store sampled values from ADC Result Registers at index ConversionCount of Voltage1 and Voltage2 global arrays
- If 40 conversions have been logged, start over
- Reinitialize for next ADC sequence
- Reset SEQ1
- Clear Interrupt SEQ1 bit
- Acknowledge Interrupt to PIE
- Return and exit Interrupt Service Routine
- Stop

Code Composer Studio Code Snippet

```
#include "DSP28x_Project.h"
void Gpio_PWM_select(void);
void Setup_ePWM2();
__interrupt void adc_isr(void);

Uint16 LoopCount;
Uint16 ConversionCount;
Uint16 Voltage1[10];
Uint16 Voltage2[10];
Uint16 AdcResult0;
Uint16 AdcResult1;

void main(void)
{
    InitSysCtrl();
    #if (CPU_FRQ_150MHZ)
    #define ADC_MODCLK 0x3
    #endif

    // Define ADCCLK clock frequency ( less than or equal to 25 MHz )
    // Assuming InitSysCtrl() has set SYSCLKOUT to 150 MHz
    EALLOW;
    SysCtrlRegs.HISPCP.all = ADC_MODCLK;
    EDIS;
    DINT;
    InitPieCtrl();
    IER = 0x0000;
    IFR = 0x0000;
    InitPieVectTable();
    Gpio_PWM_select();
    Setup_ePWM2();

    EALLOW;
    PieVectTable.ADCINT = &adc_isr;
    EDIS;
    InitAdc();  // For this example, init the ADC
    // Enable ADCINT in PIE
    PieCtrlRegs.PIEIER1.bit.INTx6 = 1;
    IER |= M_INT1;
    EINT;
    ERTM;

    LoopCount = 0;
    ConversionCount = 0;
```

```
45
46      // Configure ADC
47      AdcRegs.ADCMAXCONV.all = 0x0001;        // Setup 2 conv's on SEQ1
48      AdcRegs.ADCCHSELSEQ1.bit.CONV00 = 0x3;  // Setup ADCINA3 as 1st SEQ1 conv.
49      AdcRegs.ADCCHSELSEQ1.bit.CONV01 = 0x2;  // Setup ADCINA2 as 2nd SEQ1 conv.
50      // Enable SOCA from ePWM to start SEQ1
51      AdcRegs.ADCTRL2.bit.EPWM_SOCA_SEQ1 = 1;
52      AdcRegs.ADCTRL2.bit.INT_ENA_SEQ1 = 1;   // Enable SEQ1 interrupt (every EOS)
53
54      // Assumes ePWM1 clock is already enabled in InitSysCtrl();
55      EPwm1Regs.ETSEL.bit.SOCAEN = 1;         // Enable SOC on A group
56      EPwm1Regs.ETSEL.bit.SOCASEL = 4;        // Select SOC from from CPMA on upcount
57      EPwm1Regs.ETPS.bit.SOCAPRD = 1;         // Generate pulse on 1st event
58      EPwm1Regs.CMPA.half.CMPA = 0x0080;      // Set compare A value
59      EPwm1Regs.TBPRD = 0xFFFF;               // Set period for ePWM1
60      EPwm1Regs.TBCTL.bit.CTRMODE = 0;        // count up and start
61
62      EALLOW;
63      GpioCtrlRegs.GPAMUX2.bit.GPIO31 = 0;
64      GpioCtrlRegs.GPADIR.bit.GPIO31 = 1;
65      EDIS;
66
67      while(1)
68      {
69          // Convert, wait for completion and then
70          // Store conversion results in AdcResult0, AdcResult1
71          AdcResult0 = AdcRegs.ADCRESULT0;
72          AdcResult1 = AdcRegs.ADCRESULT1;
73
74          if(AdcResult0 <= 65500)
75              GpioDataRegs.GPADAT.bit.GPIO31 = 1;
76          else
77              GpioDataRegs.GPADAT.bit.GPIO31 = 0;
78
79          EPwm2Regs.TBPRD = AdcResult0;
80      }
81  }
82
83  __interrupt void adc_isr(void)
84  {
85      Voltage1[ConversionCount] = AdcRegs.ADCRESULT0 >> 4;
86      Voltage2[ConversionCount] = AdcRegs.ADCRESULT1 >> 4;
87      // If 40 conversions have been logged, start over
88      if(ConversionCount == 9)
89      {
90          ConversionCount = 0;
91      }
92      else
93      {
94          ConversionCount++;
95      }
96      // Reinitialize for next ADC sequence
97      AdcRegs.ADCTRL2.bit.RST_SEQ1 = 1;               // Reset SEQ1
98      AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1;             // Clear INT SEQ1 bit
99      PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;         // Acknowledge interrupt to PIE
100
101     return;
102 }
103
104 void Gpio_PWM_select(void)
105 {
106     EALLOW;
107     GpioCtrlRegs.GPAPUD.bit.GPIO2=1;
108     GpioCtrlRegs.GPAMUX1.bit.GPIO2=1;
109     EDIS;
110 }
111
112 void Setup_ePWM2()
113 {
114     EPwm2Regs.TBPRD = AdcResult0;                   // Set timer period
115     EPwm2Regs.TBPHS.half.TBPHS = 0x0000;            // Phase is 0
116     EPwm2Regs.TBCTR = 0x0000;                       // Clear counter
117
118     EPwm2Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // Count mode up_down
119     EPwm2Regs.TBCTL.bit.PHSEN = TB_DISABLE;         // Disable phase loading
120     EPwm2Regs.TBCTL.bit.HSPCLKDIV = TB_DIV4;        // Clock ratio to SYSCLKOUT
121     EPwm2Regs.TBCTL.bit.CLKDIV = TB_DIV4;
122     EPwm2Regs.CMPA.half.CMPA = (AdcResult0/2);
123
124     EPwm2Regs.AQCTLA.bit.CAU = AQ_SET;              // Set ePWM2A on Zero
125     EPwm2Regs.AQCTLA.bit.CAD = AQ_CLEAR;
126 }
```

5. MODEL BASED DEVELOPMENT USING SIMULINK

This section will be focusing on how preference is now being given to Simulink-Model Based Development over the years and how to setup software for target-based development.

NOTE: A more detailed view has been discussed in the Chapter 3, Section 3.4 above.

5.1 Advantages over Conventional

The primary advantage of utilizing MBD is the auto-generation of code, which helps to remove the possibility of human mistakes and makes it possible to reuse code. In addition, businesses have consistently achieved immediate and tangible results by adopting model-based design on an incremental basis. These results include a quicker time to the first demonstration, a quicker time to market with a product that meets quality standards as well as a quicker turnaround of iterations without the requirement of hardware, and so on. Analysis, design, and testing that are conducted on a continual basis increase the efficiency of the development process. This helps in saving the amount of time and money spent on development. Engineers are able to effortlessly adapt to model-based design and operate at even greater levels of speed, competence, and design quality especially when working with embedded systems.

5.2 Simulink Workbench Setup

5.2.1 Solver

The model is represented as a set of ordinary differential equations, and a solver uses a numerical approach to find a solution to these equations. The time of the subsequent simulation step is figured out based on the results of this computation. As the solver works through the process of finding an answer to this starting value issue, it will also ensure that the accuracy constraints provided are met. Fixed-Step Extrapolation is the method used in this scenario since it is a solution for highly difficult problems.

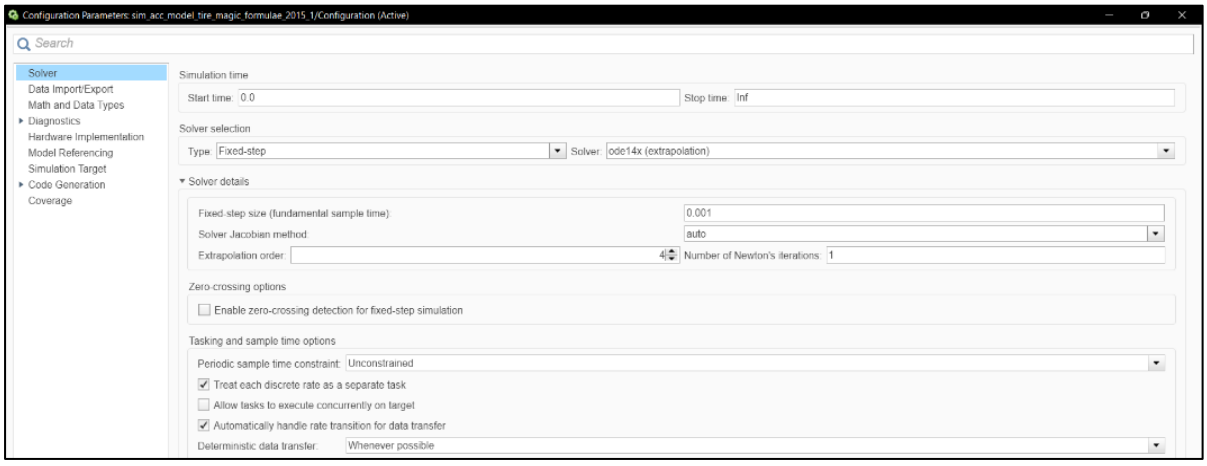


Fig. 1. Solver Configuration Window

5.2.2 Hardware Implementation

The settings necessary to configure a hardware development board so that a model may be deployed onto it can be found included in the tab titled "Hardware Implementation". Hardware implementation parameters detail a variety of possibilities for constructing models that may operate on hardware boards or devices, such as communication connections and hardware-specific characteristics. The parameters provide a description of the hardware and compiler characteristics of the MATLAB program. As mentioned previously in Chapter 2, Section 2.4, TMS320F28335 hardware board is used, which has its byte ordering set to Little Endian.

