Bachelorarbeit

Development of a Serial Peripheral Interface for a Large Area High Resolution Hall Scanner

Ausgeführt am

Atominstitut der TU Wien

unter Anleitung von

Priv. Doz. Dipl.-Ing. Dr. techn. Michael Eisterer

durch

Mario Raphael Hiti

Matr. Nr.: 1327428

Auhofstraße 66B/11
1130 Wien

Wien, 11. Dezember 2018

Contents

Abstract

This project covers the implementation of a triple channel Serial Peripheral Interface (SPI) between a F28377S micro controller from Texas Instruments and various analog-to-digital converters. Furthermore, the controller's internal digital-to-analog converters are expanded to serve as programmable function generators. For both applications dedicated hardware modules, such as Direct Memory Access (DMA) or Enhanced Pulse Width Modulation (EPWM), are utilized to minimize the required processing power. The analysis of data in real-time and the communication with a host-PC is covered in a separate document.

Zusammenfassung

Diese Arbeit dokumentiert die Umsetzung eines dreikanaligen Serial Peripheral Interface (SPI) zwischen einem F28377S Mikrocontroller von Texas Instruments und verschiedener Analog-Digital-Umsetzer. Des Weiteren werden die internen Digital-Analog-Umsetzer des Controllers erweitert um als programmierbare Funktionsgeneratoren zu fungieren. Für beide Aufgaben wird auf verschiedene Hardware Module des Controllers, beispielsweise Direct Memory Access (DMA) und Enhanced Pulse Width Modulation (EPWM), zurückgegriffen um die erforderliche Rechenleistung der CPU zu minimieren. Die Auswertung der Daten in Echtzeit und die Kommunikation mit einem Host-PC wird in einem separaten Dokument behandelt.

1. Introduction

In modern research facilities personal computers have become an integral part of capturing and analyzing data. While they do offer a lot of processing powers and a wide variety of software applications, their complexity often makes it difficult to implement low-level solutions that work closely with custom-made hardware. Furthermore they usually lack interfaces such as SPI and I²C that are otherwise commonly used in embedded systems. For those reasons it is often easier to use dedicated micro controllers instead.

Although less sophisticated than modern PCs, micro controllers provide a wide variety of interfaces and specialized hardware components such as pulse width modulation (PWM) or analog-to-digital converters (ADC). Their strength lies in their simplicity which is why many are able to function without an operating system which makes implementing or time-critical operations low-level tasks easier.

This project started with the goal to provide an SPI interface between several ADCs from a detector board and the LaunchXL-F28377S controller from Texas Instruments. During the development process the project scope quickly began to rise since it became clear that many other features were necessary to ensure the required performance.

1.1. Project scope

The main goal of this project is to provide an interface between a host PC and the detector board used for the experiment. As mentioned above additional features were required for development and performance optimizations. In summary the scope of this project can be compiled into the following list:

1. 3 parallel 50Mbps SPI channels, each connected to two different ADCs.
2. Driving ADCs and amplifier buffers using GPIOs and PWM.
3. Adjustable sample rate of up to 2MS/s and adjustable signal averaging on a hardware level.
4. Storing and processing data using programmable digital filters (FIR filtering). Filters can be exchanged during operation.

5. Usage of direct memory access (DMA) for faster data transmission within the controller.
6. 3 programmable function generator channels for generating periodic signals on each DAC. Signals can be changed during operation.
7. I2C interface for controlling additional output pins (GPIO extender)
8. UART interface for communication with a host PC[1]
9. USB interface for for sending data from a host PC to the controller and vice versa.
10. Custom USB protocol for downloading data, synchronizing controller settings and sending commands.
11. Graphical user interface for controlling the device and displaying data including interactive charts and tools for data analysis.
12. Inter-process communication using named pipes to provide an interface for other applications on Microsoft Windows.

The amount of work on those features and the necessary documentation is exceeding what would usually be considered to be the scope of a typical bachelor's thesis. This is why the decision was made to split the project into two parts. The first part which is represented by this document will cover the controller, its hardware and the techniques used to acquire and process data. The second part will cover the USB interface, the graphical user interface and inter-process communication.

1.2. Structure and scope of this Document

The next chapter will outline the experiment followed by a brief overview of the most important hardware components.

Chapter 3 will cover the architecture and used features of the F28377S controller excluding its SPI Interface which will be covered in more detail in chapter 4 since it is the main focus of this document.

The actual software which brings all those components together will outlined in chapter 5. This section also serves as a reference for the most important functions and routines. The source code for the contoller is available on:

www.github.com/hitimr/Scanner_Controller

[1]This feature is no longer used and has been replaced by a USB interface.

2. The Nano Hall Scanner

2.1. Overview

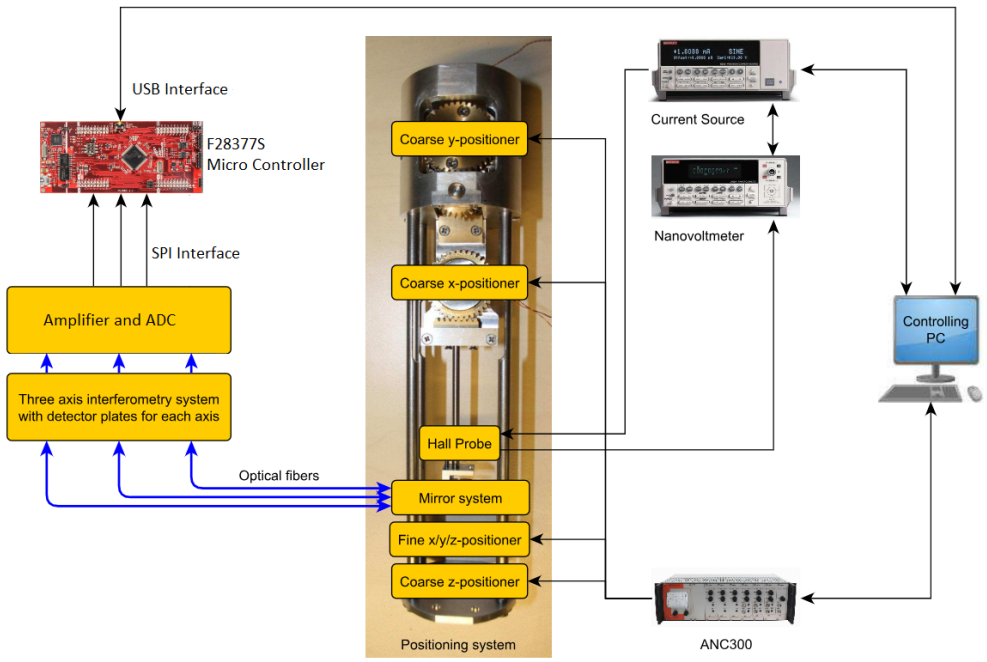

Figure 2.1.: Schematic overview of the Nano Hall Scanner. Original image from David Bader, *Process Control for a Large Area High Resolution Hall Scanner*, (2017), 15

Figure 2.1 shows the basic setup of the Nano Hall Scanner system. The positioning system consists of 6 piezoelectric actuators (2 for each axis) which are regulated by an Attocube ANC300 motion controller. At cryogenic temperatures (4K) the hall probe can be positioned in a 30x30x15 µm cube. Additionally the z-axis features a coarse mode that can quickly cover a distance of up to 5mm [1].

The relative positional change of each axis is measured with an interferometer (see section 2.2.1). The resulting change in brightness is measured with an optical sensor, amplified and converted to a digital signal using an LTC2380-24 ADC (see section 2.2.3).

The F28377S controller is responsible for controlling and reading out all three ADCs using an SPI bus system. The resulting data is processed using two digital filters and stored in a circular buffer. The controlling PC is connected to the board via an USB interface that is used to transmit the measurement data and experiment settings.

A graphical user interface is available on the PC for displaying measurement results and configuring settings for measurements and the controller (i.e. adjusting sample rates, uploading new filters, etc.).

2.2. Devices

2.2.1. Interferometer

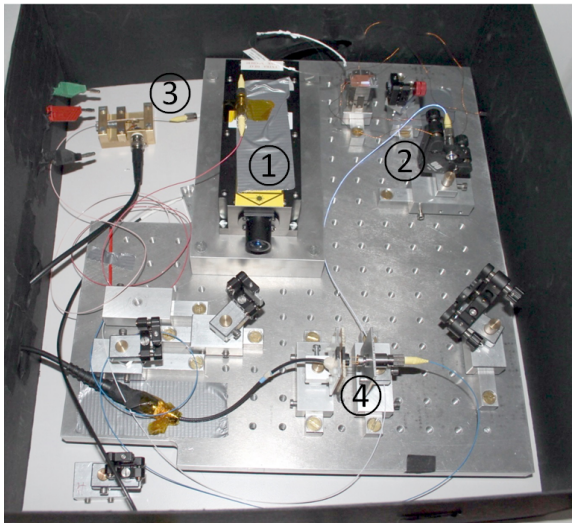

Figure 2.2.: Prototype if the interferometry system consisting of laser (1), beam splitter (2), fiber strecher (3) and detector (4).

Figure 2.2 shows an early design of the prototype used during development. It resembles the basic setup of a Michelson interferometer with a fixed and a movable mirror. The key difference is the additional fiber strecher. This part continuously stretches and releases the optical fiber at a rate of 10kHz causing an additional modulation of the signal.

2.2.2. Detector Board

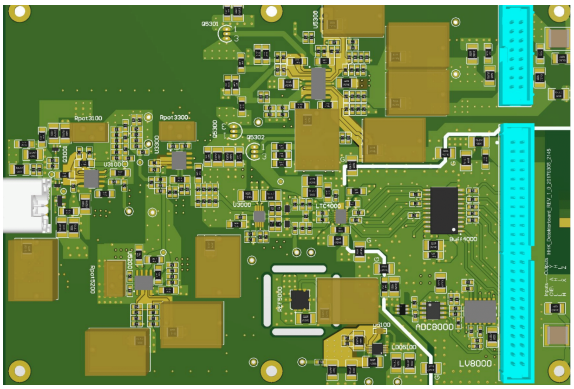

Figure 2.3.: The detector board. The separated part on the bottom right is the digital side.

The detector board is a custom built piece of hardware used to amplify the sensor's output signal. The gain is set to $\sim 1.1 \cdot 10^6$ but can be modified by exchanging certain resistors. The board is divided into a analog and a digital side to block out high frequencies created by digital signals. The LTC2380-24 is the "bridge" between the two sides. Furthermore a second ADC is located on the board that is used to translate the signal from a temperature sensor. This sensor is soldered onto the board and used to detect overheating caused by malfunctioning hardware. In total 3 detector boards (A,B and C) are connected to the controller.

2.2.3. LTC2380-24 ADC

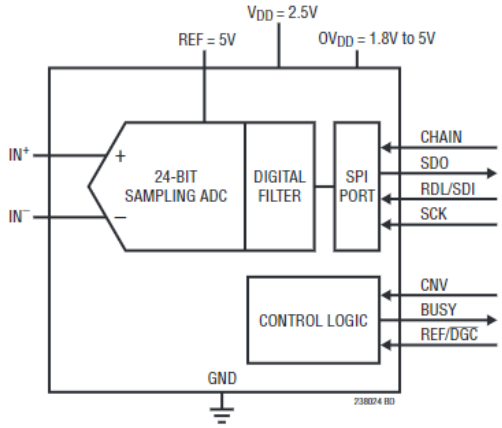

Figure 2.4.: LTC2380-24 functional block diagram from the data sheet. [2, p.10]

The LTC2380-24 is the central interface between the analog side (detector board) and the digital side (controller). Its main purpose is to translate the amplified analog

signal into a 24-bit binary format for further processing by the controller. The input signal can range between ±5V resulting in a maximum resolution of 1.2 µV/bit.

A new sample (conversion) is taken on every rising edge on the CNV input. Once the conversion request is received BUSY is set to high until the process is completed. During this time no other signals should interfere with the ADC. As an additional protective measure an external buffer is placed on the detector board to isolate the ADC from any disruptions and noise.

In order to achieve a better signal-to-noise ratio multiple samples can be taken and averaged. This is done by sending additional conversion pulses to the device. The LTC2380-24 automatically adds the result of each conversion together and divides them by the next power of 2. Since this does not equate to the actual mean average an additional weighting factor has to be calculated. This process will be illustrated using the following two examples:

1. The ADC has performed $N = 7$ conversions. The next highest power of 2 is $M = 8$. The results $R_0, R_1, \ldots R_7$ are added together and divided by M. Thew resulting value is smaller than the arithmetic mean average. Therefore an additional weighting factor is required

$$D = \underbrace{\frac{M}{N}}_{\text{weighting factor}} \cdot \left(\sum_{i=0}^{N} \frac{R_i}{M} \right) \tag{2.1}$$

   The part in parentheses of Equation 2.1 is performed automatically by the ADC. The additional weighting factor has to be calculated by the controller and multiplied on top of the transmitted result.

2. The ADC has performed $N = 16$ conversions which is already a power of 2 resulting in $N = M$. The results $R_0, R_1, \ldots R_{15}$ are added together and divided by M. This already leads to the correct arithmetic mean average resulting in a neutral weighting factor of 1.

The rationale why the LTC2380-24 performs the process of averaging results in this unusual way is not documented. However the underlying reason might be that the ADC is only capable of binary addition. Instead of dividing the sum by N, the result is merely shifted to the right by one bit which corresponds to a division by 2. This bit-shift operation is performed every 2^Nconversions resulting in the pattern described above.

The resulting data can be transferred using the SPI-port[1] by keeping CNV low and sending a clocked signal to SCK (see Figure 2.5). One transmission consists of 40 bits at a maximum bit rate of 50Mbps. The first 24 bits represent the averaged result without the weighting factor described above. The following 16 bits represent

[1]The interface is not a “true” SPI in a sense that the MOSI wire is missing

the number of averaged results $1 \leq N \leq 65536$. Each bit is transmitted on the rising edge of SCK with the most significant bit first.

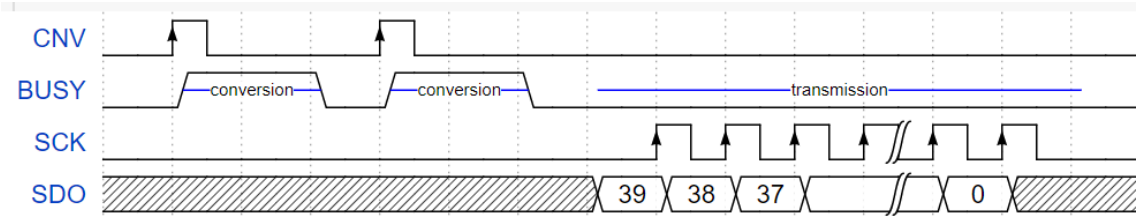

Figure 2.5.: Timing diagram for two conversions followed by a transmission

The transmission itself can be stopped prematurely by sending less than 40 SCK pulses to the port. As soon as the next conversion is started the sampling buffer is flushed and N is set to 1. This can be used to save bandwidth if the number of averaged results is already known. Testing has shown that sending more than 40 SCK pules to the LTC2380-24 has no effect.

If multiple devices are connected to the SPI the RDL/SDI pin can be used to select which slave is active. If the pin is set to high the serial port is disabled allowing the master to communicate with other devices. To re-enable communication with the LTC2380-24 RDL/SDI has to be set to low.