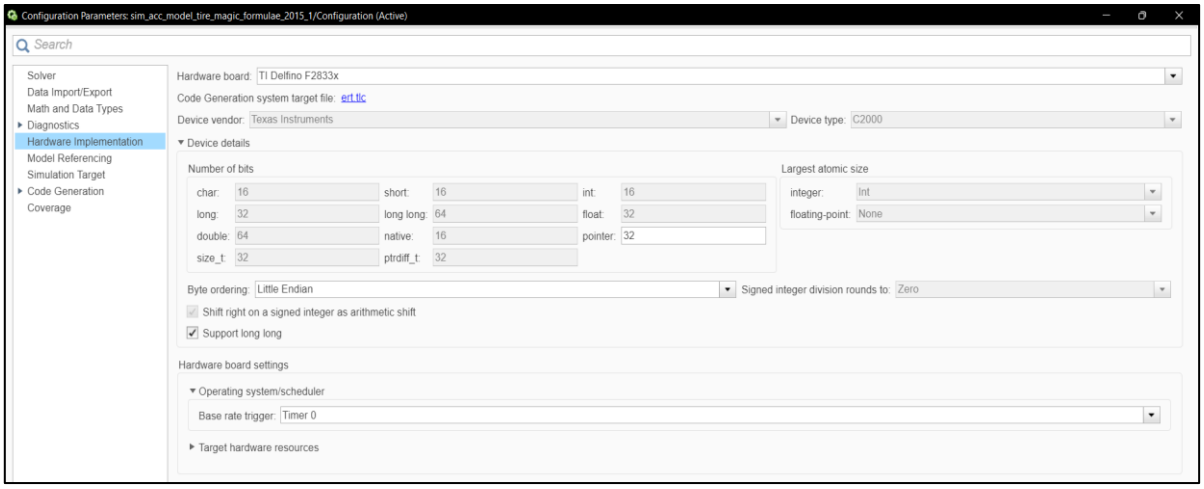


Fig. 2. Hardware Configuration Window

5.2.3 Code Generation

The code generating settings dictate the manner in which the code generator creates code and constructs an executable application based on your model. The Code Generation and pane of the Configuration Parameters dialogue box is where you will find the model configuration parameters that are used for code generation. It is possible for the target that is selected to cause a change in the contents of the Code Generation pane and the sub panes that it contains.

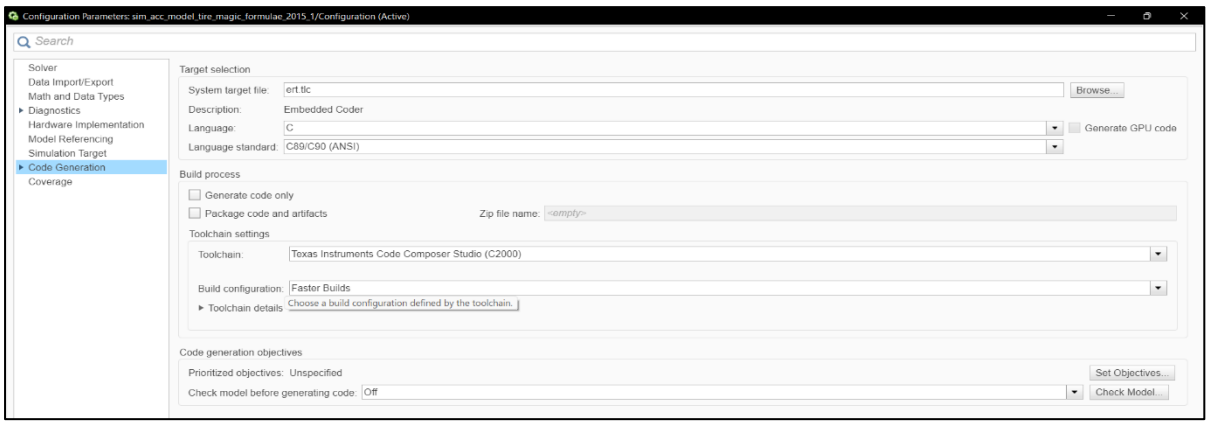


Fig. 3. Code Generation Window

6. MINIATURIZED WHEEL DYNAMICS SIMULATOR

6.1 Model Overview

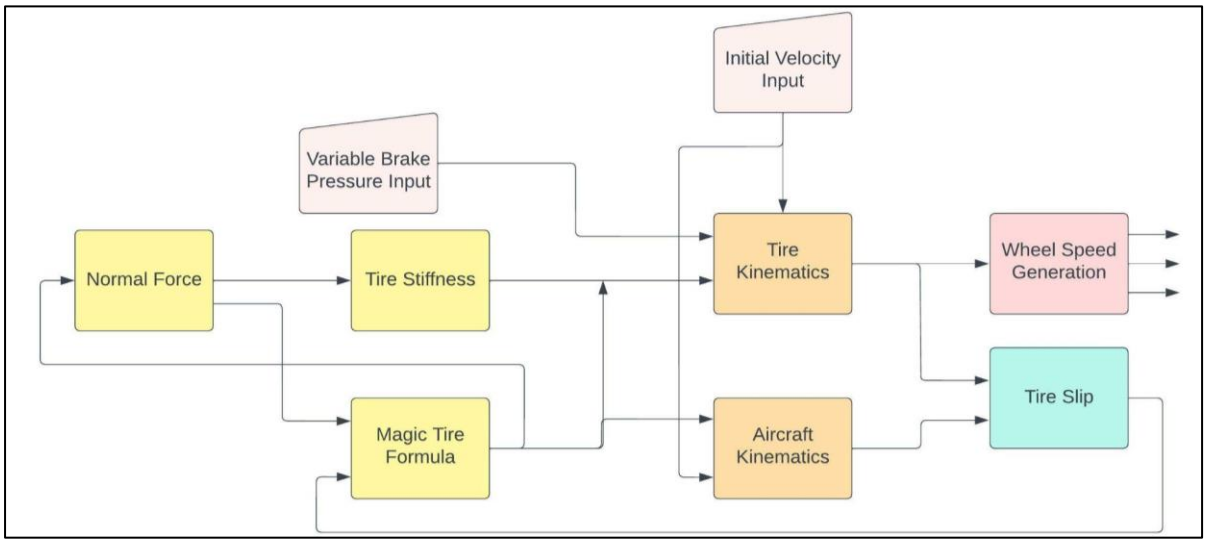


Fig. 1. Miniaturized Wheel Dynamics Simulator Block Diagram

The model discussed here is a representation of how aircraft wheels behave whence interrupted by the brakes and where several natural forces come into the picture. As it can be seen in the block diagram of the model above, there are blocks associated with how for some certain initial velocity taken as input, a certain amount of brake pressure has to be applied. The resultant then underwent through a series of blocks sometimes forming a sort of feedback with the final wheel speed data being read from the model as output. It is quite visible that there are just two inputs being given to the model externally by the user and that can be dynamically varied, that being the brake pressure and initial velocity while the other blocks form more of the computation and the velocity to frequency conversion part of the model.

There are several key components that play a role when an aircraft wheel experiences brakes, be it the kinematics involved, or the tire constants, or the stiffness of tires, kinematics of the aircraft tires or the slip associated with the three tires in the aircraft, that is, nose wheel [front side] and the 2 side wheels [port-side and starboard-side]. These factors when connected in the way shown above can help read and record the speed of these three wheels when a brake is applied to them. An aircraft when in flight is vulnerable to both positive and negative impacts, which might stem from any number of different sources and factors acting upon it at a certain point of time. The model takes into account, not just the elements that have an impact that can be described as having a positive force, but also the factors that have an effect that can be described as having a negative force. With that, the model makes an attempt to realize the goal depicted in fig.1 above, which provides a general block overview of the same.

6.2 Sub-blocks

6.2.1 Pressure-Input using ADC

In the Pressure Output Block of the Aircraft Model, an analog to digital converter, often known as an ADC is used. It is a device that takes an analog signal and transforms it into a digital one. An analog signal could be something like the sound that is picked up by a microphone or the light that is taken in by a digital camera. The voltage that is acquired from the control card of the

TMS320F28335 is used to calibrate a potentiometer, which provides the analog input. In this instance, the voltage serves as the analog input. It is possible to adjust the voltage anywhere from 0 to 3V by using a potentiometer, which is also referred to as a variable resistor. When this value is read by the ADC, it is first transformed into its digital counterpart. Next, the value is multiplied by the step size of the signal in order to obtain the equivalent analog voltage value. MATLAB's gain block is what's needed to get this done. In order to determine the associated pressure, this voltage is converted into its analog equivalent value. In this model, we have mapped 0 volts to 0 bar pressure and 3 volts to 30 bar pressure.