2.2.4. LTC2400 ADC

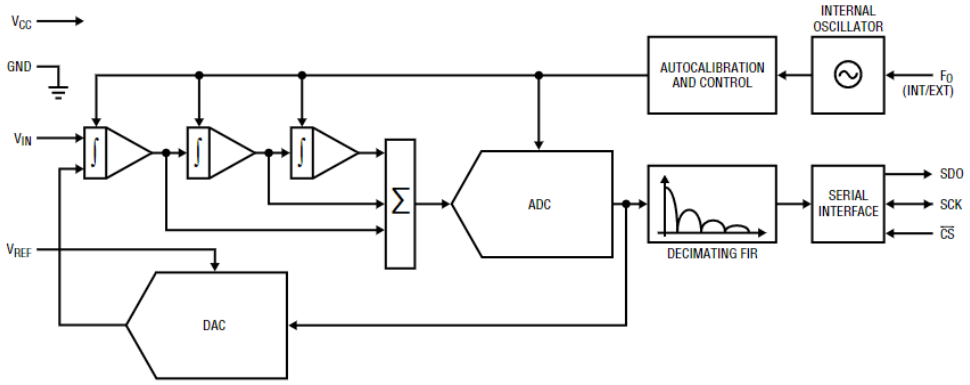

Figure 2.6.: LTC2400 functional block diagram from the data sheet [3, p.9]

The LTC2400 is a 24-bit ADC which is used to translate the analog output voltage of the temperature sensor into a binary format for the controller. This ADC has primarily been chosen because it offers a similar SPI interface as the LTC2380-24 which ensures compatibility and an easy transition between the two devices during operation. The actual precision of the sensor and the ADC is of less importance

since their main purpose is to indicate if the board has reached a critical temperature to prevent overheating.

The measured data is acquired continuously using an internal oscillator and instead of averaging results the LTC2400 simply overrides old data. If $\overline{CS}$ is set to low SDO and SCK can be used to transfer the result.

One transmission is 32 bits long. The first 4 bits represent flags indicating sign, input range and conversion rate. The next 28 bits represent the actual conversion which is transmitted MSb first. The last 4 bits are considered sub-LSb, meaning that their value is below the specified resolutionl, and can therefore be discarded.

One conversion takes up to 136ms and the maximum SCK frequency is 2MHz. This is significantly slower than the LTC2380-24 (400ns and 50MHz). This has to be taken into account if both devices are connected to the same SPI, meaning that the controller has to alter its configuration before changing the slave in order to communicate at optimal performance.

2.2.5. Break-out board

The break-out board is another piece of custom built hardware that has two main functions:

- Provide a mount for the F28377S controller
- Route signals from the controller's pins to all devices and connectors

Additionally each detector channel features a small logic unit that is necessary to ensure compatibility with all buffers and prevent conflicting signals.

3. The F28377S Controller

3.1. Overview

The TMS320F2837xS Delfino micro-controller is available in several different variations. This document primarily covers to the "C2000 Delfino LaunchPad LAUNCHXL-F28377S", which will also be referred to as "F28377S Controller" or "LaunchPad". Unlike other versions, this product is shipped on a dedicated development board which provides a wide range of extra peripheral features as seen in the board overview below (Figure 3.1).

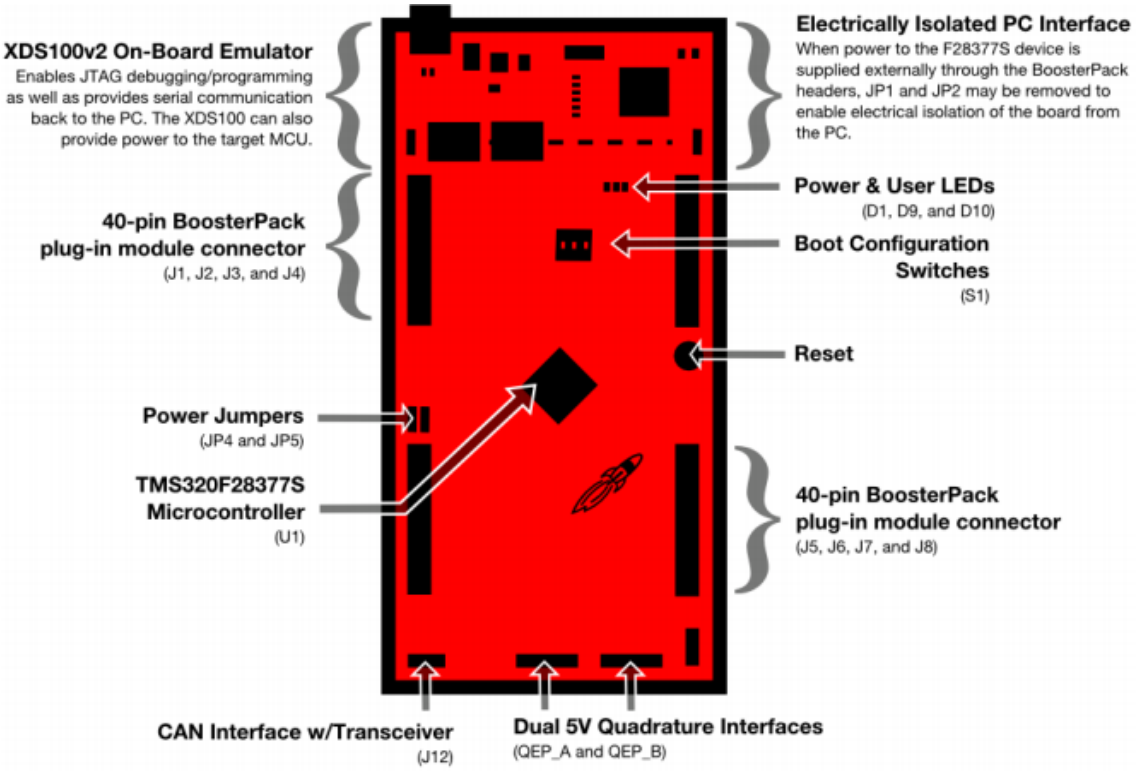

Figure 3.1.: Board Overview[4, p. 6]

The whole system is stand-alone and only requires a 5V power supply to operate which can be provided using any 5V pin or a USB cable. In order to use the controller for this particular project some hardware adjustments have to be performed before use which are described in Section 3.3 on page 21.

3.2. Summary of used features

The following section will outline all used hardware features including a short description of their functionality and how they are connected with each other. A detailed documentation on all systems and their internal structure can be found in the Technical Reference Manual[5] and the data sheet[6]. The implementation is documented in chapter 5.2. The SPI interface is covered in more detail in chapter 4 since it is the main focus of this project.

3.2.1. GPIO - General Purpose Input/Output

The GPIO module handles all digital signals that are connected to an on-chip pin. The chip, which soldered on the LaunchPad, comes with the PZP-package which has 100-Pins labeled 1-100. Most GPIOs are wired directly to one of the pins labeled J1-J80. Additionally there are some GPIOs with additional features such as Pin 4 which is also connected to an LED or Pin 7 & 8 which are connected to D- & D+ of the mini USB connector. For more information refer to the schematics [4, pp 10-15].

Internally each pin is located on one of 4 I/O ports (Port A - Port D) supporting up to 32 GPIOs, each with its own logic (see Appendix A.2). Note that some pins also offer a special analog setting for high speed operation which is relevant for SPI and USB interfaces. However the underlying wiring, logic is not documented.

Up to 12 different signals may be multiplexed on a single GPIO, though only one can be active at any given moment. A complete list of all assignable functions can be found in the data sheet [6, table 4-1]. Table 3.1 shows all used GPIOs and their corresponding signal.

Table 3.1.: The following table shows all assigned GPIOs with their respective type[1] and hardware assignments.

Name	GPIO	J-Pin	Type	Description
EPWM				
EPWM1A	NC	NC	CLK	Master EPWM
EPWM2A	2	80	CLK	Trigger for DMACH4
EPWM3A	4	79	CLK	Trigger for DMACH5
EPWM4A	6	NC	CLK	Trigger for DMACH6
EPWM7A	12	40	CLK	Conversion pulse for ADC-C
EPWM7B	13	39	CLK	Controls buffer for ADC-C
EPWM8A	14	38	CLK	Conversion pulse for ADC-B

[1]Nomenclature: CLK (clock or pulsed signal), I (digital input), O (digital output), SDI (serial data input), SDO (serial data output), AO (analog output), NC (not connected)

Name	GPIO	J-Pin	Type	Description
EPWM8B	15	37	CLK	Controls buffer for ADC-B
EPWM9A	16	36	CLK	Conversion pulse for ADC-A
EPWM9B	17	35	CLK	Controls buffer for ADC-A
EPWM10A	18	76	CLK	Trigger for SPI read out
GPIOs				
BUSY_A	41	5	I	ADC-A busy indicator
BUSY_B	99	53	I	ADC-B busy indicator
BUSY_C	20	34	I	ADC-C busy indicator
BUFF_OE_A	90	3	O	Manual control for ADC-A buffer
BUFF_OE_B	86	44	O	Manual control for ADC-B buffer
BUFF_OE_C	4	19	O	Manual control for ADC-C buffer
CS_A	89	4	O	ADC-A chip select
CS_B	87	43	O	ADC-B chip select
CS_C	21	33	O	ADC-C chip select
ADC_IO_A	61	8	O	Temperature ADC-A chip select
ADC_IO_B	66	50	O	Temperature ADC-B chip select
ADC_IO_C	72	13	O	Temperature ADC-C chip select
I2C				
SCL_A	92	59	CLK	I2C-A clock
SDA_A	91	52	I/O	I2C-A data
I2C_INT	78	11	I	I2C-A interrupt
SPI				
SPICLK_A	60	7	CLK	SPI-A Clock
SPICLK_B	65	47	CLK	SPI-B Clock
SPICLK_C	71	2	CLK	SPI-C Clock
SPISIMO_A	58	15	SDO	SPI-A slave-in master-out
SPISIMO_B	63	55	SDO	SPI-B slave-in master-out
SPISIMO_C	69	49	SDO	SPI-C slave-in master-out
SPISOMI_A	59	14	SDI	SPI-A slave-out master-in
SPISOMI_B	64	54	SDI	SPI-B slave-out master-in
SPISOMI_C	70	48 [2]	SDI	SPI-C slave-out master-in
$\overline{\text{SPISTE\_A}}$	61	61	I	SPI-A slave transmission enable
$\overline{\text{SPISTE\_A}}$	66	66	I	SPI-B slave transmission enable
$\overline{\text{SPISTE\_A}}$	72	72	I	SPI-C slave transmission enable
DAC				
DACA		27	AO	Digital-Analog Converter A
DACB		29	AO	Digital-Analog Converter B
DACC		24	AO	Digital-Analog Converter C
Other				
XCLKOUT	73	12	CLK	CPU Clock indicator ($f = CPUCLK/4$)

[2]Connected by hardware bridge

Name	GPIO	J-Pin	Type	Description
CPU_BUSY	62	18	O	CPU load indicator
DEBUG_PIN	10	78	O	Pin that can be toggled for debugging

3.2.1.1. GPIO Assignment

In order to illustrate the process of manually assigning a GPIO, EPWM11A will be connected to a pin:

1. Go through Table 4-1 Signal Descriptions in the data sheet[6, pp 16-38] and note any occurrences of EPWM11A. Some functions are available for multiple pins for additional flexibility. In the case of the PZP package EPWM11A is only available for GPIO20 (pin 12) with the mux position 5.
2. Look for GPIO20 in the tables GPyMUXx GPyGMUXx[3] linked in the Table 6-13. `GPIO_CTRL_REGS` registers[5, pp 751-753]. In this case GPIO20 is located on GPAMUX2[9:8] and GPAGMUX2[9:8].
3. Convert the mux position to binary ($[5]_{10} = [1000]_2$) and write the two most significant bits ($[10]_2$) to GPAMUX2[9:8] and the two least significant bits($[10]_2$) to GPAGMUX2[9:8].

The process described above is very laborious and prone to making errors caused by conflicting assignments and typing errors. Therefore it is highly recommended to use the PinMux online tool[4] which provides an intuitive UI and generates the necessary code automatically.

In order to modify the assignments locate the file *scanner_controller.pinmux* in the project folder and upload it. Apply all necessary changes via the GUI and download the files *f2837xs_pinmux.c* and *f2837xs_pinmux.h*. Place the downloaded files in the folder *src/peripherals* and overwrite the old ones.

3.2.2. On-Chip Memory

The controller offers three types of on-chip memory:

Registers: offer the fastest way of storing and accessing data. Most registers are predefined and dictate the controller's behaviour and its peripheral modules. Only a few are available for computational use. The assignment of those is generally performed by the compiler.

[3] y refers to the mux register of the ports A-D. Each port is furthermore split into GPAyMUX1 and GPAyMUX2 due to the limitation of 32bits per register

[4] dev.ti.com/pinmux (Requires a TI-Account)

RAM: is the primary storage for the CPU. It offers good access times and storage capacity. However all data is erased after reset or loss of power. In general RAM is used to store variables, measurement data and time critical functions and routines.

Flash: is the slowest type of memory but offers the most storage. Furthermore Flash is non-volatile, meaning that all data is secured even after a power outage. Therefore the complete program code is stored on Flash. The process of uploading code to Flash (flashing) is performed by the IDE automatically. Once the controller is flashed data can be read by accessing the corresponding memory address. Writing to Flash at run-time is not supported by default and requires a dedicated interface. In order to increase performance several time critical functions (such as interrupts) are copied from Flash to RAM after controller has booted. For more information on this refer to section 5.1.6 on page 40.

In total 1 MB of Flash and 164 kB of RAM are available as on-chip memory (16 bits per word). All data sections and their storage location are defined in the Linker Command File. For more information on this refer to section 5.1.5 on page 38.

3.2.3. DMA - Direct Memory Access

Direct memory access is a key systems that provides a significant boost to processing power by transmitting data between different modules without utilization of the CPU. Appendix A.1 on page 70 shows which subsystems are connected by the DMA-Bus[5]. The TMSF28377S controller offers a total of 6 identical DMA channels which can be triggered via software or events created by other components such as EPWM, GPIO, ADC, SPI, etc.

Table 3.2 on the following page lists all channels and their assignment. CH1-3 are used to trigger SPI transmissions by writing dummy data to their respective FIFO-buffers. CH4-6 continuously transfer data from a predefined RAM segment to the DAC to create custom analog signals.

For basic operation the following settings are required:

Source/destination address: A physical address or a pointer to the data source and destination. The addresses are defined in shadow registers (`DmaRegs.CH1.SRC_BEG_ADDR_SHADOW`) which are loaded before each new transmission. This allows redefinition of source and destination while a transfer is active.

Transfer/burst size: For each trigger event defined in `DmaClaSrcSelRegs.DMACHSRCSELx.bit.CHy` one transfer is initiated. A transfer can consist of up to 65536 bursts, each holding up to 32 words of data. In our case for every event only

[5]Note that some modules only have limited DMA functionality. See TMS320F2837xS Datasheet Table 6-9 for more details

one burst is generated. Alternatively it is possible to serve each SPI with the same DMA by using 3 bursts and defining a destination burst step size (see below).
Despite allocating more hardware modules it was decided to use one DMA channel per SPI module since it is the simpler approach and allowed for more flexibility during development (i.e. different sampling rates for each ADC). After each transmission `TRANSFER_COUNT` is incremented by one. If `TRANSFER_COUNT = TRANSFER_SIZE` the modules starts over again with the first transfer and the first burst. This can be prevented by enabling one-shot mode.

Source/Destination Transfer/Burst Step: Between each transfer and burst both source and and destination may be incremented or decremented by a predefined step. This is required for CH4-6 to load the next value for the DAC.

Data size: It is possible to transfer 16 and 32 bit words. All channels are set to 16 bit because since destination registers only hold 16 bits.

Other features such as interrupts or overrun detection are not used.

Table 3.2.: DMA channel assignment

DMA Channel	Destination	Source	Trigger	Description
CH1	SPI-A	RAM	EPWM10A	Triggers SPI-A Transmission
CH2	SPI-B	RAM	EPWM10A	Triggers SPI-B Transmission
CH3	SPI-C	RAM	EPWM10A	Triggers SPI-C Transmission
CH4	DAC-A	RAM	EPWM2A	Generate DAC-A Function
CH5	DAC-B	RAM	EPWM3A	Generate DAC-B Function
CH6	DAC-C	RAM	EPWM4A	Generate DAC-C Function

Since all channels use the same hardware bus simultaneous transmissions have to be serialized. This is achieved by using the Round-Robin Mode which gives each DMA equal priority by serving them in the following circular order:

$$\text{CH1} \rightarrow \text{CH2} \rightarrow \text{CH3} \rightarrow \text{CH4} \rightarrow \text{CH5} \rightarrow \text{CH6} \rightarrow \text{CH1} \rightarrow \text{CH2} \rightarrow ...$$

This ensures a constant delay between each SPI transmission since all three channels are triggered by the same source (EPWM10A).

3.2.4. DAC - Digital Analog Converter

The controller offers three buffered 12-bit digital to analog converters which are used as a programmable function generator. For more information see section 5.2.5.

Compared to other peripheral components the DAC is rather simple. The output can vary between 0V and the internal reference (VDAC = 3.3V). Alternatively an

external pin may be used to determine a custom reference voltage (0 < VREFHI < 3.3V). The desired output is set by writing to `DacxRegs.DACVALS`[6] and can be calculated as follows:

$$V_{DAC} = \frac{DACVALS \cdot DACREF}{4096} \tag{3.1}$$

Testing has shown that the DAC has a non-linear behavior if the difference to the voltage boundaries is < 0.3V.

3.2.5. EPWM - Enhanced Pulse Width Modulator

The EPWM offers typical pulse width modulation capabilities i.e. creating pulse signals with varying duty cycles, and incorporate several sub modules for more specialized applications. The following section will only cover features which have been used in this project. A detailed documentation on all sub modules for EPWMs can be found in the TRM Chapter 13.

In total 12 modules (EPWM1-12) with 2 channels each (EPWMxA, EPWMxB) are available which are grouped in a tree structure (see TRM Figure 13-7). This is relevant for synchronization because it is not possible to start all channels simultaneously because a sync-pulse has to propagate through the whole system. This is why time-critical modules are grouped into blocks that synchronize consecutively (EPWM 7-9 and EPWM 10-12).

Table 3.3 lists all used EPWMs and their assigned function.

In order to fully utilize the LTC2380-24 ADC the following sub-modules are used:

Time-Base Submodule(TB): Controls clock rate (EPWMCLK), event timings and synchronization logic. At its core it consists of a counter (TBCTR) that periodically increases its value until it reaches a specified period (TBCTR == TBPRD) which resets TBCTR back to 0[7]. The clock rate is determined by CPUCLK and an additional divider. For this project EPWMCLK is set to 100 MHz for every EPWM which corresponds to 10 ns for every increment of TBCTR.

Action Qualifier Submodule (CC): This submodule is responsible for producing the output using 4 different events:

- PRD: TBCTR == TBPRD

[6] DACVALS is actually a shadow register which is periodically copied to DACVALA. Since the driving frequency is SYSCLK the process may be regarded as instantaneous. If a slower update-rate is required an EPWM can be used as well.

[7] Other modes are possible such as down-count or up-down-count but are not used.

Table 3.3.: EPWM assignment

EPWM No.	GPIO	Connected to	Function
1A	NC	EPWM2-12	sync master
2A	2	DMACH4	Trigger data transfer to DAC-A
3A	4	DMACH5	Trigger data transfer to DAC-B
4A	6	DMACH6	Trigger data transfer to DAC-C
7A	12	ADC-C	Generate conversion pulse
7B	13	Buffer C	Open/Close buffer
8A	14	ADC-B	Generate conversion pulse
8B	15	Buffer B	Open/Close buffer
9A	16	ADC-A	Generate conversion pulse
9B	17	Buffer-A	Open/Close buffer
10A	18	DMACH1-3	trigger data transfer to SPI-A/B/C

- ZRO: TBCTR == 0
- Counter-Compare A: TBCTR == CMPA (user defined)
- Counter-Compare B: TBCTR == CMPB (user defined)

Once TBPRD is equal to one of those events an action qualifier can be used to modify the signal output by setting, clearing or toggling the pins for EPWMxA and EPWMxB. Furthermore interrupts and other modules can be triggered by those events. This feature is used to define the duty cycle of clock signals and trigger interrupts or DMA transfers.

Trip-Zone Submodule (TZ): Trip-Zones can be used to force the output of an EPWM channel to high ort low if certain conditions apply such as the state of a GPIO. Each module is connected to 6 $\overline{\text{TZn}}$ signals however only $\overline{\text{TZ1}}$ to $\overline{\text{TZ3}}$ are linked to GPIOs. This is being used to suppress the generation of CNV-pulses on EPWM7-9 during an SPI transmission.

The maximum period of all EPWMs is limited by

$$T_{max} = \frac{TBPRD_{max}}{f_{CPU}} \cdot EPWMCLKDIV = \frac{\text{0xFFFF}}{\text{200MHz}} \cdot 2 \approx 655\mu\text{s} \approx 1.5\text{kHz} \quad (3.2)$$

While the first part of equation 3.2 is a hardware limitation EPWMCLKDIV can be adjusted. The resulting period is a limit for all EPWMs. Increasing T_{max} enables longer signals but also decreases the possible resolution.

3.2.6. I²C

I²C is a simple serial interface commonly used for microcontrollers. The F28377S controller offers two I²C ports (I2CA, I2CB). The interface is solely used to communicate with the PCF8574A 8-bit I/O expander which provides additional pins for the controller. Note that those pins do not offer the same performance as the F28377S' GPIOs. For more information refer to the PCF8574A data sheet.

3.2.7. SCI / UART

UART is a basic serial protocol for asynchronous transmission of data which is available on almost every modern USB port. The F28377S offers three SCI ports (SCIA, SCIB, SCIC) that are compatible with UART. Compared to other common interfaces it is rather easy to set up and configure at the cost of a slow bandwidth and the potential of flipped bits during longer transmissions. Therefore it was soon replaced by a more powerful and reliable USB interface.

The SCIA port and its wiring is still available if needed but the code base is no longer being maintained.

3.3. Necessary hardware modifications before use

Before all three channels can be used the following modifications to the hardware are necessary:

1. Remove Resistor R44
2. Create a bridge between the previous upper end of R44 and Pin 48
3. Remove R2
4. Create a bridge between U2-OE and ground

The first change (Figure 3.2 - bottom left) is necessary because SPI-C-SOMI is not connected to a pin on the LaunchPad. Pin 48 does not have any other connections or assignment otherwise.

The second change (Figure 3.2 - bottom right) is required to fix an issue that creates random spikes on several EPWMs. The exact cause of this problem could not be evaluated however it seems likely that this is a hardware design flaw. This issue might be resolved on newer versions of the board.

The two jumpers seen in (Figure 3.2 - top left and top right) connect the 5V USB power supply with the rest of the board and have to be removed if the device is mounted on the breakout board.

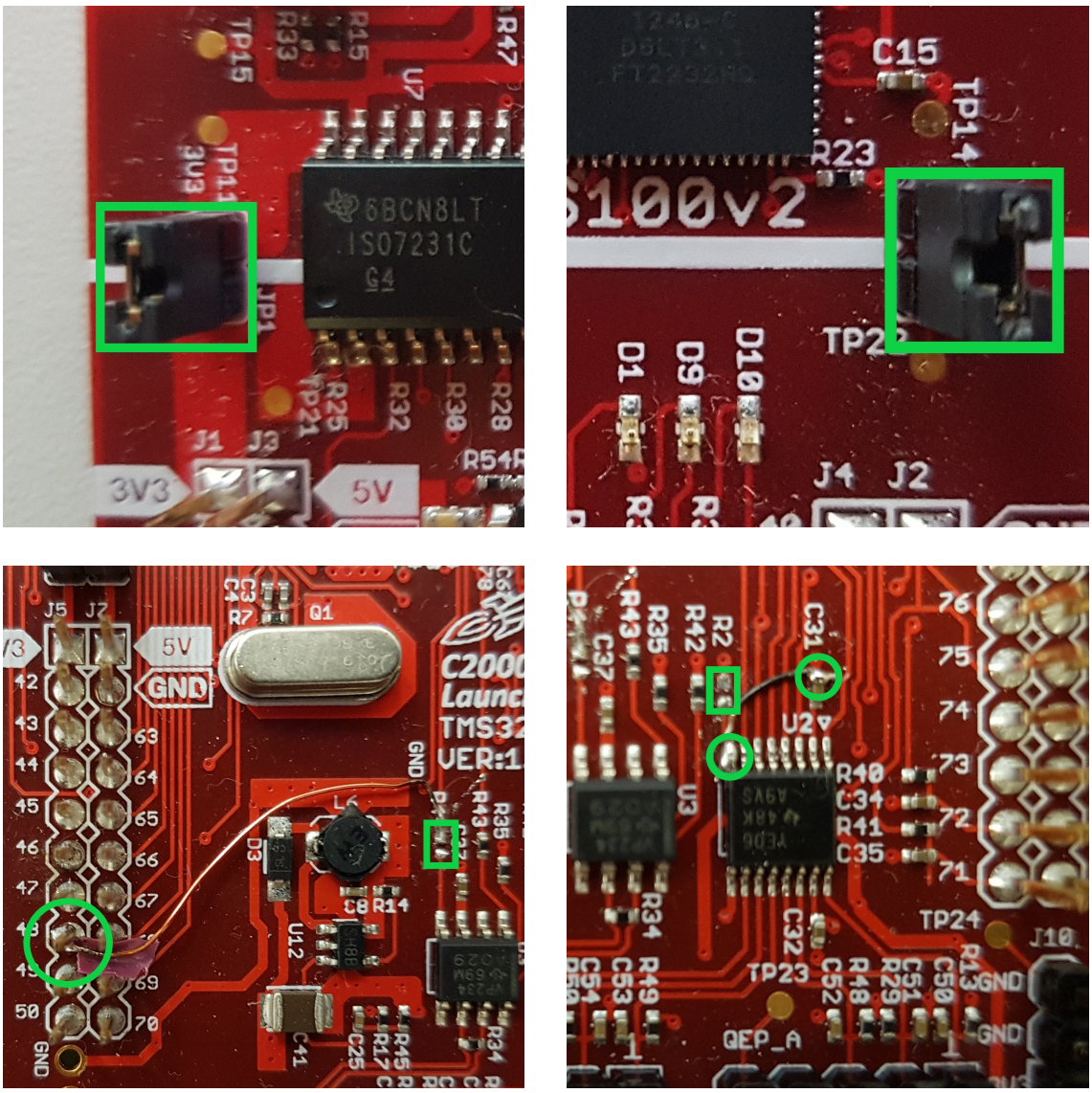

Figure 3.2.: Required changes to the hardware

4. Serial Peripheral Interface

This chapter will cover the Serial Peripheral Interface that is used to transfer data between the ADCs and the controller. In order to represent the operation of the interface on the F28377S controller several listings are used. The programming language, development environment and software structure are covered in Chapter 5.

4.1. General description and features

The Serial Peripheral Interface is a de-facto standard developed by Motorola and often used for short range communication between embedded systems. It offers bi-directional communication and a higher baud rate than similar interfaces such as UART or I²C while maintaining a simple structure. However, several technical details (i.e. clock polarity, transfer format, wire names) are not defined and have to be matched for every part.

4.1.1. Wire types

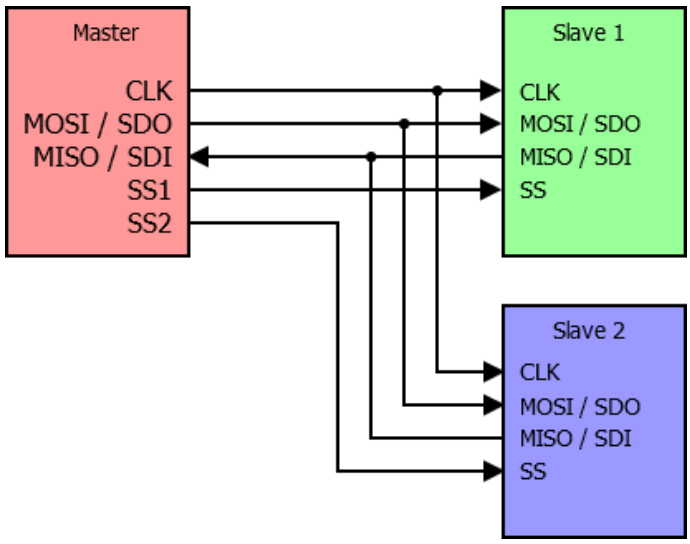

Figure 4.1.: SPI master connected to 2 slaves

The SPI consists of one master and one or more slaves. All devices are connected by 3 wires and one additional wire for every slave (see Figure 4.1). Note that depending on the manufacturer the designated labels may vary.

Serial Clock (CLK/SCLK/SCK): This signal is used to determine the baud rate for all devices by transmitting a pulsed signal from the master to all slaves. It is only active while a transmission is in progress.

Master Output Slave Input (MOSI/SDO): This wire serves as the data output for the master and input for the slave.

Master Input Slave Output (MISO/SDI): This wire serves as the data input for the master and output for the slave.

Slave Select ($\overline{\text{SS}}$): This wire is pulled down to select a slave and initiate communication. For every additional slave, one dedicated wire is required. Only one slave can be active at any given time.

Depending on the manufacturer $\overline{\text{SS}}$ may or may not be part of an SPI module or chip. Since no data is transmitted if the state of $\overline{\text{SS}}$ changes, its task can be accomplished by using a GPIO.

4.1.2. Signal timing

The exact timing of the transmission depends on the chosen mode. The SPI offers 4 different operating modes defined by the state of CPOL and CPHA (see Figure 4.2). Both have to be matched between the master and the slave to ensure proper communication.

Clock Polarity (CPOL): Defines if the clock's state at the start and the end of each transmission is low (CPOL = 0) or high (CPOL = 1)

Clock Phase (CPHA): Defines if the first bit of data is read after one cycle of SCK (CPHA = 1) or on the first edge of SCK (CPHA = 0). The second mode requires the slave's output pin to be already in the correct state before the transmission starts.

Depending on the manufacturer the settings are sometimes referred to Mode 0-3 while others set any naming convention aside. In the case of the LTC2380-24 the author describes the mode as:

> "The conversion result or daisy-chain data is output on [MISO] on each rising edge of SCK MSB first"[2, p. 9]

On the next page a timing diagram is depicting MISO and SCK as phase shifted. This means the LTC2380-24 requires CPOL = 0 and CPHA = 1.