This mapping is accomplished by using the gain block to multiply the voltage with a proportionality constant of 10 in order to obtain the output pressure that corresponds to the varying voltage.

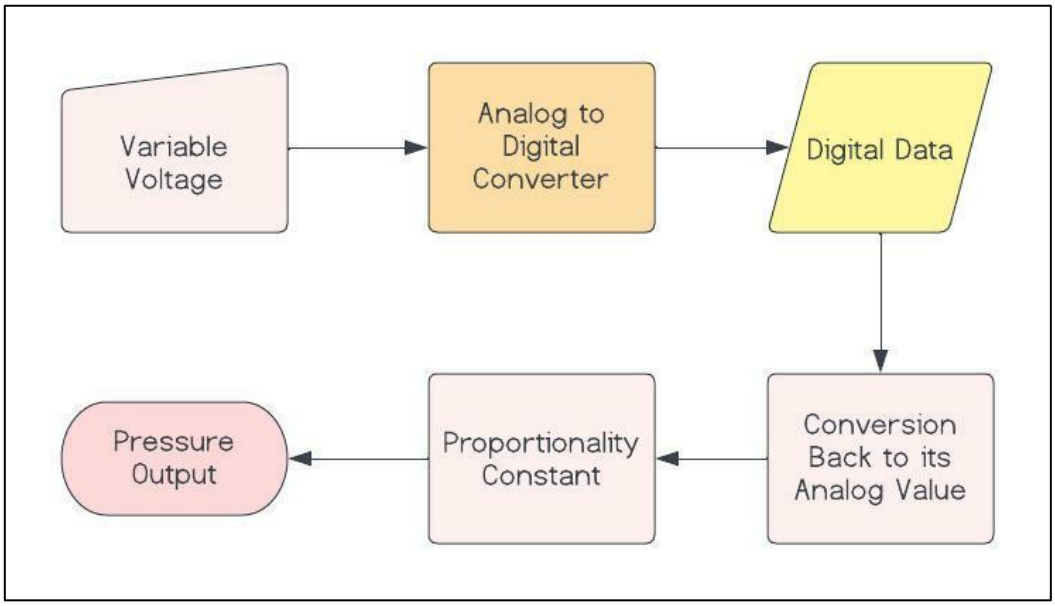


Fig. 2. Block Diagram of Pressure Input Block

6.2.2 Initial Velocity Input via SCI_B and Data Receive

In the Data Receive Block, a serial communications interface, also known as an SCI is used. It is a piece of hardware that, when connected to a microprocessor and various peripherals like printers, external drives, scanners, or mice, permits the transfer of data in a sequential manner, or one bit at a time.

In this regard, it is analogous to what is known as a serial peripheral interface (SPI). TMS320F28335 is the control card that is used, and it features SCI Transmit and SCI Receive registers that can be used to receive data as well as transmit it. In this application, we make use of a host system in order to send a value for the initial velocity. This would set the initial condition of the aircraft model in such a way that it would observe the changes brought about by the application of brake pressure. This is accomplished with the use of a UART Bridge Controller, which grants the user the ability to transmit data directly to the Receive pin of the microcontroller.

In case a new value is transmitted over the SCI as the Initial velocity, the New Input Detection Block will detect the change in the input and will reset the whole model by resetting the integrator. This will result in a new simulation being generated for the newly specified value that was received.

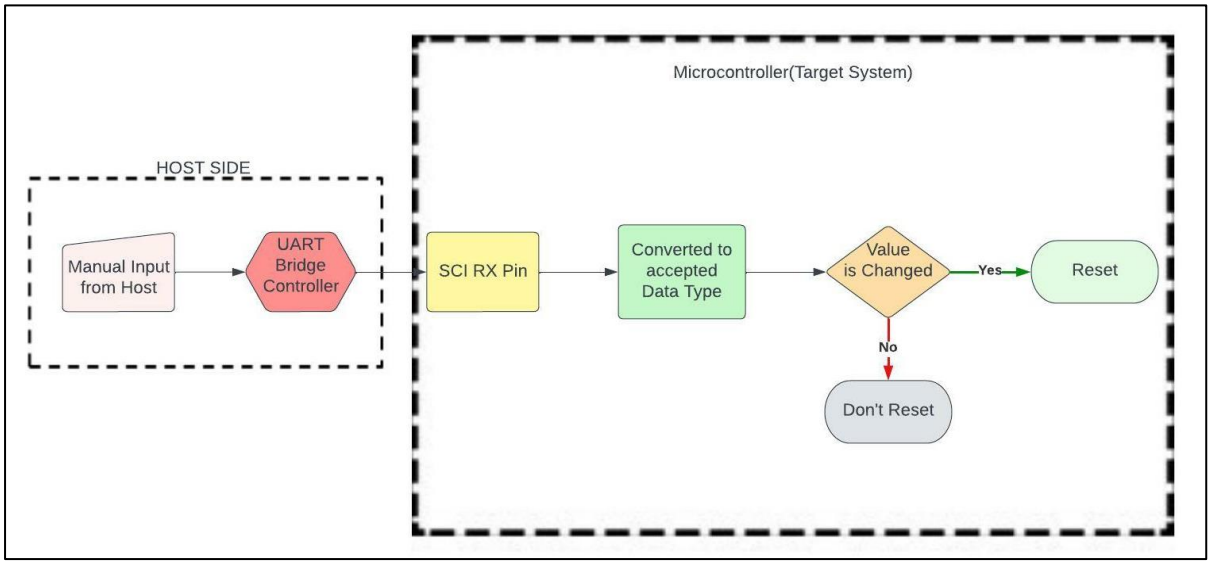


Fig. 3. Block Diagram of Initial Velocity Transmission

The block diagram above is a representation of how this serial transmission takes place from the Host Transmitter side sending the value of Initial Velocity in real-time setting to the Model deployed on the controller.

6.2.3 Normal Force Main

This block calculates the total normal force that is acting on the entire model by taking into account the mass of the aircraft as well as all of the factors that actively act on it such as gravity, acceleration, and a few other factors such as the distance between the wheels of the aircraft. This information is used to create the model.

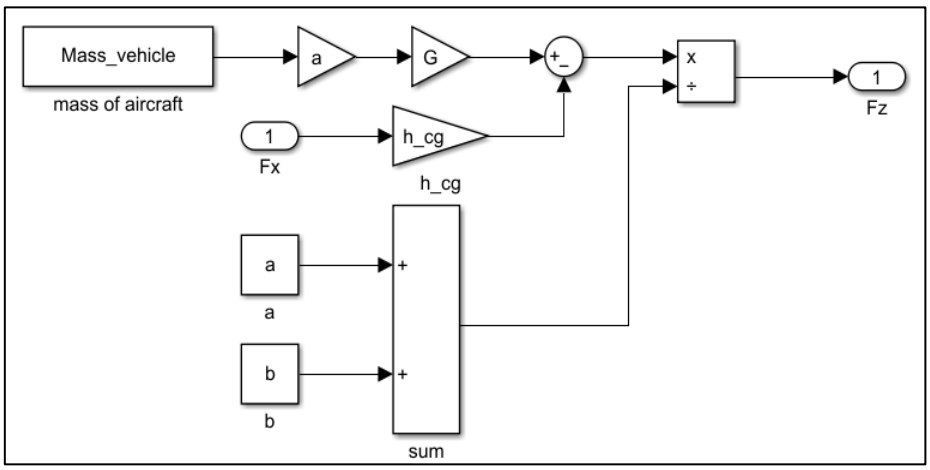


Fig. 3. Normal Force Main Block

6.2.4 Magic Tire Formula

The Magic Tire Formula is employed to empirically represent and interpolate tire force and moment curves that have been measured in the past. To do this, one must first measure the interaction that occurs between the tire and the road by estimating the location of the brake pad and its relating slip to tire friction on the road. This is often calculated with the formula:

Magic formula : Tire-Road Interaction
$Fx = f(\kappa, Fz) = Fz \cdot D \cdot \sin(C \cdot \arctan[\{ B\kappa - E \cdot [B\kappa - \arctan(B\kappa)] \}])$

Through the use of a formula, which is then entered into the calculation, it also takes into consideration the slip that occurs with the tires.

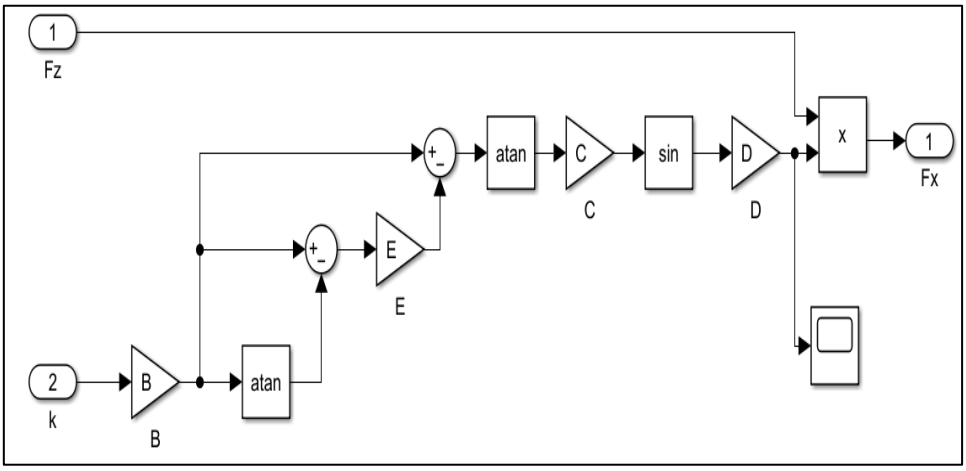


Fig. 4. Magic Tire Formula Block

6.2.5 Tire Kinematics Block

The Initial Velocity that the host transmits is received by the Tire Kinematics Block, which then integrates the value to turn it into the angular velocity while taking into consideration the moments that are operating on the tire. This Integrator block calculates the angular velocity, and each time the reset flag is set to 1, it causes this Integrator to be reset so that it may calculate the resultant velocity for the new initial velocity and multiply it with the radius of the tire to give the tire velocity which is later on fed to the Wheel speed Block which presents the output of the entire Model.