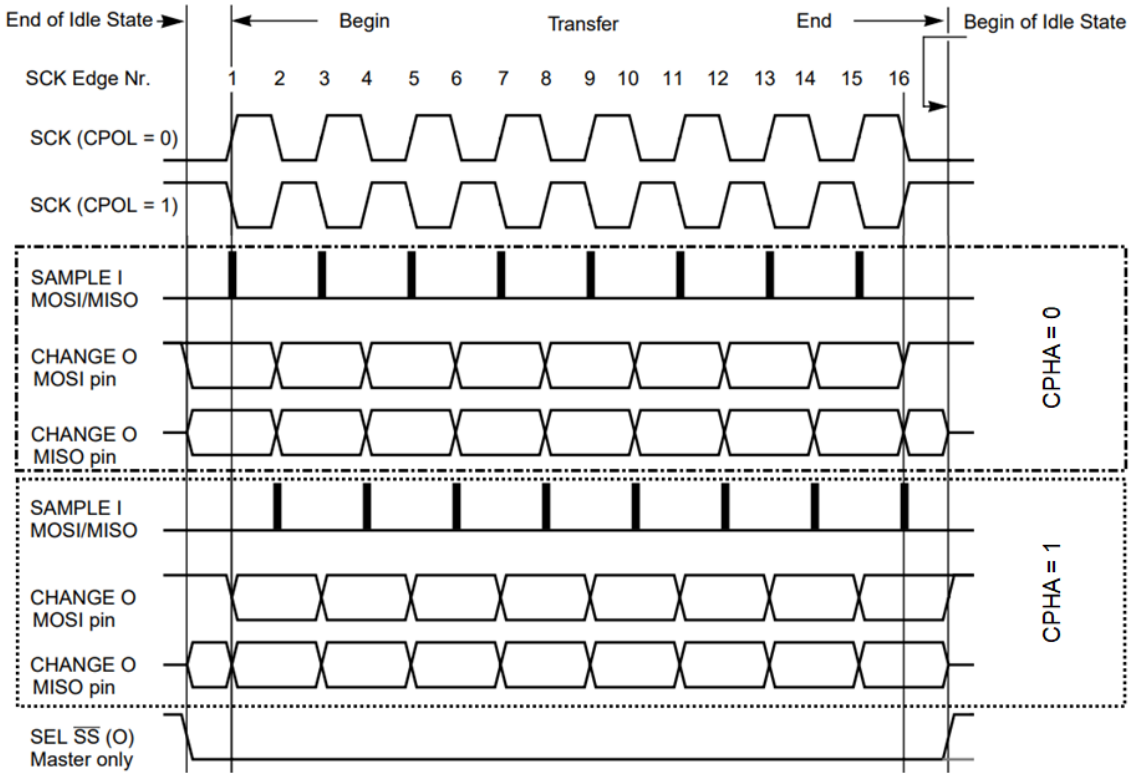

Figure 4.2.: Timing diagram of an 8-bit transmission depending on the selected modes of CPOL and CPHA. SAMPLE I denotes the moment when the states of MOSI and MISO are interpreted as data bits.

The frequency of all signals is determined by the baud rate of SCK and is limited by the slowest part of the system. The F28377S SPI module has a maximum baud rate of 50MHz while the LTC2380-24 is able to perform at up to 100 MHz. However, the LTC2400 has an upper limit of only 5 MHz. This means that if the master is communicating with the slower ADC the baud rate has to be adjusted accordingly. If the LTC2380-24 is selected the frequency is set to 50 MHz.

4.2. The F28377S SPI

The F28377S controller comes with 3 identical SPI modules (SPI-A, SPI-B and SPI-C) that can operate independently of each other. This makes the controller an ideal choice for the experiment since a constant data stream from 3 detector boards has to be processed.

The SPI module of the F28377S controller is linked to several other peripheral systems (see Figure 4.3). This allows to trigger interrupts depending on the state of the module or transmit data to the SPI using DMA.

One distinctive feature is the replacement of $\overline{\text{SS}}$ with $\overline{\text{SPISTE}}$ which switches to

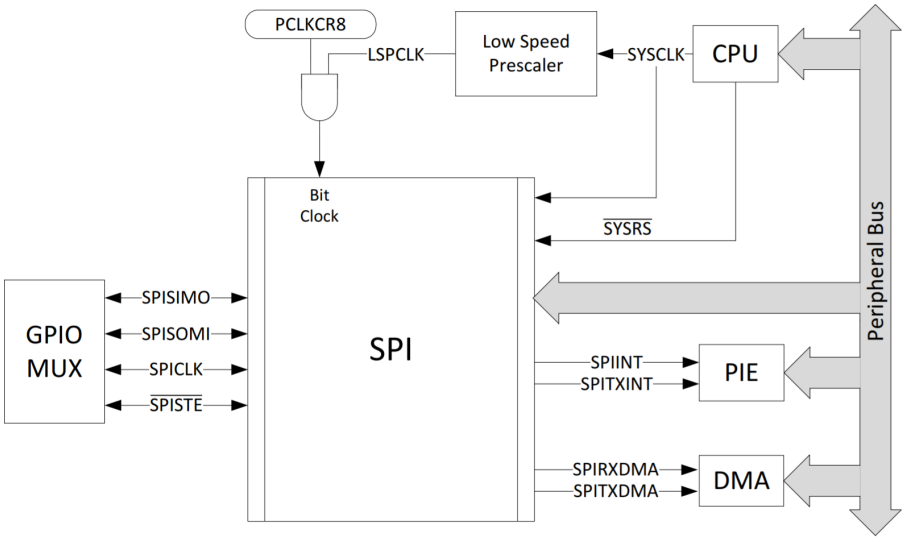

Figure 4.3.: The F28377S SPI structure and connections [5, Figure 17-1]

low, shortly[1] before a transmission is initiated. Therefore, if two or more slaves are connected to the master, dedicated GPIOs have to be used to serve as $\overline{SS1}$, $\overline{SS2}$, etc.

Note that for this project $\overline{SPISTE}$ is also used to suppress other signals and control the behavior of external hardware components. For more information refer to Section 5.2.3 on page 45.

The clock source for the SPI is LSPCLK which is set to 100MHz. The baud rate is defined using the register `SPIBRR` and can be calculated using the following formula:

$$SPIBAUD\,[MHz] = \frac{LSPCLK\,[MHz]}{SPIBRR + 1} \tag{4.1}$$

In order to achieve a maximum bandwidth of 50 MHz, the documentation recommends enabling high speed mode (HS_MODE = 1). This option is available on all three modules but only on certain high-speed capable pins as shown in Table 4.1 below.

The difference between HS_MODE = 1 and HS_MODE = 0 and the underlying logic of those high-speed pins is not documented. Further testing has also shown that 50 MHz can be achieved on other pins as well. It is possible that those pins offer improvements such as a better slew rate but this could not be tested due to limitations of the available oscilloscope.

[1] The exact timing of this is not documented.

Table 4.1.: GPIOs that support SPI high-speed mode

Name	SPI-A	SPI-B	SPI-C
SPISIMO	GPIO58	GPIO63	GPIO69
SPISOMI	GPIO59	GPIO64	GPIO70
SPICLK	GPIO60	GPIO65	GPIO71
$\overline{\text{SPISTE}}$	GPIO61	GPIO66	GPIO72

Every module includes a pair of 16-level FIFO (First In - First Out) hardware buffers for transmitted and received data. These buffers can be accessed by the CPU and DMA and generate interrupt signals. The associated logic is shown in Figure A.5 on page 72

Transmissions are not limited in length but have to be split into words of up to 16 bits. If one or more word is written to the FIFO buffer the data is serialized accordingly as long as the buffer limit is not exceeded. For more information refer to Section 4.3.3 on page 31.

4.3. Implementation on the controller

4.3.1. Initializing the SPI

Before the SPI module can be used several settings have to be applied before it is operational. Before any changes are made it is recommended to put the module into reset mode by setting `SPISWRESET` = 0. This disables all transmissions and ignores all status flags such as OVERRUN_FLAG (SPI Receiver Overrun Flag) or INT_FLAG (SPI Interrupt Flag). Table 4.2 shows all values that are written to the SPI when `void SpiInit()` is called. Bits and registers that are left with their default values are not covered. An extensive list of all registers can be found in the Technical Reference Manual Tables 17-8 to 17-20.

Table 4.2.: Description of all values that are written to the SPI registers

Bit	Name	Value	Description
SPICCR - SPI Configuration Control Register			
7	SPISWRESET	*	SPI Software Reset This bit is set to 0 whenever modifications to the control registers are made.
6	CLKPOLARITY	0x0	Clock Polarity Defines the polarity of SPICLK as described in subsection 4.1.2.
5	HS_MODE	0x1	High Speed Mode Enable Bit

Bit	Name	Value	Description
4	SPILBK	0x0	SPI Loopback Mode Select Loopback Mode is mainly required for debugging or development. When SPILBK = 1 all outgoing transmission are fed back into SPIDAT. This can be used to emulate a transmission or to check if the device is working properly
3-0	SPICHAR	0xB	Character Length Control Bits Defines the number of bits that are shifted out for every transmission. 0x0 = 1 bit per transmission 0x1 = 2 bits per transmission ... 0xF = 16 bits per transmission The transmission of data is covered in Section 5.2.3 on page 45.
SPICTL - SPI Operation Control Register			
4	OVERRUNINT ENA	0x0	Overrun Interrupt Enable Whenever a transmission completes before the previous character is read from the buffer and overrun flag is generated. This does not happen if the SPI is operating in FIFO mode
3	CLK_PHASE	0x1	SPI Clock Phase Defines the phase of SPICLK as described in subsection 4.1.2.
2	MASTER_ SLAVE	0x1	SPI Network Mode Control The SPI has to operate as a master in order to issue transmission from the ADC 0x0 = slave mode 0x1 = master mode
1	TALK	0x1	Transmit Enable If the SPI is configured in as a master this bit has the following effect: 0x0 = SPISOMI is put in the high-impedance-state and no outgoing data is transmitted 0x1 = outgoing transmission are enabled Since the ADC only responds to the SPICLK this bit has no effect on this project

Bit	Name	Value	Description
0	SPIINTENA	*	SPI Interrupt Enable Defines if an interrupt flag is generated once a transmisison is complete. SPIINTENA is set to 1 for SPIA and to 0 for SPIB and SPIC. Since all three transmissions happen simultaneously only one interrupt is required to read out all FIFO-buffers.
SPIBRR - SPI Baud Rate Register			
6-0	SPI_BIT_RATE	0x1	SPI Baud Rate Control Defines the bit rate accoring to formula 4.1
SPIFFTX - SPI FIFO Transmit Register			
14	SPIFFENA	0x1	Enable SPI FIFO Enhancements
13	TXFIFO	0x1	TX FIFO Reset This bit can be used to flush the FIFO. In order to receive more transmission the value has to be set back to 0x1
12-8	TXFFST	*	Transmit FIFO Status 0x0 = TX FIFO is empty 0x1 = TX FIFO has 1 word ... 0x10 = TX FIFO has 16 words and is full
5	RXFFIENA	0x0	Disable all TX FIFO Interrupt signals
4-0	RXFFIL	0x0	Transmit FIFO Interrupt Level Bits 0x0 = A TX FIFO interrupt request is generated when there are no words in the TX buffer This field has no affect since RXFFIENA = 0 by default.
SPIFFRX - SPI FIFO Receive Register			
14	SPIFFENA	0x1	Enable SPI FIFO Enhancements
13	TXFIFO	0x1	RX FIFO Reset This bit can be used to flush the FIFO. In order to receive more transmission the value has to be set back to 0x1
12-8	RXFFST	*	Receive FIFO Status 0x0 = RX FIFO is empty 0x1 = RX FIFO has 1 word ... 0x10 = RX FIFO has 16 words and is full
5	RXFFIENA	0x1	RX FIFO Interrupt Enable This bit has no effect on SPIB and SPIC since they are not generating interrupt events

Bit	Name	Value	Description
4-0	RXFFIL	*	Receive FIFO Interrupt Level Bits This register controls the interrupt event that is generated after a certain amount of words is stored in the FIFO. 0x1F = no interrupt is generated 0x10 = an interrupt is generated when the FIFO is full (16 words) RXFFIL is set to 0x10 for SPIA and 0x1F for SPIB and SPIC.

4.3.2. Initiating transmissions

Once the SPI module has been properly configured data can be transmitted. The simplest way of starting a transmission is to write data directly to SPIDAT. However, it is highly recommended to use the the FIFO by writing to SPITXBUF instead in order to prevent overrun errors. If the first two lines of Code 4.3.2 were to be swapped new data might be written over an active transmission leading to undefined behavior.

Code 4.1: Example of a transmissions while loop back mode is enabled (`SpiaRegs.SPICCR.bit.SPILBK = 1;`)

```
SpiaRegs.SPIDAT = 42;  // write directly. Not recommended!
SpiaRegs.SPITXBUF = 1337; // write data to FIFO buffer

// wait until transmissions have been completed
while(SpiaRegs.SPIFFRX.bit.RXFFST !=0) {}

int rdata1 = SpiaRegs.SPIDAT; // read directly. Not recommended!
int rdata2 = SpiaRegs.SPIRXBUF; // read from buffer
int rdata3 = SpiaRegs.SPIRXBUF;

// Output:
// rdata1 = 42
// rdata2 = 1337
// rdata3 = 0 (buffer is empty)
```

Since the SPI standard always requires an equally sized response from the slave, incoming data is shifted into SPIDAT as seen in Figure 4.4. After all bits have been transmitted the received message is copied to SPIRXBUF. Once a word is read from SPIRXBUF the FIFO level (`SpiaRegs.SPIFFRX.bit.RXFFS`) decrements by

CLK	SPIDAT							
0	O0	O1	O2	O3	O4	O5	O6	O7
1	O1	O2	O3	O4	O5	O6	O7	I0
2	O2	O3	O4	O5	O6	O7	I0	I1
3	O3	O4	O5	O6	O7	I0	I1	I2
4	O4	O5	O6	O7	I0	I1	I2	I3
5	O5	O6	O7	I0	I1	I2	I3	I4

⟵ shifted out

Figure 4.4.: Data in SPIDAT during an 8-bit transmission over the course of several cycles. The bits O0 - O7 represent outgoing data. I0-I7 represents incoming data

1. If `RXFFST == 0` the FIFO is considered empty and will only return 0. Therefore received data must be stored in RAM for further processing.

Both SPITXBUF and SPIRXBUF can only hold up to 16 words. If `RXFFST == 0x10` and another transmission is initiated an overflow-flag is generated (`SpiaRegs.SPIFFRX.bit.RXFFOVF == 1`) and the next received word is lost. This flag can be used to trigger an interrupt.

4.3.3. Serializing data

If data is written to the TX FIFO buffer before the previous transmission is completed the module will automatically serialize the data. However, testing has shown that this can lead to unexpected behavior depending on the program flow.

Consider a function similar to 4.3.3 below which generates an SPI transmission generates followed by a short pulse. Since the F28377S is only features a single-core CPU all operations have to be executed in a strict consecutive order which would imply an alternating output of a transmission and a pulse on the respective GPIO and SPI pins.

Code 4.2: Example of generating a short pulse followed by a 4bit SPI transmission in two different ways

```
1 // Generate SPI signal first
2 void signal1()
3 {
4    int i;
5    for(i = 0; i < 3; i++)
6    {
7       SpiaRegs.SPITXBUF = 0xAA; // Send data to SPIA for transmission
8       GPIO_WritePin(DEBUG_PIN, GPIO_HIGH);  // Generate Pulse
9       GPIO_WritePin(DEBUG_PIN, GPIO_LOW);
```

```
10    }
11 }
12
13 // Generate GPIO signal first
14 void signal2()
15 {
16    int i;
17    for(i = 0; i < 3; i++)
18    {
19       GPIO_WritePin(DEBUG_PIN, GPIO_HIGH);
20       GPIO_WritePin(DEBUG_PIN, GPIO_LOW);
21       SpiaRegs.SPITXBUF = 0xAA;
22    }
23 }
```

However, many peripheral systems are able to perform actions without further instructions from the CPU. This is especially important for the SPI since the CPU only writes the value 0xAA into a FIFO-Buffer (`SpiaRegs.SPITXBUF`). After this instruction is complete the SPI loads the next word into `SpiaRegs.SPIDAT` and starts the transmission on its own. While this is happening, more data may be written into the buffer as long as a maximum of 16 words is not reached otherwise data is lost due to an overflow error.

Similar to most other peripheral components the SPI is driven by LSPCLK which is set to 100MHz. Those sub modules therefore operate on a slower clock frequency than the CPU which may cause timing issues and unexpected behavior. Code 4.3.3 is written to demonstrate this issue: While `void signal1()` outputs the expected signal, `void signal2()` creates a delay between the second and third SPI transmission (see Figure 4.3.3).

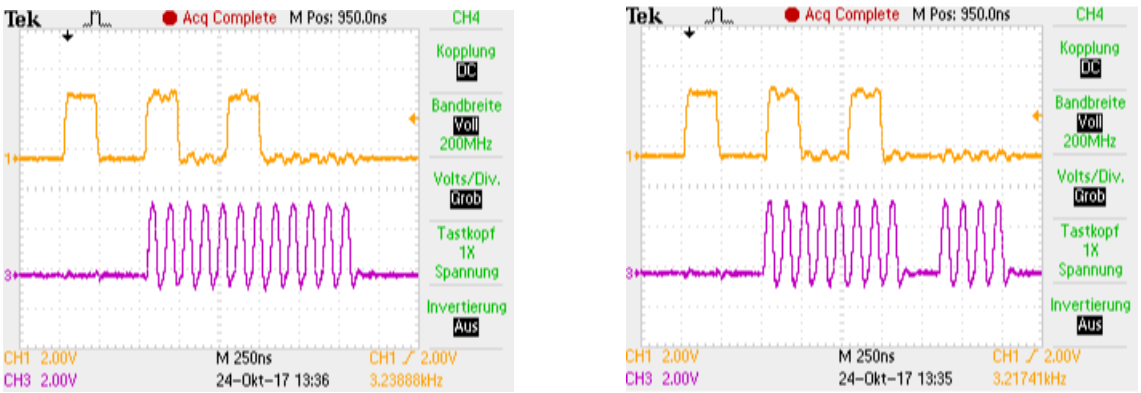

Figure 4.5.: Output signal of `void signal1()`(left) and `void signal2()` (right) from Code 4.3.3

The reason for this is not documented but testing revealed the following behavior:

- Setting a GPIO, transmitting data or any other peripheral operation is linked to a delay of up to 250ns between the initial instruction and its execution.
- The moment of execution of most peripheral operations is synchronized across the whole chip (events happen in "ticks") i.e.: in Code 4.3.3 the data transmission starts at the same time as the second pulse.
- All delays scale with LSPCLK.
- This does not apply to EPWM modules

A possible explanation is the use of synchronized shadow registers similar to those used in the DAC sub module. Every 25 LSPCLK cycles a sync-pulse is generated and data is loaded into the actual register. In the case of Figure 4.5 (right) this sync-pulse happens after the instruction to set the GPIO to low but before data is written to `SpiaRegs.SPITXBUF,` and therefore the SPI transmission is delayed until the next system tick.

This means that while many peripheral modules offer great performance by being able to operate in parallel, the exact timing of those operations may vary. This effect leads to certain issues such as the inability to synchronize SPI and EPWM signals together.

5. Software

The following chapter offers a brief overview of the framework and the IDE that is used to develop applications for C2000 controllers. The main part of this chapter will outline the project and software structure created for the F28377S controller and provide a detailed explanation of the software integration of used modules and their interaction with each other.

5.1. The Texas Instruments development environment and programming language

TI offers several different ways to develop and launch applications for their controllers:

1. Code Composer Studio is the most prominent IDE. Its main purpose is to provide an all-in-one tool for every platform and it comes with a wide range of debugging tools and compilers. In order to use CCS in combination with the F28377S controller, controlSUITE[1] must be installed which provides essential frameworks, definitions, examples and useful libraries.
2. Energia is a fork of the Arduino IDE that closely resembles its user interface. It started out as a community driven open source project and was later integrated in TI's development tool portfolio. Energia is essentially a simplified version of CCS for beginners and small projects.
3. CCS cloud is a web-based alternative. It was released after this project started and therefore, not used.
4. Other platform specific solutions such as HALCOGEN are also available but may not support every platform.

For this project the following software versions have been chosen:

- CCS v6.2
- Compiler TI v15.12.4 LTS
- controlSUITE v3.4.5

[1]Since December 2017 controlSUITE has been replaced by a new release called C2000WARE. The new software has not been adapted due to several compatibility issues.

It is highly recommended to stick to those versions even if updates are available. Compiler version 16.6.0 or higher for example breaks in combination with the newest version of controlSUITE.[2]

5.1.1. Programming language

CCS does support C and C++, however both make use of some non-standard techniques. For example `#pragma` is used to map certain variables to physical memory sections.

Code 5.1: An example of non-standard C-syntax for TI compilers

```
1 // Map dacFunc_data to a physical data section
2 #pragma DATA_SECTION(dacFunc_data, "dacFunc_data_sec");
```

Code 5.1 however does not compile in C++ because `#pragma` does not name the entity it operates on[8] (in this case `dacFunc_data`). This leads to the problem that the C++ compiler for the C2000 platform may not be able to process certain header files.

Furthermore some C++ features such as exceptions and polymorphism can have a significant performance penalty even if not used [9].

Since performance is a limiting factor and almost all libraries, source files and examples that are shipped with controlSUITE are written in C, the decision was made to write the complete project in C. The only exception to this are some time critical routines (such as FIR-fitlering) which are included in assembly.

5.1.2. F2837xS Support Library

As mentioned in the beginning of this chapter controlSUITE is used as a supporting library. It is a collection of headers, algorithms and examples for the whole C2000 series. For this project v210 is being used which was released in November 2016. If controlSUITE was installed before that an update is required.

The most important part of this framework are its register bit definitions. All settings that dictate the behavior of the CPU and its peripheral modules are defined by internal registers. The only way to access them is to read or write to the physical hardware address they are mapped to.

Each module has a base address such as `0x0000_6100` for `SpiaRegs`. A specific register can be modified using the offset defined in the TRM and by writing the corresponding 16 or 32 bit value (depending on the register size) to it. Alternative single bits can also be set or cleared using bit wise operators (AND, OR, XOR, etc.).

[2] This bug has been acknowledged by now. For more information visit: e2e.ti.com/support/microcontrollers/c2000/f/171/t/579091

17.5.2.10 SPIFFRX Register (Offset = Bh) [reset = 201Fh]

SPIFFRX is shown in Figure 17-23 and described in Table 17-18.

Return to Summary Table.

SPI FIFO Receive Register

Figure 17-23. SPIFFRX Register

15	14	13	12	11	10	9	8
RXFFOVF	RXFFOVFCLR	RXFIFORESET	RXFFST				
R-0h	W-0h	R/W-1h	R-0h				
7	6	5	4	3	2	1	0
RXFFINT	RXFFINTCLR	RXFFIENA	RXFFIL				
R-0h	W-0h	R/W-0h	R/W-1Fh				

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; *-n* = value after reset

Figure 5.1.: An example from the Technical Reference Manual. This register is part of the groups SpiaRegs, SpibRegs and SpicRegs

Since this is a very tedious and error-prone process the F2837xS Support library offers a simple syntax that matches the register names listed in the Technical Reference Manual (see Code 5.2).

Code 5.2: Modifying registers

```
// modifying registers by writing to their physical address
int16_t *reg = (int16_t *)(0x6100 + 0xB); // Spiaregs + SPIFFTX
reg |= 0x2000; // set only bit 14
reg  = 0x2040; // write multiple bits at once

// the same process but using the support library's syntax
SpiaRegs.SPIFFRX.bit.RXFIFORESET = 0x1; // setting one bit
SpiaRegs.SPIFFRX.all = 0x2040; // setting multiple bits
```

The support library also offers functions that are used for almost every project such as initializing the PIE vector table (`InitPieVectTable()`) and macros such as `ESTOP0` (halt the processor) or `EALLOW` (enable write protection).

In order to load all functions and macros to a project include *"F28x_Project.h"*.

5.1.3. Debugging

Code Composer Studio offers a wide range of different debugging tools that are expected to be present in every modern IDE such as break points, stepping through code, watching variables, etc. However CCS also provides several other very powerful features which will be outlined in the following section.

Register Editor: (View > Registers) During development this was by far the most used feature. The register view offers a list of all (!) registers that define the behavior of the CPU and its peripheral modules. This also includes registers

protected by the EALLOW-mechanism (see Section 5.1.4) except read-only bits. Using the drop-down menus single bits can be changed at run time to quickly observe the effect on the controller. If the device is rebooted all changes are lost.

View Memory: (Right click on a watched expression > View Memory) This opens a separate window listing the actual memory contents including stack and heap. The list is sorted by the physical address and can be modified at run time.

Memory Allocation: (View > Memory allocation) Shows all memory sections defined in the Linker Command File (see Section 5.1.5) using bar graphs. The length of each bar represents the utilization of each section. This can be useful to optimize RAM usage. Note that this only represents the statically defined sections and it is not possible to make changes at run time.

Terminal: (View > Other > Terminal > Terminal) Except for the debugger the F28377S controller does not offer default communications interface. The simplest way to transmit data back to the host is using SCI/UART. CCS offers a basic terminal similar to Putty or Hyperterminal to display data. Note that for this project the serial interface has been replaced by USB and a custom GUI.

Disassembly (View > Disassembly) Displays the actual instructions for the CPU along with an arrow indicating the current program step.

5.1.4. Write Protection

Many registers that hold settings which are critical to the core system operation are protected by the EALLOW mechanism. This feature secures certain registers from being unintentionally overwritten (i.e. by buffer overflows). In order to write data to protected registers the macro EALLOW can be used to disable the mechanism (see Code 5.8). It is highly recommended to use EDIS afterwards to re-establish protection.

Code 5.3: Writing to protected registers

```
1 // all CPU clock registers are EALLOW protected
2 ClkCfgRegs.SYSCLKDIVSEL.all += 1; // has no affect
3 EALLOW; // lift protection
4 ClkCfgRegs.SYSCLKDIVSEL.all += 1; // bit can now be set
5 EDIS; // restore protection
```

5.1.5. Linker Command File

One of the downsides of not having an operating system is the absence of automated memory management. Although it would be possible to design such a system most

of the time it is much easier to use a linker command file (LCF). This file is basically a blueprint of how the internal memory is structured. The TI support library already ships with a selection of different files for different applications but with the addition of USB, digital filtering and the need for bigger data buffers a custom LCF had to be written to accommodate those requirements.

In total three different linker command files have been written:

2837xS_Flash_3Ch_mode: standard configuration which utilizes all three channels and USB.

2837xS_Flash_Single_Ch_mode.cmd: special configuration for debugging purposes where only one channel is active to collect a bigger set of data.

2837xS_RAM_lnk_cpu1_USB.cmd (deprecated): Currently not maintained since the program no longer fits purely on RAM while still keeping reasonable big data buffers.

Note that only one can be active at any time. To change the file associated with the current build configuration right click on the LCF (in CCS) > Exclude from build.

The first part (`MEMORY`) of the LCF allocates memory segments using the physical starting address and its length. The second part (`SECTIONS`) defines different sections and links them to a physical memory segment.

Code 5.4: An exemplary linker command file (incomplete)

```
MEMORY
PAGE 0:
{
   // Ram sectors
   // Part of M0, BOOT rom will use this for stack
   BEGIN          : origin = 0x080000, length = 0x000002
   BOOT_RSVD      : origin = 0x000002, length = 0x000120
   RAMM1          : origin = 0x000400, length = 0x000400
   RAMD1          : origin = 0x00B800, length = 0x000800
   RAMGS0         : origin = 0x00C000, length = 0x001000
   RAMGS1         : origin = 0x00D000, length = 0x001000

   // Sample buffers
   SAMPLE_CB_A    : origin = 0x014B00, length = 0x002500
   SAMPLE_CB_B    : origin = 0x017000, length = 0x002500
   SAMPLE_CB_C    : origin = 0x019500, length = 0x002500
   RESET          : origin = 0x3FFFC0, length = 0x000002

   // Flash sectors
   FLASHA         : origin = 0x080002, length = 0x001FFE
   FLASHB         : origin = 0x082000, length = 0x002000
   FLASHC         : origin = 0x084000, length = 0x002000
}
```

```
24
25 SECTIONS
26 {
27    // Required for basic operation
28    .cinit          : > FLASHB     PAGE = 0, ALIGN(4)
29    .pinit          : > FLASHB,    PAGE = 0, ALIGN(4)
30    .text           : >> FLASHB | FLASHC PAGE = 0, ALIGN(4)
31    codestart       : > BEGIN      PAGE = 0, ALIGN(4)
32
33    // sections for sample buffers
34    cb_a_sec        : > SAMPLE_CB_A,  PAGE = 0
35    cb_b_sec        : > SAMPLE_CB_B,  PAGE = 0
36    cb_c_sec        : > SAMPLE_CB_C,  PAGE = 0
37 }
```

In order to write custom LCFs it is recommended to modify one of the files provided by TI. For more information on memory management and how to write LCFs see processors.wiki.ti.com/index.php/Linker_Command_File_Primer.

In order to actually use the allocated space the defined sections have to be connected to a regular pointer. The syntax is as follows:

Code 5.5: Linking section `cb_a` to a pointer in a .c-file

```
1 #pragma DATA_SECTION(cb_a, "cb_a_sec"); // [pointer][section]
2 int32_t *cb_a; // has to be declared globally
```

Code 5.5 is unusual in a sense that a variable is used before it is declared (another difference to standard C). It still works because the first line is only parsed by the preprocessor while the second line is processed by the compiler. Furthermore the pointer has to be declared globally.

5.1.6. Running code from RAM

One way of optimizing code which has to be stored on Flash is to copy time critical functions to RAM to allow for faster execution. This can be achieved by using the `__attribute__((ramfunc))`-macro provided by the support library. Code 5.6 shows the necessary steps. Note that the required memory is not shown in the memory allocation view of CSS since this step is performed after compiling the program.

Code 5.6: Loading functions to RAM

```
1 // required for macros
2 #define WORDS_IN_FLASH_BUFFER 0xFF
3 #define PUMPREQUEST *(unsigned long*)(0x00050024)
```

```
#define ramFuncSection ".TI.ramfunc"

// load the following function to ram for faster execution
__attribute__((ramfunc))
void importantFunction()
{
   // Everything in here will be executed from RAM
}

void main()
{
   // Copy all makred funcitons to RAM
   memcpy(
      &RamfuncsRunStart,
      &RamfuncsLoadStart,
      (size_t)&RamfuncsLoadSize)
   );

   importantFunction();
}
```

5.1.7. Interrupts

Interrupts are signals triggered by hardware or software which cause the CPU to suspend its current operation and switch to a different subroutine. Once the end of said routine has been reached the CPU switches back to its initial operation before the interrupt occurred. Almost every module of the F2837xS controller can emit one or more interrupt signals.

Interrupt service routines (ISRs) are marked with the `__interrupt`-macro. Only one such function can be linked to an interrupt source. Code 5.7 shows the basic process of enabling an interrupt and connecting an ISR to it. Once the interrupt signal is emitted the CPU will jump out of the loop and service `myISR()`. The ISR can access global and local variables. After completion the program will jump back to the loop and local variables are cleared.

An interrupt is considered to be serviced once it is acknowledged in the PIE control register by setting the corresponding bits in `PieCtrlRegs.PIEACK`. Otherwise any other interrupt signal of the same group is ignored.

Code 5.7: Example of an ISR that is triggered once the FIFO-buffer of SPI-A has reached maximum capacity

```
__interrupt void myISR(void)
{
   // Code executed during the interrupt
```

```
4
5     // Acknoledge Interrupt once its completed
6     PieCtrlRegs.PIEACK.all |= 0x20; // group 6
7  }
8
9  void main()
10 {
11    // Initialization. Functions are provided by the support library
12    InitPieCtrl();
13    InitPieVectTable();
14
15    // Connecting signals
16    EALLOW; // Lift EALLOW protection
17    PieCtrlRegs.PIEIER6.bit.INTx1 = 1; // Enable PIE Group 6, Interrupt 1
18    PieVectTable.SPIA_RX_INT = &myInterrupt; // Connect function
19    EDIS; // lock protected registers
20
21    EINT; // Enable interrupts
22
23    for(;;) {} // loop forever until an interrupt occurs
24 }
```

Nested interrupts are disabled and therefore only one interrupt may be active at any time. If two or more interrupts are pending they are serviced consecutively according to the TRM Table 2-2. This means that USB transmissions have a lower priority than signals emitted by the SPI module.