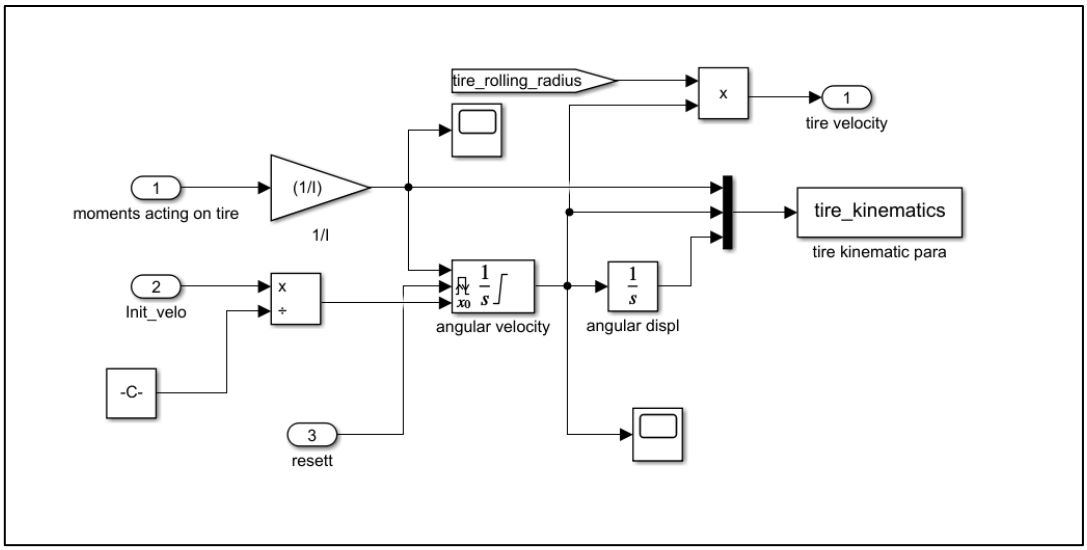


Fig. 5. Tire Kinematics Block

6.2.6 Wheel Speed Dynamics [via ePWM]

The Wheel Speed block mainly constitutes an ePWM. Controlling many of the power electronic systems that may be found in commercial and industrial machinery requires the utilization of a peripheral known as an enhanced pulse width modulator (ePWM). These systems include uninterruptible power supplies (UPS), switch mode power supply control, digital motor control, and other kinds of power conversion. The ePWM peripheral may also perform a digital to analog (DAC) function when the duty cycle is comparable to a DAC analog value; it is also frequently referred to as a power DAC. This DAC function takes place when the duty cycle is equivalent to a DAC analog value.

In implementing the electronic pulse width modulation (ePWM), the velocities of each wheel—the port side wheel, the starboard side wheel, and the nose wheel are converted to their respective frequencies, the factor of slip is also taken into account, and added to both the starboard and port side wheels.

Slip can be defined as an aerodynamic situation of uncoordinated flying in which an aircraft slides towards the inside of a turn or is moving somewhat sideways, in addition, to forward compared to the airflow that is coming from the opposite direction.

TBPRD = 0.5 [Ftbclk/Fpwm]
CMPA = 1/2 [TBPRD] (for 50% Duty Cycle)

Then, the equation as shown above is used in order to compute the Timer Based Period and from there, the CMPA value is derived to get the desired Duty cycle, which in this case is fifty percent. Real-time adjustments to the frequency of the wheels are made using the timer-based period register of the electronic pulse width modulator. These adjustments are made in response to changes in the brake pressure and the speed of the vehicle.

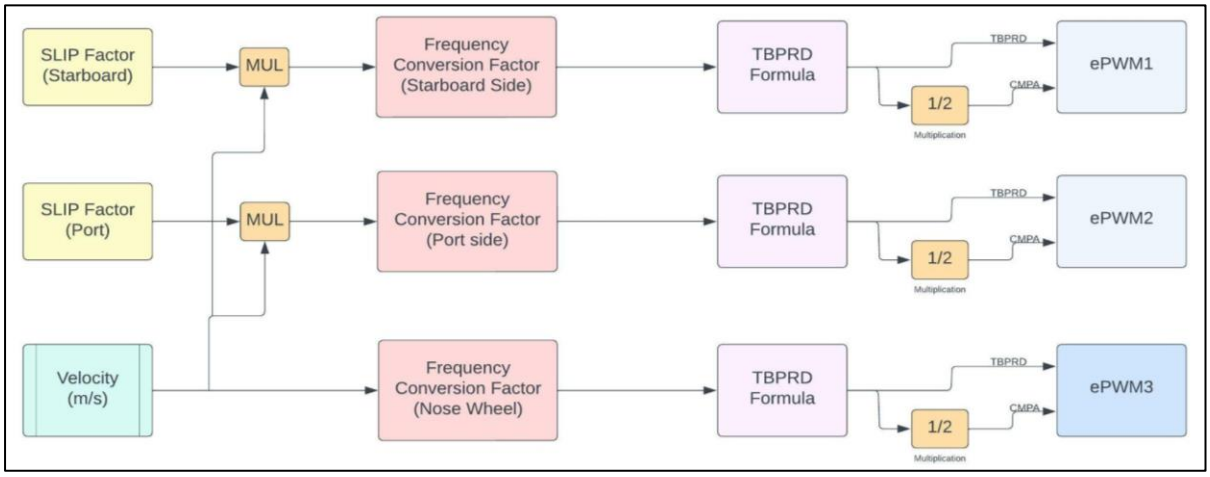


Fig. 6. Wheel Speed Dynamics using ePWM Module Block Diagram

6.2.7 Final Miniaturized Wheel Dynamics Simulator Model

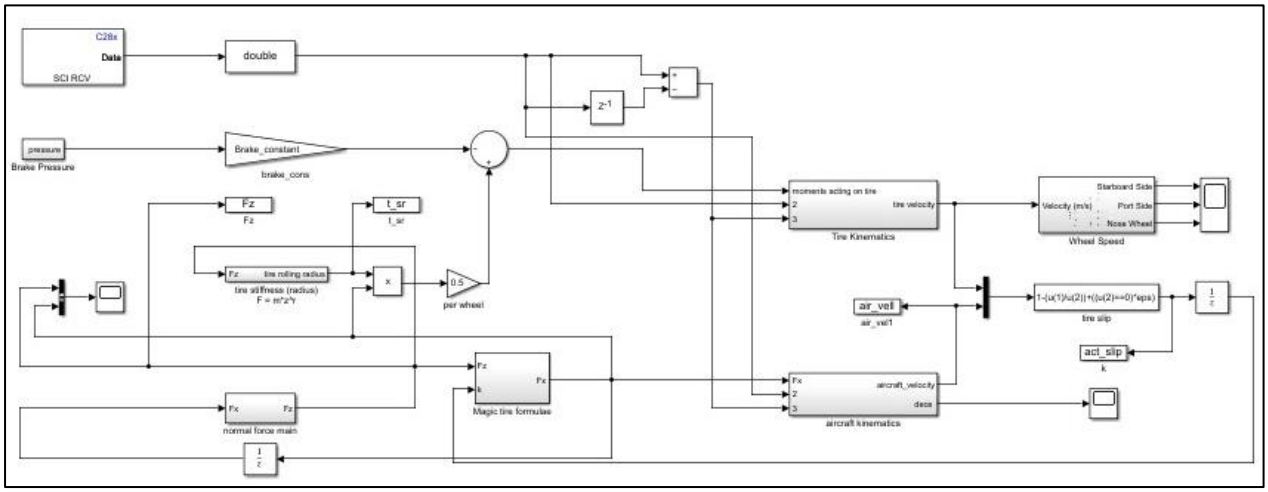


Fig. 7. Complete Simulink Model of Wheel Dynamics Simulator

The final model is constructed by integrating all of the submodules and components and then adding any delays or scopes that are required in order to see and collect data.

6.3 Graphic User Interface

The objective was straightforward that is to design a graphical user interface (GUI) capable of sending data to the serial communications interface (SCI) of the controller in order to provide an initial velocity as an input to the model at a baud rate that was defined in accordance with the rate of the SCI of the controller. Because of Python's extensive library, which makes it comparatively easy to create for our application, it was decided that Python would be the best choice for building the graphical user interface (GUI). The Tkinter library was used to design the graphical user interface (GUI) and interact with the microcontroller. The serial library is used to read and write data from and to the controller.