Figure 5.2 shows the required steps the CPU has to perform before an ISR can be serviced. This results in a minimum latency of 14 cycles (70ns at 200MHz clock speed). For more information refer to TMS320C28x CPU and Instruction Set Chapter 3.

5.2. Software Documentation

5.2.1. Project Structure

All files required to compile and upload the program are available at `github.com/hitimr/Scanner_Controller`. The project itself is available in several different configurations:

RELEASE: This configuration is intended for regular operation. It's settings are optimized for speed but only provides limited debugging functionality. Setting break points in the IDE does not work reliably either. Code enclosed with the `_DEBUG` symbol is ignored by the compiler as shown in Code 5.8.

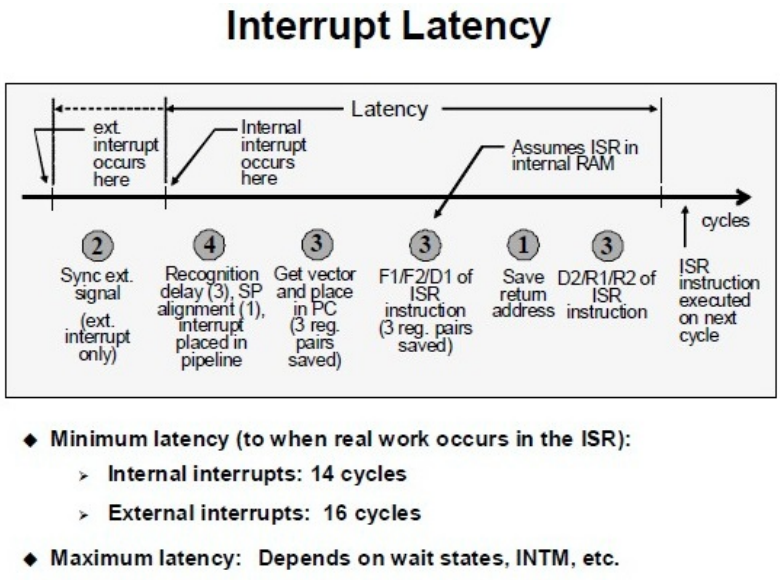

Figure 5.2.: Standard operation for interrupts. Original image from the C2000 Workshop material

Code 5.8: Excluding code from the release build

```
#ifdef _DEBUG
   SciMsg("Debug Mode active\n\r\0"); // ignored by the compiler
#endif
```

DEBUG_FLASH: For development and testing it is recommended to use this configuration. It offers the same features as RELEASE but has a bigger memory footprint and offers extended debugging functionality.

DEBUG_SINGLE_CH: This configuration is similar to DEBUG_FLASH but the channels for SPI-B and SPI-C are disabled. The extra memory is added to SPI-A thus tripling the maximum amount of saved measurement values. This is useful for recording long uninterrupted sets of data.

DEBUG_RAM (deprecated): Loading data to flash memory is a slow process. To speed up development process only release builds are usually loaded to flash. However due to the high memory footprint of the USB interface this configuration is no longer available because it would exceed the available RAM. Note that in contrary to FLASH, RAM is volatile and therefore all data is lost after power is disconnected.

The configuration can be changed by right-clicking on the project > Build Configurations > Set Active. Code Composer Studio creates a unique set of binaries for every configuration and puts them in a separate folder.

The actual source code is located in *src* which is split into several modules.

src\cmd contains the linker command files for every configuration. Only one file can be loaded per configuration.

src\F2837xS files provided by TI are located here. It is not advised to change its contents.

src\math Contains all required methods for digital signal processing and various other useful mathematical functions

src\peripherals Initialization and operation routines for all peripheral modules except USB which is located in a separate folder due to its size.

src\system Core systems and interrupt routines

src\usb Contains all methods and handlers for the USB module

Furthermore two additional files are placed in the source folder:

src\main.c Contains the program entry point.

src\Project.h Contains various global definitions and project settings

5.2.2. Software Structure

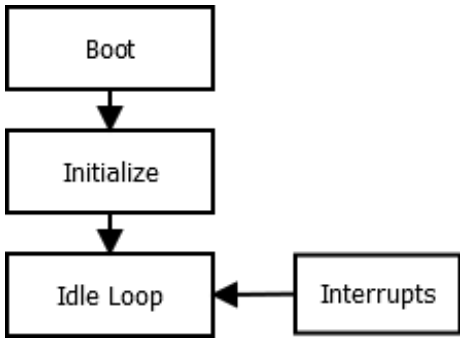

Figure 5.3.: Fundamental software structure.

Except for the boot process the whole program is based on interrupt service routines. After initialization the program runs into an empty idle loop. This loop is suspended by interrupts that are generated by events such as a full SPI buffer or a USB transmission. After the interrupt service routine (ISR) has been completed the program returns into its idle mode. The most important interrupt routines are:

void main_ISR() is the standard ISR which is called every time the SPI FIFO-buffer reaches 16 words. For more information refer to section 5.2.4 on page 47.

void Record_HW_ISR() `main_ISR()` can be replaced with this routine to record one set of data.. After the buffer has reached its maximum capacity the ISR replaces itself with `main_ISR()` and saving new measurements is disabled.

uint32_t RxHandler() handles incoming USB transmissions. It is built upon the USB bulk driver example v210 shipped with control suite. For more information refer to the USB interface documentation.[11]

void ILLEGAL_ISR() is the default error ISR. The device is halted if this interrupt is generated.

5.2.3. Data transmission

In order to achieve maximum performance the complete data processing routine is designed to incorporate the least amount of interrupts necessary and handle as much workload as possible during one routine. This minimizes the overhead and delay that occurs before and after every interrupt.

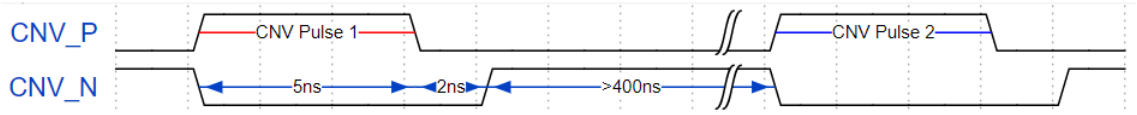

Figure 5.4.: Pulsed signals created for the ADC. CNV_P triggers a conversion and CNV_N opens the respective buffer for CNV_P to propagate.

The cycle starts out with EPWM7A (SPIA), EPWM8A (SPI-B) and EPWM9A (SPI-C) generating conversion pulses (CNV_P). Each pulse is 5ns long and tells the ADC to take 1 sample which takes 343 - 392ns. During that time the ADC is unable to perform any other actions and its BUSY indicator is set to high. It would possible to start the next conversion as soon as this pin switches back to low again however this would lead to an unknown delay that may be different for every sample. Therefore a fixed delay of 400ns (CNV_PERIOD) is used between every conversion.

As a protective measure against high frequency signals and noise, a hardware buffer is situated on the break-out board. EPWM7B, EPWM8B and EPWM9B (CNV_N) are used to open those buffers by switching to low thus allowing CNV_P to propagate. Ideally CNV_N is toggled shortly before and after CNV_P. However due to hardware limitations only one such delay is possible.[3]

Testing has shown that aligning the falling edge of CNV_P with the rising edge of CNV_N, leaves the buffer output signal in an undefined state between high and low. Therefore making a trailing delay of CNV_N absolutely necessary. The initial simultaneous toggle of CNV_P and CNV_N has not shown any effect on the ADC or the buffer.

[3]The reason is that EPWM signals can only be toggled at the start of each counter period and at 2 custom points. TBCTR = 0 is used to toggle both signals simultaneously. CMPA and CMPB are used to toggle the trailing edges.

After a certain amount of transmissions (CNV_NUM) have been performed EPWM10 is used to create an additional delay before the SPI transmission starts (see Figure 5.5). The falling edge of the Delay EPWM signal triggers the DMA channels CH1 (SPI-A), CH2 (SPI-B) and CH3 (SPI-C) to transfer dummy data in order to start their respective SPI transmissions. After a short delay which is required for the SPI to power up SPISTE switches to low indicating an active transmission. Since the ADC must not receive any conversion pulses during this time, this pin is used as a trip zone (see section 3.2.5) for EPWM7, EPWM8 and EPWM9 to suppress any conversion pulses. Furthermore $\overline{\text{SPISTE}}$ is also used to open the buffer for the SPI signals.

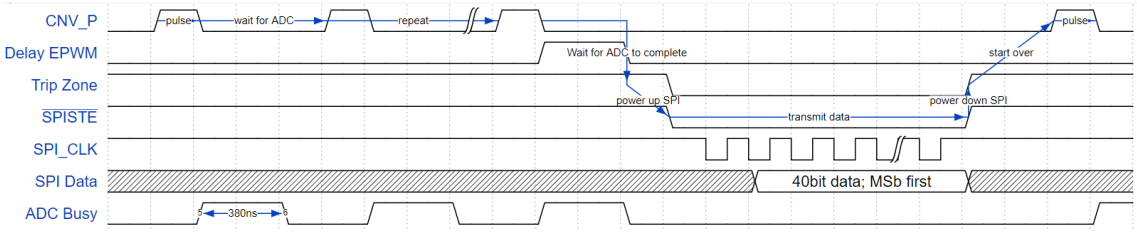

Figure 5.5.: Repetition of CNV pulses followed by an SPI transmission (not to scale)

Testing has shown that SPISTE is toggled shortly before and after the actual transmission which prevents undefined behavior caused by prematurely closing the buffer as described above. The exact duration of this delay is not documented despite being visible in their documentation [12, p. 38]. However the delay seems to be "long enough" to not cause any unwanted behavior.

The transmission itself contains 40 bits of data split into 4 words each holding 12 bits. This means that the last 8 bits do not hold any information and are discarded. Splitting the data this way simplifies data assembly and debugging. The resulting overhead is negligible. Once a word is transmitted it is automatically stored in the SPI's hardware receiver buffer (SPIRXBUF). After all transmissions has been completed the SPI is powered down again and the trip zone is lifted thus allowing further conversion pulses to trigger the ADC. This process is repeated until SPIRXBUF has reached its maximum capacity of 16 words which will trigger an interrupt event.

Only then the CPU becomes active by reading out the FIFO buffer and processing the newly received data. This design has the advantage of handling all conversion pulses and SPI communications on a hardware-level thus allowing the CPU to process the data independently. However it is important that the ISR is completed before the next interrupt is generated otherwise samples are lost due to a buffer overflow. To counteract this two options are available:

- increase the time between the data processing routine by increasing the number of conversions (CNV_NUM) or the delay between (CNV_PERIOD). Note that this affects the sample rate!

- decreasing the CPU load by simplifying the data processing routine (i.e. reducing the filter order).

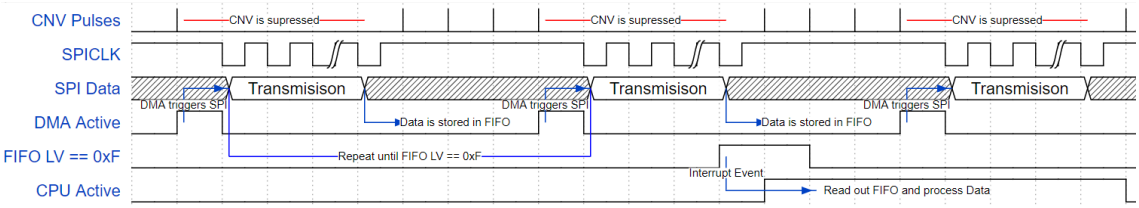

Figure 5.6.: The complete cycle of conversion, transmission and data processing. Note that data can be transmitted while the CPU is active.

As long as the CPU is busy processing the data GPIO62 (pin 18) is set to high allowing a visual representation of the CPU load using an oscilloscope. Due to the appearance of random USB transmissions it is recommended to keep the load below 80%. If a buffer overflow still occurs the first received data in the FIFO is discarded and the FIFO-overflow-flag (RXFFOVF) is set. This flag can be used to trigger a special interrupt but such a mechanism is not implemented and buffer overflows are simply ignored.

Testing has shown that it is very difficult to synchronize all three conversion pulses. This is due to hardware delays described in sections 3.2.5, 4.3.3 and 5.1.7. As a result the actual number of conversions pulses can fluctuate by ±1.

5.2.4. Data processing

Once the interrupt for data processing is generated either `void main_ISR()` or `void Record_HW_ISR()` is called. The latter routine sends a signals back to the host indicating that the data buffer has reached full capacity. However at its core both functions perform the same task by calling `uint32_t SPIBufferRead()`. This function performs several tasks:

1. Emptying all FIFO buffers
2. data assembly
3. calculating and applying a weighting factor
4. digital filtering

The first step must be completed before the next SPI transmission in order to prevent loss of data. The buffers are read simply by copying all contents into a 2D array located on RAM .

The rows of `buffer[][]` represent the channels SPI-A, SPI-B and SPI-C. The entries in each row are holding the 16 words from the respective FIFO.

Code 5.9: Copying FIFO contents to RAM

```
uint16_t buffer[ACTIVE_SPI_CNT][16];
for(i = 0; i < ACTIVE_SPI_CNT; i++)
{
   while(spi_ptr[i]->SPIFFRX.bit.RXFFST != 0) // Read until buffer empty
   {
      buffer[i][rx_cnt[i]*4] = spi_ptr[i]->SPIRXBUF; // data High Word
      buffer[i][1 + rx_cnt[i]*4] = spi_ptr[i]->SPIRXBUF; // data LW
      buffer[i][2 + rx_cnt[i]*4] = spi_ptr[i]->SPIRXBUF; // avg_cnt HW
      buffer[i][3 + rx_cnt[i]*4] = spi_ptr[i]->SPIRXBUF; // avg_cnt LW
      rx_cnt[i]++;
   }
}
```

Next the raw data is assembled for each channel as seen in Code 5.10. The first 2 words contain the actual measured value from the ADC. Since the most significant bit is transmitted first the first word has to be shifted by 12 bits to the left. The second word can simply be appended. A cast to `uint32_t` is required otherwise words are treated as if they consist of only 16 bits which would lead to the first 12 bits being discarded by the shift operation.

Bits 24-39 hold the number of conversions that have been performed by the ADC for a certain measured value. The data is assembled just as above but since it consists of only 16 bits another shift of 8 bits to the right is required. In order to properly calculate the weighting factor this number is also incremented by one.

Code 5.10: Data assembly. Note that this snippet enclosed in another loop iterating over `channel_num`

```
for(i = 0; i < rx_cnt[channel_num]; i++)
{
   // assemble measured value
   measure_val = (uint32_t) buffer[channel_num][4*i] << SPI_XMIT_LENGTH;
   measure_val |= (uint32_t) buffer[channel_num][1 + 4*i];

   // assemble number of averaged points
   avg_cnt = (uint32_t) buffer[channel_num][2 + 4*i] << SPI_XMIT_LENGTH;
   avg_cnt |= (uint32_t) buffer[channel_num][3 + 4*i];
   avg_cnt >>= 8;
   avg_cnt++;

   // sign extension if necessary
   if((measure_val >> 23) & 1)
      measure_val |= 0xFF000000;

   // apply weighting factor
```

```
18    measure_val *= (float)NextHighestPowerOfTwo(avg_cnt) / (float)avg_cnt;
19 }
```

The possible output values of the ADC range between -8388608 and 8388608. The actual SPI transmission does not differentiate between any data types or mathematical sign. This means that whenever a signed 24-bit value is stored as a signed 32-bit value all negative numbers are interpreted wrong by the processor because the assignment operator simply moves the data bit wise to its new variable.