- **Python Code**

```
import tkinter as tk
from tkinter import *
from serial import *
import threading
from serial.tools import list_ports
class Graphics():
    pass
def connect_menu_init():
    global root, connect_btn, refresh_btn,text,lbl
    root=Tk()
    root.title("SCI")
    root.geometry("600x500")
    root.config(bg="lavender")
    port_label = Label(root, text="Available Port(s):", bg="lavender")
    port_label.grid(column=1 ,row=2, pady=20, padx=10)
    port_bd = Label(root, text="Baud Rate:", bg="lavender")
    port_bd.grid(column=1 ,row=3, pady=20, padx=10)
    init_vel = Label(root, text="Initial Velocity:", bg="lavender")
    init_vel.grid(column=1 ,row=4, pady=20, padx=10)
    refresh_btn=Button(root, text="Refresh", height=2, width=10, command=update_coms)
    refresh_btn.grid(column=3, row=2)
    connect_btn=Button(root, text="Connect", height=2, width=10, state="disable",
command=connections)
    connect_btn.grid(column=3, row=3)
    baud_select()
    update_coms()
    graph = Graphics
    graph.canvas= Canvas(root,width=300,height=300)
    graph.canvas.grid(row=5,columnspan=5)
    text=Text(root,width=25,height=1,background="lavender",font=('Sans Serif',8,'italic'))
    text.grid(column=2,row=4,padx=20,pady=10)
    sendButton=tk.Button(root,text="Send",command=write_to)
    sendButton.grid(column=3,row=4,padx=50)
```

```
    lbl=tk.Label(root,text="")
    lbl.grid(column=4,row=4)
def connect_check(args):
    if "-" in clicked_COM.get() or "-" in clicked_bd.get():
        connect_btn["state"] = "disable"
    else:
        connect_btn["state"] = "active"
def baud_select():
   global clicked_bd,drop_bd
   clicked_bd=StringVar()
   bds=["-
","300","600","1200","2400","4800","9600","14400","19200","28800","38400","56000","57600","11
5200","128000","256000"]
   clicked_bd.set(bds[0])
   drop_bd=OptionMenu(root,clicked_bd,*bds, command=connect_check)
   drop_bd.config(width=20)
   drop_bd.grid(column=2,row=3,padx=50)
def update_coms():
    global clicked_COM,drop_COM
    ports = list_ports.comports()
    #print(ports)
    coms=[com[0] for com in ports]
    #print(coms)
    coms.insert(0,"-")
    try:
        drop_COM.destroy()
    except:
        pass
    clicked_COM=StringVar()
    clicked_COM.set(coms[0])
    drop_COM=OptionMenu(root,clicked_COM,*coms, command=connect_check)
    drop_COM.config(width=20)
    drop_COM.grid(column=2,row=2,padx=50)
    connect_check(0)
def read_serial():
    global serialData
    while serialData:
        data=ser.readline()
        if len(data)>0:
            sensor=data.decode('utf8')
            print(sensor)
        #else:
            #print("NO DATA")
def write_to():
    while():
        send_data=text.get()
        if not send_data:
            print("Sent Nothing")
```

```
        ser.writelines(send_data)
    #UNCOMMENT TO TEST INPUT
    #inp=text.get(1.0,"end-1c")
    #lbl.config(text="Initial Velocity: "+inp)
def connections():
    global ser,serialData
    if connect_btn["text"] in "Disconnect":
        serialData=False
        connect_btn["text"] = "Connect"
        refresh_btn["state"] = "active"
        drop_bd["state"] = "active"
        drop_COM["state"] = "active"
    else:
        serialData=True
        connect_btn["text"] = "Disconnect"
        refresh_btn["state"] = "disable"
        drop_bd["state"] = "disable"
        drop_COM["state"] = "disable"
        port=clicked_COM.get()
        baud=clicked_bd.get()
        print(port,baud)
        ser= Serial(port,baud,timeout=0)
        t1=threading.Thread(target=read_serial)
        t1.daemon=True
        t1.start()
connect_menu_init()
root.mainloop()
```

- **Window**

The graphic user interface (GUI) includes a drop-down list that provides information about the connected COM ports as well as a collection of Baud Rates from which the user is given the opportunity to choose a baud rate that is appropriate for their controller. This is the speed at which the data that is being sent will be transferred. There's a Connect button that establishes a connection to the microcontroller. In order to enter and transmit a value to the model, a blank text box is provided.

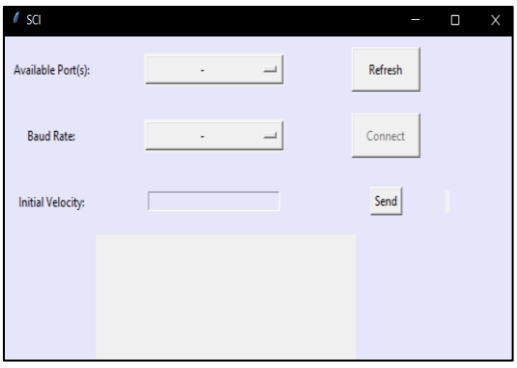


Fig. 8. GUI Window

7. TESTING & OBSERVATION

7.1 Overview

At various points throughout the development of the project, testing was performed on many levels. At first, accomplished this with the help of code composer studio. When the model is developed, Embedded coder creates a CCS Workspace that can be accessed to inspect the code generated. This workspace may be viewed to examine the code.

By inspecting the registers of the microcontroller, it can be determined whether or not the modifications that have been made are being reflected in its memory once the code has been built with CCS. The inputs of both the ADC and the SCI were checked using this method. After determining whether or not the associated ADC and SCIRX registers delivered the appropriate value, the change was then made to the model to reflect the updated information.

Additional testing of the model was carried out on an oscilloscope, and the goal of this testing was to establish whether or not the values derived from the ePWM waves are comparable to the values that were computed. This provides a graphical depiction of the frequency that is being output by the microcontroller as well as the consequent speed, the action of the brakes, and the aircraft's deceleration. This visual depiction was helpful in testing and verifying the outputs that were collected from the microcontroller. It was also helpful in fine-tuning the system to ensure that the results were accurate.

7.2 CCS Debugging

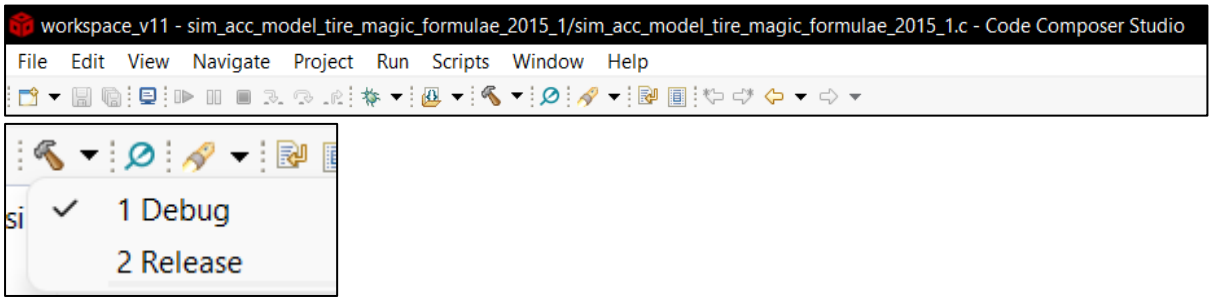


Fig. 1. CCS Debug Window

This is where CCS reads the values that have been stored into the registers of the microcontroller where debugging takes place on the software side. These numbers demonstrate whether or not the microcontroller is operating in an appropriate manner.

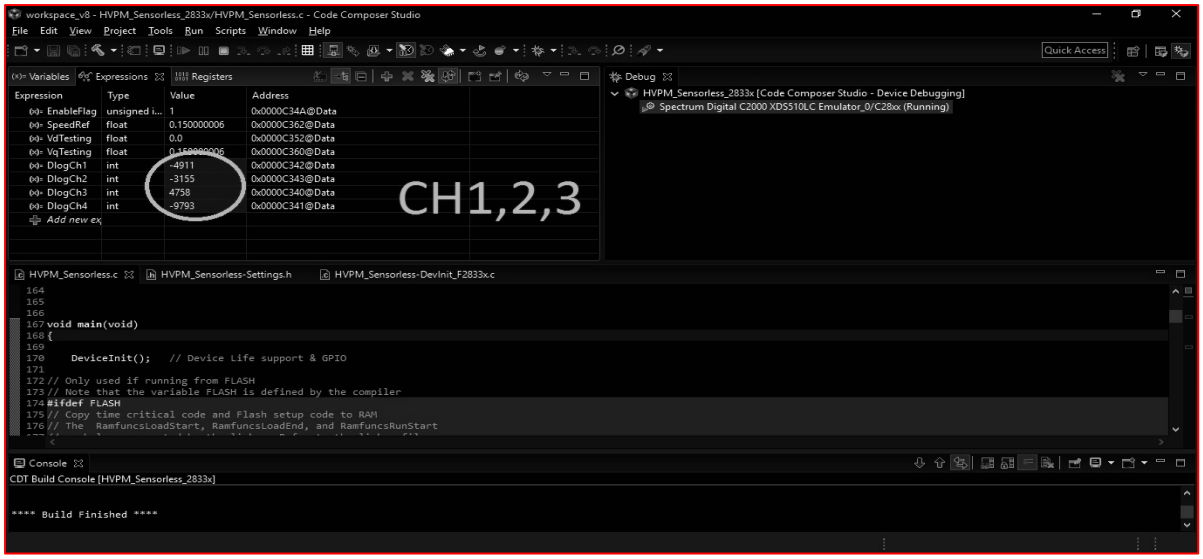


Fig. 2. CCS Expressions View in Debug Mode

7.3 Interfacing with the Landing Gear Controller [LGC]

The Landing gear Controller (LGC) is incorporated with this model of the Brake dynamics simulator. This is accomplished by first upscaling the signal in order to interface it with the LGC, which converts the 3.3V logic that our controller uses to 5V logic. This is done using a 5V logic interface PCB.