As an example consider the 6-bit signed representation of $[-1]_{10} = [111111]_2$ stored in an 8-bit register. Since the processor has no knowledge of our interpretation the data is simply moved bit wise to its location thus changing to $[00111111]_2 = [63]_{10}$. This means that an additional step is required for negative numbers to fill up the leading 0s with 1s. This can be achieved by first checking if the 23^{rd} bit is high indicating a negative value. If true a bit-mask is added using an OR-operator to fill up the leading 0s.

In a final step a weighting factor has to be calculated as described in Section 2.2.3.

5.2.5. Programmable function generator

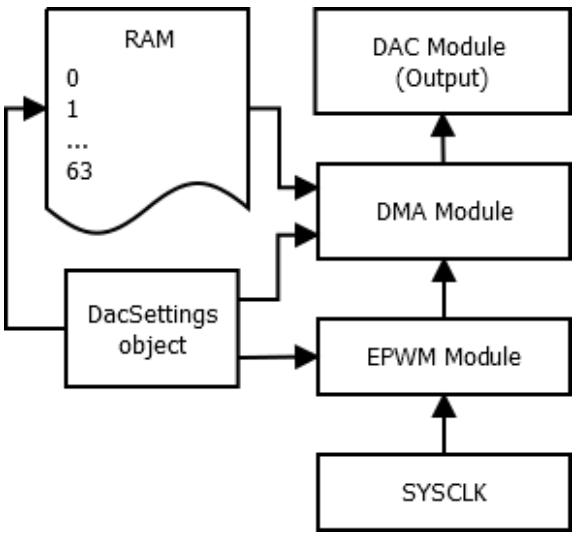

Figure 5.7.: Signal generator architecture

Another feature that has been requested later during development is the ability to create arbitrary signals on all three DAC channels in order to use the controller as a signal generator.

Figure 5.7 shows the basic architecture of the system. After reset a `DacSettings`-object is created and initialized. It contains all parameters that define the signal (type, amplitude, offset, frequency, etc) and a pointer to a RAM section that is 64

words long. Calling `DacInit()` configures the DMA and EPWM modules. Furthermore the function is generated and written to RAM.

During operation the DMA continuously reads those points from RAM and transfers them to the DAC. Every transmission is triggered using an EPWM channel. The whole system is driven by SYSCLK and therefore affected by changes of `GLOBAL_CLOCK_SCALE`.

3 identical channels are available in total . Their module assignment is as follows:

Channel A: DACA - DMACH4 - EPWMCH2

Channel B: DACB - DMACH5 - EPWMCH3

Channel C: DACC - DMACH6 - EPWMCH4

The advantage of this system is that it does not require any interaction with the CPU (except for the initialization), and therefore has no impact on performance as long as the signal is not changed.

Since the function has a well defined number of points the frequency can be adjusted by changing the EPWM cycle period. In order to change amplitude or offset of a signal the whole function has to be recalculated. It is possible to safely write new data to the RAM section during operation. If CPU and DMA access RAM[4] on the same cycle the controller automatically avoids conflicts by stalling the CPU until the DMA transfer is finished. [5, Section 4.4 CPU Arbitration]

Code 5.11: Example of initializing a DAC channel

```
1  // location for the DAC data points
2  // must be defined globally for DMA to work
3  #pragma DATA_SECTION(dynamicDacFunc_data, "dacFunc_data_sec");
4  uint16_t dynamicDacFunc_data[64];
5
6  void InitDacFunction()
7  {
8     PeriodicFunc function; // Math. function object
9     DacSettings settigns; // DAC settings object
10    settings.dacFunc = &function;
11
12    // Generate function and save values to RAM
13    GenerateFunction(
14       settings.dacFunc, /*pointer to function object*/
15       SINE, /*function type*/
16       9000, /*frequency [Hz]*/
```

[4]Or any other two systems that is located on the same peripheral frame

```
17      2000, /*amplitude*/
18      500, /*offset*/
19      dynamicDacFunc_data, /*Pointer to first value*/
20      64 /*Number of points period*/
21    );
22
23    // set up peripherals
24    InitDacDma(g_dacSettings);
25    InitDacEpwm(g_dacSettings);
26  }
```

The EPWM module is limited in its maximum period according to Formula 3.2 on page 20. This results in a minimum frequency of 23.4Hz for 64 bit functions. If lower frequencies are required the following options are available:

- Increase the the number of points per function. Note that this requires adjustments in the Linker Command File to account for the additional memory.
- Change `EPwmXRegs.ETPS.bit.SOCAPRD` which tells to the EPWM module to skip up to 2 events before triggering the a DMA transmission. This increases the maximum period by a factor of up to 3.
- Change EPWMCLKDIV or SYSCLK. The resulting minimum frequency per increment is calculated using Equation 3.2 on page 20. This is not advised since EPWMCLKDIV and SYSCLK are linked to most parts of the system and changes can have a severe impact.

By default SOCAPRD is set to ignore every second event resulting in a minimum frequency of 11.7 Hz for each DAC channel.

The maximum signal frequency depends on multiple factors such as: function type, number of points per function, maximum signal slope, SOCAPRD, DMA utilization, etc. For a sine function similar to Code 5.11 the function generator is capable of reliably producing output signals of at least 100 kHz.

5.3. Detailed Module Description

The following section describes all modules and routines that are required for regular operation. Furthermore, all modified register settings are summarized in a tabular format including their name, value and a short description. A comprehensive list of all registers and their default values after reset can be found in the Technical Reference Manual.

5.3.1. Internal System

The internal system refers to all components that define the behavior of the CPU and memory. It consists of 3 main components:

- System: Contains functions for initialization and controlling the program flow.
- Interrupts: Routines that are linked to peripheral interrupt signals and executed when triggered.
- Data buffers: circular buffers for samples, communications, DAC functions, etc

The file main.c contains the program entry point. The header file Project.h contains a list of commonly used header files, macros and project wide settings.

Table 5.1.: General program settings defined in Project.h. All entries except global variables are defined as macros and replaced by the compiler when the project is built.

Name	Value	Description
System and data buffers		
GLOBAL_CLOCK_SCALE	1	A global scaling factor for the system clock. 1 = CPU clock rate is 200MHz 2 = CPU clock rate is 100MHz ... 127 = CPU clock rate is 1.56MHz This setting has no effect on the USB interface which is driven by a seperate clock.
USB_BULK_BUFFER_SIZE	1024	Number of 16 bit words reserved for USB communication. Must be at least twice as big as USB_8BIT_BUFFER_SIZE
USB_8BIT_BUFFER_SIZE	512	Maximum number of bytes per USB transmission
SAMPLE_BUFFER_LENGTH	4095	Maximum number of 32bit data samples that can be stored per channel. If the controller is operating in single channel mode this number can be multiplied by 3.
ADC control		
SPI_FAST_BAUD_RATE	1	Baud rate for SPI transmissions while the LTC2380-24 ADC is selected 1 = SCK is 50MHz 2 = SCK is 25 MHz ... 127 = SCK is 393.7 kHz

Name	Value	Description
SPI_SLOW_BAUD_RATE	19	Baud rate for SPI transmissions while the LTC2400 ADC is selected 1 = SCK is 50MHz 2 = SCK is 25 MHz ... 19 = SCK is 5MHz ... 127 = SCK is 393.7 kHz
SPI_XMIT_LENGTH	12	Number of bits per word for SPI transmissions
ACTIVE_SPI_CNT	3	Number of active SPI channels 0 = Disable all channels 1 = SPI-A is active 2 = SPI-A,B is active 3 = SPI-A,B,C is active
CNV_PERIOD	50	Time between 2 conversions 50 = 500ns 51 = 510ns ... This value must always be >40 (400ns)
CNV_NUM	20	Number of conversions that are averaged by the LTC2380-24 ADC. The actual number of conversions will fluctuate by ±1
CNV_MULT	1	Multiplier for CNV_PERIOD. 1 = Number of conversions is CNV_PERIOD 2 = Number of conversions is CNV_PERIOD*2 ... The EPWM module limits the maximum period of signals to 65535 TBCTL cycles. CNV_MULT can be used to extend this limit by satisfying the following equation: $(CNV\_PERIOD * CNV\_NUM) + CNV\_NUM - 1 < 65535 * CNV\_MULT$
Temperature sensor ADC control		
TEMP_ADC_VREF	3.3	Reference voltage for the temperature ADC in Volts
TEMP_ADC_K	9.313 E-9	Linear response of the LTC2400 ADC
MV_PER_KELVIN	0.01	Temperature sensor sensitivity in Volts/Kelvin
I²C		
I2C_GPIO_READ_ADDR	0x71	I²C Read address for the external GPIO extender

Name	Value	Description
I2C_GPIO_WRITE_ADDR	0x72	I²C Write address for the external GPIO extender
Global variables		
uint16_t g_dataSource	1	Measurement data source 1 = Use data from experiment 2 = Generate data using a predefined formula 3 = Take predefined data samples from "testDataIn.txt"
bool g_bStoreData	true	true = store data in RAM false = discard transferred samples and skip processing

Files:

Header: core_system.h, isr.h, data_buffers.h, Project.h

Source: core_system.c, isr.c, data_buffers.c, main.c

Object definitions:

Code 5.12: Circular Buffer definition

```
typedef struct circular_buffer {
   size_t capacity;  // maximum number of samples
   size_t count;     // holds number of 32bit samples
   int32_t *buffer;  // points to the beginning of the data-array
   int32_t *buffer_end; // end of data buffer
   int32_t *head;   // latest value in buffer
   int32_t *tail;   // points to the physical end
   int32_t *usb_tail;  // first value for USB transmission
} circular_buffer;
```

Important functions:

void Initialize() This function must be executed from RAM. It initializes all systems and modules to their default state. This function has to be called right after the program entry point. After completion `void SystemStart()` may be called to start the system.

void SystemStart() Enables all measurements by starting DMA, EPWM and SPI;

void SystemPause() Stops all measurements. DMA, EPWM and SPI. This does not disconnect the modules from their peripheral bridge. Values can still be written and read from their respective registers. USB and SCI modules are unaffected.

void RxSpi_ISR_ack() This function has to be called at the end of every interrupt service routine that gets initiated by the SPI-transmissions otherwise further interrupts are not getting dispatched. Note: This does not clear any flags for other interrupt sources.

__interrupt void main_ISR(void) This is the default ISR that gets called after the SPI buffer is full. Any code that concerns digital signal processing should be put between the the annotated lines shown in Code 5.13. GPIO62 (CPU_BUSY) is high while this ISR is active. `SPIBufferRead()` has to be called even if data is not being stored. (`g_bStoreData = false`) in order to empty the SPI's FIFO-buffers and prevent overflow errors.

Code 5.13: The main interrupt service routine

```
1  __interrupt void main_ISR(void)
2  {
3     GPIO_WritePin(CPU_BUSY, 1);
4     // Put code for data processing below this line
5     //-----------------------------------------------------------
6
7     // SPIBufferRead must be called every time to empty SPI FIFO
8     // even when generating debug data or g_bStoreData == false
9     SPIBufferRead(g_sample_buffer, g_bStoreData);
10
11    //-----------------------------------------------------------
12    // Data processing end
13    GPIO_WritePin(CPU_BUSY, 0);
14    RxSpi_ISR_Ack(); // Acknowledge ISR
15 }
```

__interrupt void Record_HW_ISR(void) This function is useful for debugging purposes. It records data until the sample buffer is full. At this point the routine detaches itself form the PIE vector table and restores `void main_ISR()` as the standard ISR. Since this function is usually called via the USB interface a `COMMAND_RECORD_HW` is written back to the USB buffer to signal successful completion to the host.[5] If no host is available (i.e. USB not connected) `void fflush()` should be called to empty the buffer. GPIO62 (CPU_BUSY) is high while this ISR is active.

void InitBuffers() Initializes the USB buffers and one sample buffer for every active SPI channel. If the device is operating in single channel mode only one buffer is initialized.

void cb_init(circular_buffer *cb, size_t size, int32_t data_section) Initializes `cb` as a new circular buffer object. `size` defines the maximum

[5]For more information on USB communications refer to [11]

number of elements that can be stored in the buffer. The size of each element is set to 32 bits. `data_section` is a pointer to the physical address of the buffer head. It is advised to define data sections that are bigger than 512 Bytes in the linker command file as described in Section 5.1.5.

void cb_flush(circular_buffer *cb) Resets the circular buffer object `cb` by resetting its pointers and setting the size to 0. The actual data is unaffected until it is overwritten (i.e. adding new data with `cb_push_back();`

void cb_push_back(circular_buffer *cb, int32_t *item) Adds `item` to the circular buffer object `cb`. If the buffer is already at full capacity the oldest value is discarded.

void cb_push_back_arr(circular_buffer *cb, int32_t *arr, int length) A convenience function that acts like `cb_push_back()` buts adds the array `arr` with length `length` to the circular buffer object `cb`.

uint32_t cb_uploadToUSB(circular_buffer *cb) Uploads all entries in `cb` to the USB interface and flush it after completion.

5.3.2. Peripheral Systems

5.3.2.1. DAC

The DAC module consists of two parts. The DAC peripheral module as described in Section 5.3.2.1 and the function generator. The latter is a software feature that holds information about a mathematical function (amplitude, frequency, offset, etc) and generates a fixed amount of data points. The data is written to the DAC in regular intervals using DMA.

Files:

Header: DAC.h, function_generator.h

Source: DAC.c, function_generator.c

Object definitions:

Code 5.14: Periodic function object definition

```
typedef struct sPeriodicFunc_t {
   uint16_t type;  // Sine, Rect, user deifned, etc..
   float frequency;
   uint16_t amplitude;
   uint16_t offset;
   uint32_t updateRate; // number of EPWM ticks between each update
   uint16_t * data; // Pointer to data array
   uint16_t size; // number of points for 1 Period
} PeriodicFunc;
```

Important functions:

void DacInit() Requires DMA and EPWM to be initialized. Configures all three DAC channels for basic use by applying settings as defined in Table 5.2. `DacInit()` also defines the initial output signals that are generated.

Table 5.2.: Settings for the DAC module

Bit	Name	Value	Description
CpuSysRegs.PCLKCR16 - Peripheral Clock Gating Register 16			
all	DAC_C	0xFFFF	Buffered DAC Clock Enable Connect the clock to all DAC modules. Otherwise no values can be written to it.
DACCTL - DAC Control Register			
0	DACREFSEL	0x1	DAC reference select Use VREFHI and VREFLO as reference voltages
DACOUTEN - DAC Output Enable Register			
0	DACOUTEN	0x1	Enable DAC Output
DACVALS - DAC Value Register (shadow)			
11-0	DACVALS	0x0	Shadow output code to be loaded into DACVALA

void DacSetAbs(int dac_num, uint16_t val) Sets `dac_num`[6] to `val`. If `val > 0xFFF` the device is halted.

void GenerateFunctionPoints(PeriodicFunc *hFunc) Generates data that represent the function which is output on the DAC according to the properties defined in the handle.

void InitDacDma(PeriodicFunc **hFunc) Sets up DMA for transmissions between RAM and DAC. The argument must be an array of `PeriodicFunc` of length 3.

void InitDacEpwm(PeriodicFunc **hFunc) Sets up EPWM for triggering DMA. The argument must be an array of `PeriodicFunc` of length 3.

5.3.2.2. DMA

Files:

Header: DMA.h

[6]GPIO.h defines the macros DACA (1), DACB (2) and DACC (3) which can be passed to this function.

Source: DMA.c

Important functions:

void DMAInit() Resets and initializes all three DMA modules according to tables 3.2 and 5.3. Source and destination addresses are configured using `DMACHxAddrConfig()` which is provided by the TI support library.