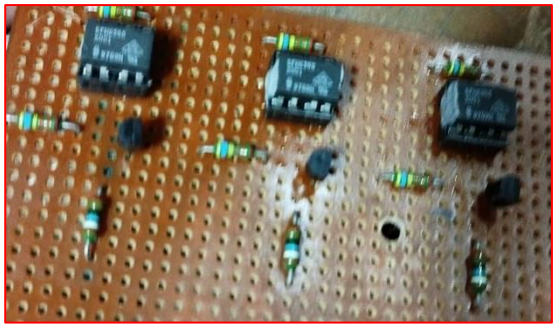


Fig. 3. 5V Interface PCB

After that, the up sampled 5V signal from the PCB is fed into the LGC, which is hooked to the signal that has been upscaled. This Signals can be viewed in the monitoring software where the data can be logged for results and graphs. The controller of the Landing Gear is coupled to two power supplies that provide energy to it. From thereon, the corresponding wheel-speed data is read from the LGC and displayed onto the Aircraft Telemetry Display.

Fig. 4. below represents the general flow of the entire hardware connection and setup for the model.

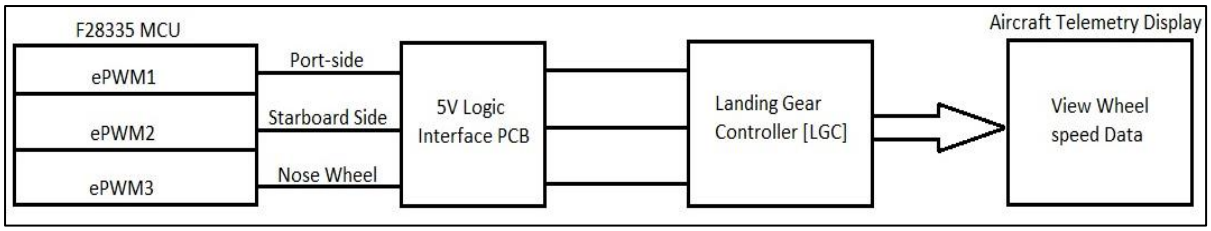


Fig. 4. Hardware Interface Flow

7.4 Testing using Oscilloscope

An oscilloscope is a sort of electronic test instrument that graphically shows varying electrical voltages as a two-dimensional plot of one or more signals as a function of time. Informally, the term "scope" refers to this type of electronic testing device. The primary functions are to display repeating or single waveforms on the screen that would otherwise occur too briefly for the human eye to see them. If these waveforms were not displayed, they would not be visible. The exhibited waveform can then be examined in order to determine several features, including amplitude, frequency, rising time, time interval, and distortion, amongst others.

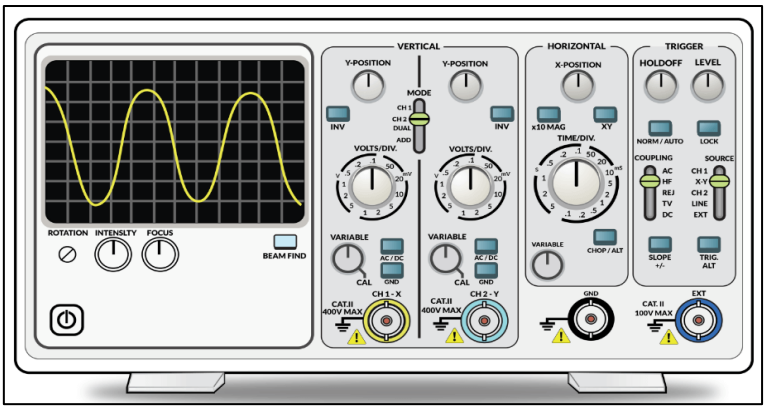


Fig. 5. Digital Oscilloscope

At first, there are no waves being output through the microcontroller, therefore the scope only displays flat lines. As shown in the figure below.

Fig. 6. Ideal Scope

Shown below is oscilloscope observation for a variety of initial velocities and their respective equivalent frequencies for the conversion factors of:

Nose Wheel Velocity	Conversion Factor – 2.82
Port Side & Starboard Side Wheels Velocity	Conversion Factor – 49.9

➢ **Initial Velocity: 57m/s**

As per the conversion factors, the frequencies are determined at which the three wheels of the aircraft rotate, assuming an initial velocity of 57 meters per second:

Port Side & Starboard side Wheels [Yellow and Blue]: 2.8kHz [approx.]

Nose Wheel [Green]: 160Hz [approx.]

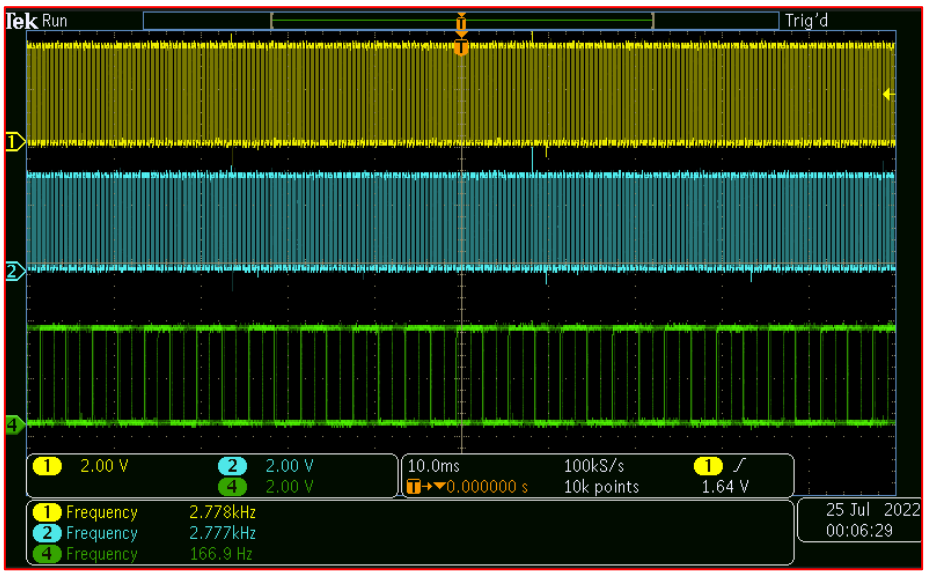


Fig. 7. Scope Readings [approx.] when Initial Velocity = 57m/s

➢ **Initial Velocity: 33m/s**

As per the conversion factors, it can be determined that the frequencies at which the three wheels of the aircraft rotate are, assuming an initial velocity of 33 meters per second:

Port Side & Starboard side Wheels [Yellow and Blue]: 1.6kHz [approx.]

Nose Wheel [Green]: 93Hz [approx.]

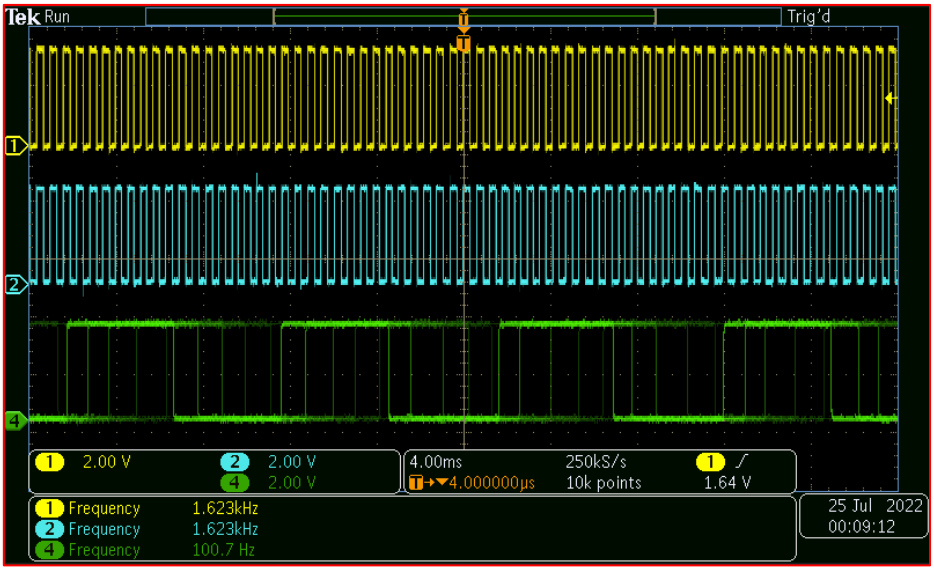


Fig. 8. Scope Readings [approx.] when Initial Velocity = 33m/s

➢ **Initial Velocity: 100m/s**

As per the conversion factors, it can be determined that the frequencies at which the three wheels of the aircraft rotate are, assuming an initial velocity of 100 meters per second:

Port Side & Starboard side Wheels [Yellow and Blue]: 5kHz [approx.]

Nose Wheel [Green]: 282Hz [approx.]

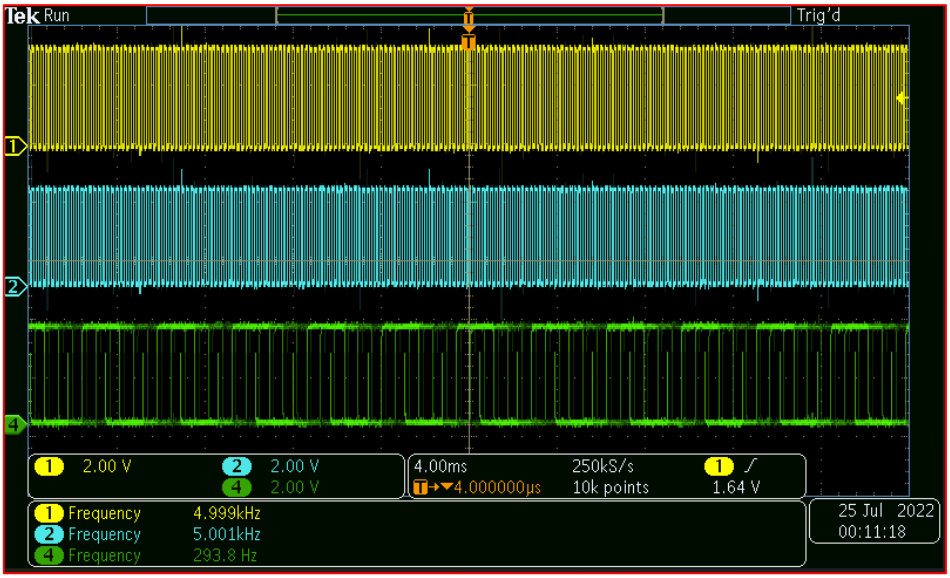


Fig. 9. Scope Readings [approx.] when Initial Velocity = 100 m/s

7.4 Observation-Aircraft Wheel speed Telemetry Data

To display the results and verify them, the data was logged to obtain values from the System Integrated with the Landing Gear Controller. This made possible observing the functioning of the brakes when they were applied. The following observations were made:

- Initial Velocity: 33m/s

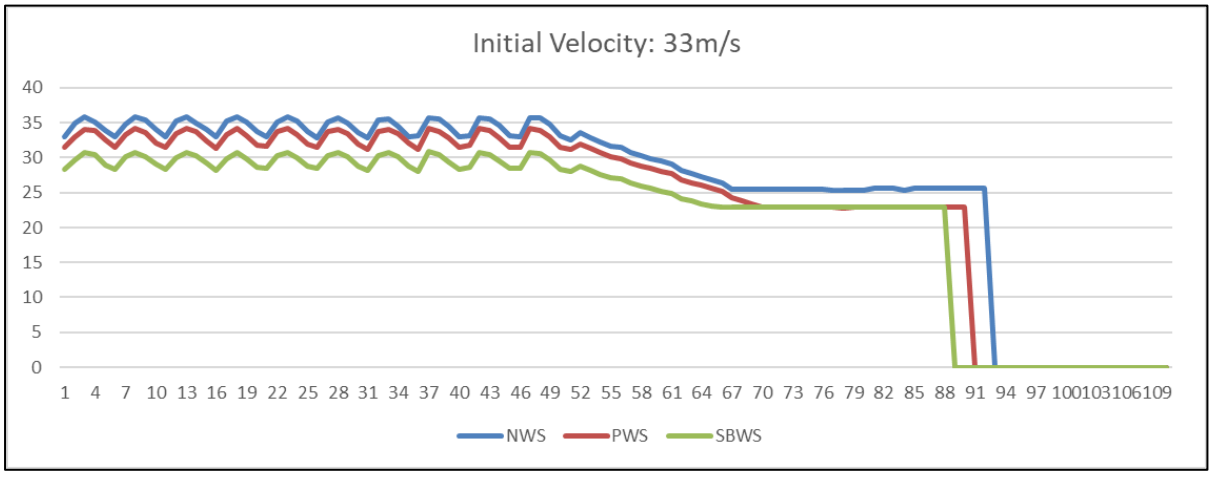


Fig. 10. Wheel-speed Graph for 33m/s

- Initial Velocity: 57m/s

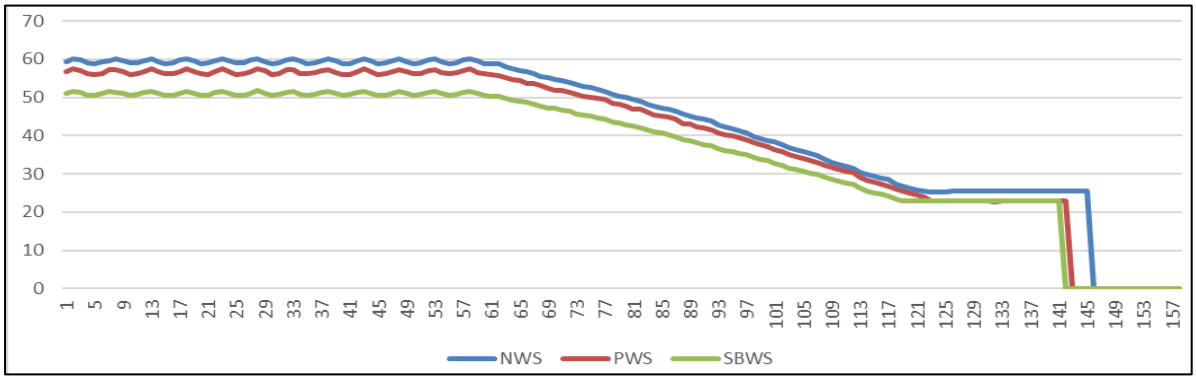


Fig. 11. Wheel-speed Graph for 57m/s

- Initial Velocity: 100m/s

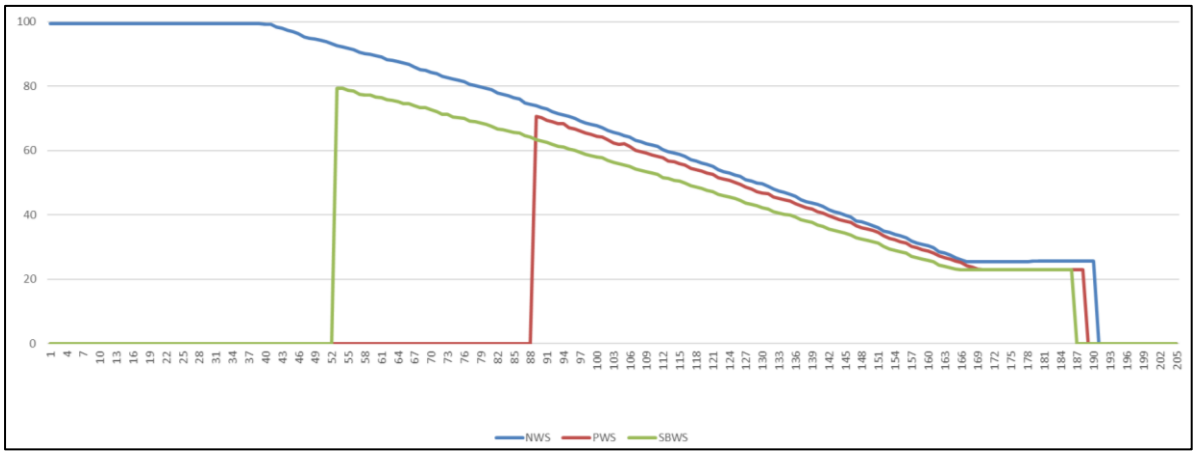


Fig. 12. Wheel-speed Graph for 33m/s

NWS- Nose Wheel Side PWS- Port Wheel Side SBWS- Starboard Wheel Side

The initial velocity is transmitted to the microcontroller's SCI interface, and it can be seen that it is mirrored in the charts that are shown above. The potentiometer, which simulates the brakes, has been set to 0 at this point, so there is no pressure acting on the wheels; as a result, the aircraft continues to go at the same speed during the entire flight. The wheels run at a constant speed, which is the same as the initial velocity. When the value of the potentiometer is turned all the way up to its maximum, it is visible to see the brakes working on the wheels of the aircraft. The velocity will fall off gradually until it is at a standstill (the aircraft stops). When the initial velocity is assumed to be 100 m/s, it is apparent to see that the initial velocity does not manifest itself in either the port side or the starboard side wheels. The reason for this is due to a limit filter that only shows values for these wheels that are lower than 77. Because of the acting slip that is delivered to the starboard side wheel, it's visible how the port side and starboard side wheels have different speeds. This is because the starboard side wheel has a slip of 0.1.

8. CONCLUSION & FUTURE SCOPE

A miniaturized model for measuring wheel dynamics of an aircraft using enhanced pulse width modulation discussed in Chapter 4, section 4.2 was presented and wheel pressure taken through ADC studied in Chapter, section 4.3 was varied using a variable resistor, that is, potentiometer and a successful check was made for the corresponding reduction in the velocity of aircraft wheels upon applied pressure using a Landing Gear Controller.

The work was successfully completed while keeping the general objectives mentioned in section 1.1 in mind. Embedded coder successfully accelerates the design process for the measuring the wheel dynamics application. The hardware used TI-F28335, a floating-point processor had the capabilities to perform complex mathematical operations without any issues. The velocity given externally as input was observed and recorded successfully on the aircraft telemetry display. All the observations that were obtained were in real-time through external measurements using equipment's such as oscilloscope**,** aircraft telemetry display.

The hardware implementation requires a little more upgrades, that is, if the timer-based period register were to be 32-bits, it could be programmed in such a way that it works for a variable frequency range from the lowest engineering values of 2.82hz for nose wheel and 49.9 hz for port side & starboard side to highest engineering values of 282hz for nose and 4999hz for port side and starboard side. Alternatively, instead of TBPRD being 32-bits wide, the clock based prescale divider [CLKDIV] and high-speed clock prescale divider [HSPCLKDIV] could be made to fit the engineering range of frequencies for the three wheels as stated above. The value of the slip for port-side and starboard side could also be dynamically varied and given as input like initial velocity and sent via the COM10 port and serial communication interface [SCI_B] of MCU. The values would then be sent by segregation into different packets each a byte long.

Moreover, using a python program that is capable of transmitting the data to the microcontroller using the UART Bridge controller is the method by which the values are sent to the SCI Receive of the microcontroller. This can be expanded further by packaging a fully functional application that has the ability to refresh and update itself whenever a new value is set in accordance with the requirements of the application. Minor tweaks in these aforementioned aspects could provide some more upgradations to the overall wheel dynamics model. All-in-all, the processor used here has shown the capability to handle and control multiple system(s) at the same time.

REFERENCES

[1] Priye Kenneth Ainah, Niger Delta University, DSP TMS320F28335 Implementation of dq-PI Vector Controller for Voltage Source Inverter using SPWM Technique, Nigerian Journal of Engineering, Vol. 27, No. 2, August 2020

[2] Mr. Bhanu Babaiahgari, Dr. Jae-Do Park, University of Colorado, Denver, Work in Progress: Design and Implementation of an Advanced Electric Drive Laboratory using a Commercial Microcontroller and a MATLAB Embedded Coder, 126th Annual Conference & Exposition, ASEE, Paper ID #27080

[3] Texas Instruments.TMS320x2833x, 2823x Technical Reference Manual.

[4] MathWorks. Files and folders created by build process - MATLAB & Simulink.

[5] MathWorks. Getting Started with Embedded Coder

[6] TMS320F2833x, TMS320F2823x Digital Signal Controllers (DSCs) Datasheet, sprs439p – June 2007 – Revised February 2021.

[7] Texas Instruments. The Essential Guide for Developing with C2000™ Real-Time Microcontrollers, spracn0d – September 2020 – Revised October 2021

[8] TMS320C28x CPU and Instruction Set. Reference Guide, spru430f August 2001–Revised April 2015.

[9] Texas Instruments. 2000 Real-Time Control MCU Peripherals, spru566o June 2003 – Revised October 2021

[10] Texas Instruments.TMS320x2833x Analog-to-Digital converter (ADC-spru812a.pdf.

[11] Texas Instruments. TMS320x2833x, 2823x enhanced pulse width modulator (ePWM) reference guide (rev. a) - sprug04a.pdf.

[12] Akrem Mohamed Elrajoubi, Simon S Ang, Ali A. Abushaiba, University of Arkansas, TMS320F28335 DSP programming using MATLAB Simulink embedded coder: Techniques and advancements, IEEE 18th Workshop on Control and Modeling for Power Electronics (COMPEL), July 2017

[13] MathWorks. Set Up Serial Communication with C2000 Hardware Board

[14] MathWorks. ADC-PWM Synchronization Using ADC Interrupt

[15] Suying Zhou, Hui Lin, and Bingqiang Li, Research on HILS Technology Applied on Aircraft Electric Braking System, Journal of Electrical and Computer Engineering
Volume 2017, Article ID 3503870

APPENDIX

Hardware Setup

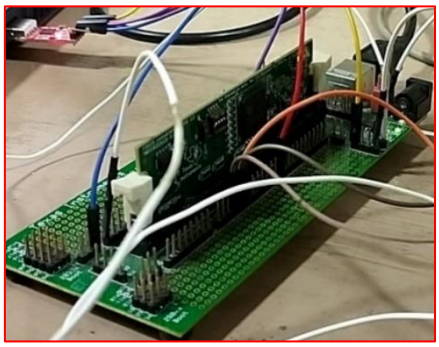


Fig. 1. Control card and Docking Station

Fig. 2. USB2TTL module for Serial Communication

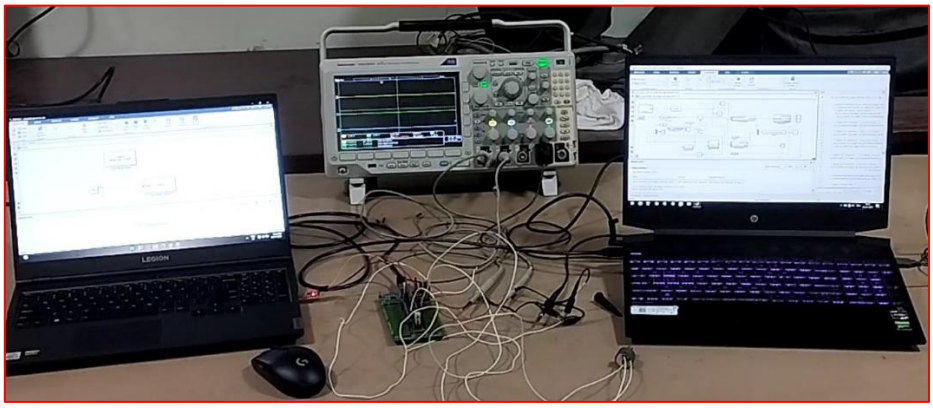


Fig. 3. Initial Simulation setup using DSO

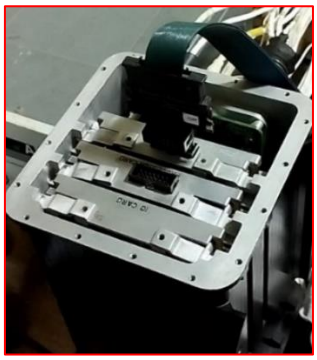


Fig. 4. Landing Gear Controller

Fig. 5. 5V Up-scale Logic Interface PCB

Fig. 6. LGC Interfacing with 5V logic Interface board

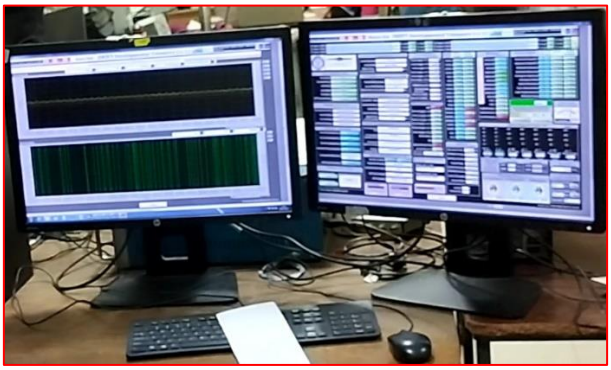


Fig. 7. Aircraft wheel speed data Telemetry Display

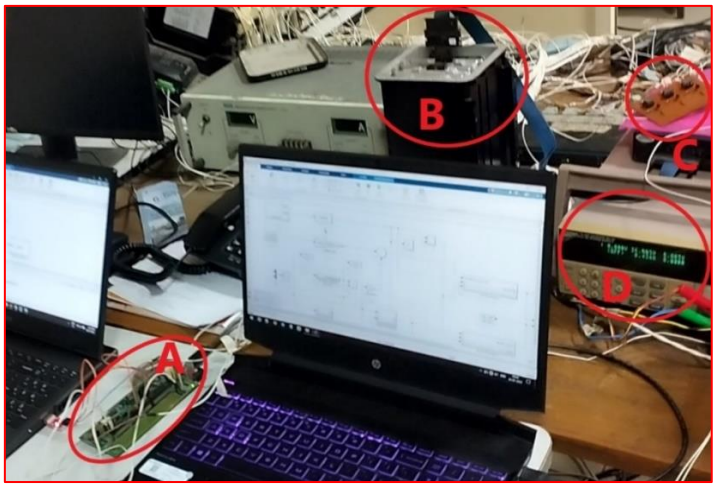


Fig. 7. Final Test using Complete Setup

A	TMS320F28335 MCU
B	LGC [Landing Gear Controller]
C	5V Up-scale logic PCB Interface
D	Power Supply