Table 5.3.: DMA Settings performed by `DMAInit()`

Bit	Name	Value	Description
DMA Control, Mode and Status Registers			
0	HARDRESET	0x1	This bit is set right after `DMAInit()` is called performing a hard reset by setting all bits to their default state
15	DEBUGCTRL	0x1	Emulation Control Bit HALT points do not affect transmissions. This can prevent errors that occur while debugging by completing a transmission even when a break point is hit
CPU_SYS_REGS - CPU System Registers			
3-2	PF2SEL	0x1	Use the secondary peripheral bus for DMA. DMA transmission no longer affect the C28x main bus. This limits performance of the CLA
1-0	PF1SEL	0x1	same as PF2SEL
DMA Channel 1-6 Registers			
all	BURSTSIZE	*	Number of bursts + 1 occurring during each transfer. CH1-3: BURSTSIZE = 3 CH4-6: BURSTSIZE = 0
all	SRC_BURST_STEP	*	Source memory address increment between after each burst. CH1-3: SRC_BURST_STEP = 1 CH4-6: SRC_BURST_STEP = 0
all	SRC_TRANSFER_STEP	*	Source memory address increment between after each transfer CH1-3: SRC_TRANSFER_STEP = 0 CH4-6: SRC_TRANSFER_STEP = 1
all	DST_TRANSFER_STEP	0x0	Destination memory address increment between after each transfer The memory addresses of SPITXBUF and DACVALS never change. Thus, no step is required.

Bit	Name	Value	Description
all	TRANSFER_ SIZE	*	Number of transfers + 1 until source and destination addresses are reset CH1-3: TRANSFER_SIZE = 0 CH4-6: TRANSFER_SIZE = number of data points - 1 used to generate the DAC output
DmaClaSrcSelRegs.DMACHSRCSEL1 - Trigger source select register 1			
31-24	CH4	0x26	EPWM2A
23-16	CH3	0x36	EPWM10A
15-8	CH2	0x36	EPWM10A
7-0	CH1	0x36	EPWM10A
DmaClaSrcSelRegs.DMACHSRCSEL2 - Trigger source select register 2			
15-8	CH6	0x28	EPWM4A
7-0	CH5	0x2A	EPWM3A

void DMAStart() Start or continue all DMA transfers.

void DMAStop() Stop all DMA transfers. Modules remain connected to their peripheral bridge, therefore respective registers can still be accessed.

5.3.2.3. GPIO

Files:

Header: GPIO.h, f2837xs_pinmux[7]

Source: GPIO.c, f2837xs_pinmux.c[7]

Important Functions:

void GPIO_SetPinMux() Load configuration from f2837xs_pinmux.h, apply them to all GPIOs and set them to LOW.

void GPIO_SetPin(int gpio, int state) Set `gpio` as a digital output pin. If `state = GPIO_HIGH` set it to high otherwise set it to low. Warning: This overrides any previous signal configurations for a GPIO!

[7]Generated with PinMux Tool. For more information refer to section 3.2.1.1.

Code 5.15: Example of defining the state of a GPIO

```
// set this Pin to high
GPIO_SetPin(DEBUG_PIN, GPIO_HIGH);

// set this pin to low.
// Warning: EPWM configuration will be lost
GPIO_SetPin(EPWM7A, GPIO_LOW);
```

void GPIO_SetToDefaultState() Set all GPIOs of type O from Table 3.1 to their default operation state defined in Table 5.4. Warning: This overrides any previous signal configurations for those GPIOs!

void GPIO_SetToTemperatureReading() Set all GPIOs from Table 5.4 to the state defined below. To select the LTC2400 on all three SPI channels.

Table 5.4.: GPIO state for default and temperature reading operation

Name	GPIO	Default	Temp. Read
BUFF_OE_A	90	HIGH	LOW
BUFF_OE_B	86	HIGH	LOW
BUFF_OE_C	4	HIGH	LOW
CS_A	89	HIGH	LOW
CS_B	87	HIGH	LOW
CS_C	21	HIGH	LOW
ADC_IO_A	61	LOW	HIGH
ADC_IO_A	66	LOW	HIGH
ADC_IO_A	72	LOW	HIGH
DEBUG_PIN	10	LOW	LOW
CPU_BUSY	62	LOW	LOW

5.3.2.4. EPWM

Files:

Header: EPWM.h

Source: EPWM.c

Object Definitions:

Code 5.16: `EpwmSettings` holds information about the EPWMs that control the behaviour of the LTC2380-24 ADC. It is also used to communicate settings between the controller and the USB interface.

```
typedef struct EpwmSettings {
   uint16_t cnv_period;
   uint16_t cnv_num;
   uint16_t cnv_mult;
   bool bXmitPeriodOverflowFlag;
} EpwmSettings;
```

Important functions:

void EpwmInit() Initializes all 12 EPWM channels.

void EpwnCnvReset() It is not possible to start multiple EPWMs simultaneously because a synchronization pulse needs to propagate through the EPWM tree (see TRM Figure 13-7). This creates a different delay on every output. In order to better align samples among all three channels `EpwnCnvReset()` can be used to partially compensate for this by resetting the internal counter and adding an offset depending on the EPWM channel.

void EpwmStart() Starts all EPWM channels by connecting the peripheral clock (PERCLK) to the time-base counter input and calling `EpwnCnvReset()`.

void EpwmStop() Stops all EPWM modules.

void EpwmInit() Initializes all 12 EPWM modules. This process is performed using multiple steps. First a base configuration for all modules is applied. Next the settings for the channels responsible for CNV-pulses and delay triggers are applied. In a last step functions for EPWM1-12 are called for settings that are unique to a certain EPWM. Note that after completion all counters are still halted and need to be started using `EpwmStart()`.

void EpwmBase() Loads a base configuration for EPWM1-12

Table 5.5.: Base Configuration for every EPWM

Bit	Name	Value	Description
TBCTL - Time Base Counter Control Register			
5-4	SYNCOSEL	0x0	Sync output Select Defines when a sync-event is forwarded to the next EPWM. 0x0: follow previous EPWM. Note this happens at a delay. Refer to Section 3.2.5 for more information.

Bit	Name	Value	Description
2	PHSEN	0x1	Setting this bit allows the counter to be synchronized by a sync-event.
1-0	CTRMODE	0x0	Counter Mode Defines how counting is performed by the EPWM (up, down, up-down, no counting). All channels are set to up-count mode.
TBCTL2 - Time Base Counter Control Register 2			
7	OSHTSYNC	0x1	Oneshot sync bit OSHTSYN = 1 allows sync pulse to propagate
6	OSHTSYNCMODE	0x1	Enable oneshot sync mode
TBCTR - Time Base Counter Register			
all	TBCTR	0x0	This register holds the current counter value for the time based counting mode. For the base configuration synchronization delays are disregarded and all counters are set to 0.

void InitEpwmCnv(); Configures EPWM7-9 to generate pulses that trigger the ADC conversion.

Table 5.6.: Configuration for EPWM7-9 (CNV)

Bit	Name	Value	Description
TBPRD - Time Base Period Register			
all	TBPRD	0x32	Defines ADC's conversion rate. The time between conversions is $\Delta t = TBPRD \cdot 10ns$ This value can be changed at run time.
CMPA - Counter Compare A Register			
31-16	CMPA	0x5	Compare A Register Defines the falling edge of the CNV_P pulse. Note that the minimum pulse width defined in the LTC2380-24 data sheet is not sufficient because of a delay created by the buffer. This value was determined by experiment.
CMPB - Counter Compare B Register			
31-16	CMPB	0xF	Compare B Register Defines the falling edge of the CNV_N
AQCTLA - Action Qualifier Control Register for Output A			

Bit	Name	Value	Description
9-8	CBU	0x1	Action when TBCTR = CMPB 0x1: force output low
5-4	CAU	0x1	Action when TBCTR = CMPA 0x1: force output low
1-0	ZRO	0x2	Action when TBPRD = 0 0x2: force output high
AQCTLB - Action Qualifier Control Register for Output B			
9-8	CBU	0x2	Action when TBCTR = CMPB 0x2: force output high
5-4	CAU	0x1	Action when TBCTR = CMPA 0x1: force output low
1-0	ZRO	0x1	Action when TBPRD = 0 0x1: force output low
TZCTL - Trip Zone Control Register			
3-2	TZB	0x1	Override output B to high when trip zone applies
1-2	TZA	0x2	Override output A to low when trip zone applies

void InitEpwmDelay(); Configures EPWM10-12 to generate an additional delay before a transmission is initiated. The delay is required so the ADC can finish its final conversion.

Table 5.7.: Configuration for EPWM10-12 (Delay)

Bit	Name	Value	Description
TBPRD - Time Base Period Register			
all	TBPRD	*	This value is calculated using `uint16t xmit_period()` correlates to the sample rate.
CMPA - Counter Compare A Register			
31-16	CMPA	*	Compare A Register Defines the start of the delay. Depends on the settings for the conversion pulses.
CMPB - Counter Compare B Register			
31-16	CMPB	*	Compare B Register Defines the end of the delay. Depends on SPI settings.
AQCTLA - Action Qualifier Control Register for Output A			
9-8	CBU	0x1	Action when TBCTR = CMPB 0x1: force output low

Bit	Name	Value	Description
5-4	CAU	0x2	Action when TBCTR = CMPA 0x2: force output high
ETSEL - Event Trigger Selection Register			
11	SOCAEN	0x1	Enable EPWMxSOCA (start of converison A) Must be enabled for the DMA to trigger
10-8	SOCASEL	0x1	EPWMx start of conversion A select options Determines which at which point of the EPWM cycle the SOCA pulse is generated 0x1: Generate pulse when TBCTR = 0
ETPS - Event trigger prescale register			
9-8	SOCAPRD	0x0	SOCA Period select By design every EPWM is limited to a maximum period (see eq. on page 20). If $TBPRD > T_{max}$ SOCAPRD can be used to extend the maximum period by a factor of up to 3 by ignoring the first SOCAPRD events. 0x0: ignore no events 0x1: ignore every second event ... 0x3: ignore the first 3 events

void InitDacEpwm(const DacSettings*); Configures EPWM2-4 for DAC signal generation. CMPA is used to trigger DMACH4-6 to transfer 1 word to the DAC.

Table 5.8.: Configuration for EPWM2-4 (DAC)

Bit	Name	Value	Description
TBPRD - Time Base Period Register			
all	TBPRD	*	Can be used to control the frequency of the output signal.
CMPA - Counter Compare A Register			
31-16	CMPA	0x1	Compare A Register This value has no effect on the operational characteristics of the controller. CMPA is used to generate a short pulse on the EPWM-pin for debugging purposes. The rising edge is indicating the start of the next DMA transmission.

Bit	Name	Value	Description
AQCTLA - Action Qualifier Control Register for Output A			
5-4	CAU	0x1	Action when TBCTR = CMPA 0x2: force output low
1-0	ZRO	0x2	Action when TBPRD = 0 0x1: force output high
ETSEL - Event Trigger Selection Register			
11	SOCAEN	0x1	Enable EPWMxSOCA (start of converison A) Must be enabled for the DMA to trigger.
10-8	SOCASEL	0x1	EPWMx Start of Conversion A select options Determines which at which point of the EPWM cycle the SOCA pulse is generated. 0x1: Generate pulse when TBCTR = 0
ETPS - Event trigger prescale register			
9-8	SOCAPRD	0x2	Start of Conversion A Period select 0x2: ignore every second trigger event. This is used to enable longer signal periods. For more information refer to Section 5.2.5.

void InitEpwmUnique(); Loads all settings that are unique to a certain EPWM. The actual code is placed in `void InitEpwmx()`(x being the respective EPWM channel number. `void InitEpwmUnique()` is executes all 12 functions in numerical order.

Table 5.9.: Unique configuration for EPWM1-12

Bit	Name	Value	Description
EPWM1			
TBCTL - Time Base Counter Control Register			
5-4	SYNCOSEL	0x1	Sync output Select Generate a sync event for the following EPWMs making EPWM the sync master. 0x1: generate event when TBCTR = 0
EPWM7			
TZSEL - Trip Zone Select Register			
1	CBC2	0x1	0x1: Enable TZ2 for this EPWM
0	CBC1	0x1	0x1: Enable TZ1 for this EPWM
InputXbarRegs - Input X-Bar Selection register[8]			

[8]This register is not a part of the EPWMx register bank. For more informarion see Technical Reference Manual Section 7.3.2

Bit	Name	Value	Description
all	INPUT1SELECT	0x48	Select SPISTE_C as a source for Trip Zone 1
all	INPUT2SELECT	0x12	Select EPWM10A as a source for Trip Zone 2
EPWM8			
TZSEL - Trip Zone Select Register			
1	CBC2	0x1	0x1: Enable TZ2 for this EPWM
0	CBC1	0x1	0x1: Enable TZ1 for this EPWM
EPWM9			
TZSEL - Trip Zone Select Register			
1	CBC2	0x1	0x1: Enable TZ2 for this EPWM
0	CBC1	0x1	0x1: Enable TZ1 for this EPWM

5.3.2.5. SCI

The SCI system is no longer used since it is replaced by the USB interface. It can however act as a simple terminal for debugging purposes.

Files:

Header: SCI.h

Source: SCI.c

Important Functions:

void SciInit() Initializes SCI-A for basic operation. Furthermore its respective GPIOs and FIFO buffers for transmitted, and received data are configured. The baud rate is set to 9600 bits/s

void SciXmit(char a) Transmits a single byte of data. If more than one word is in the FIFO buffer the function waits until the previous transmission is completed.

void SciMsg(char *msg) A convenience function used to transmit a 0-terminated string of data

6. Conclusion and Outlook

The main goal of creating 3 parallel Serial Peripheral Interfaces (SPI) for the LTC2380-24 ADC using the F28377S LaunchPad Development Kit has been achieved. 24 bit data is acquired at the ADC's maximum rate of 2 Megasamples per second, averaged and transmitted at 50 Mb/s.

Furthermore, the controller's 12-bit DAC channels are used to operate as programmable function generators. Output signals can range between 0 and 3.3V at frequencies of up to100kHz.

Both systems incorporate hardware-based features such as EPWM and DMA in order to keep the CPU load to a minimum and allow for as many cycles as possible to be used for digital signal processing.

One of the main limiting factors of the controller is the available memory. In total 164kB of volatile RAM and 1MB of non-volatile flash memory are available. Although most software parts are stored on flash, RAM has to be allocated for USB buffers, time critical functions, filtering coefficients, filter buffers, stack, heap, etc. This leaves ~50kB for measurement data or 4096 32-bit samples per channel.

There are 3 options to increase available memory:

Code Optimization: buffer sizes for USB transmission and filtering have been determined by trial and error. It is possible to scale those down at the cost of longer transmission times or decreased filter performance. Furthermore, functions stored on RAM for faster execution can be moved to flash. The Linker Command File includes several intentional gaps which allow for easier adjustment during development. Once the required memory has been optimized this file can be updated to free up unused RAM.

Flash Interface: Once a program is uploaded to the controller, the CPU is only able to read data from flash memory. Writing is not supported by default but TI does provide documents for developing an interface capable of writing to flash. During development attempts were made to implement such an interface but have been discarded due to the complexity of the problem.

Hardware upgrade: It is possible to stack three LaunchPads on top of each other and dedicate one controller to each channel thus tripling the available memory. This can be performed without making any changes to the hardware. The biggest downside is that each controller requires it's own program since

pin assignments will differ for every chip.
Alternatively the controller can be replaced by the F2837xD LaunchPad which features ~25% more RAM and a dual-core processor. The dual-core architecture also provides an increase in computational performance, since communications and signal processing can be split between the two processors.

The next document, *Development of a Universal Serial Bus Interface for a Microcontroller,* will cover the remaining parts of the project, which include:

- Digital signal processing by removing noise and analyzing the results. This is performed by using two finite impulse response filters and linear regression.
- USB interface for transmitting data to a host PC and modify measurement settings.
- Graphical user interface built on the Qt framework for displaying measurement data and configuring settings.

A. Appendix

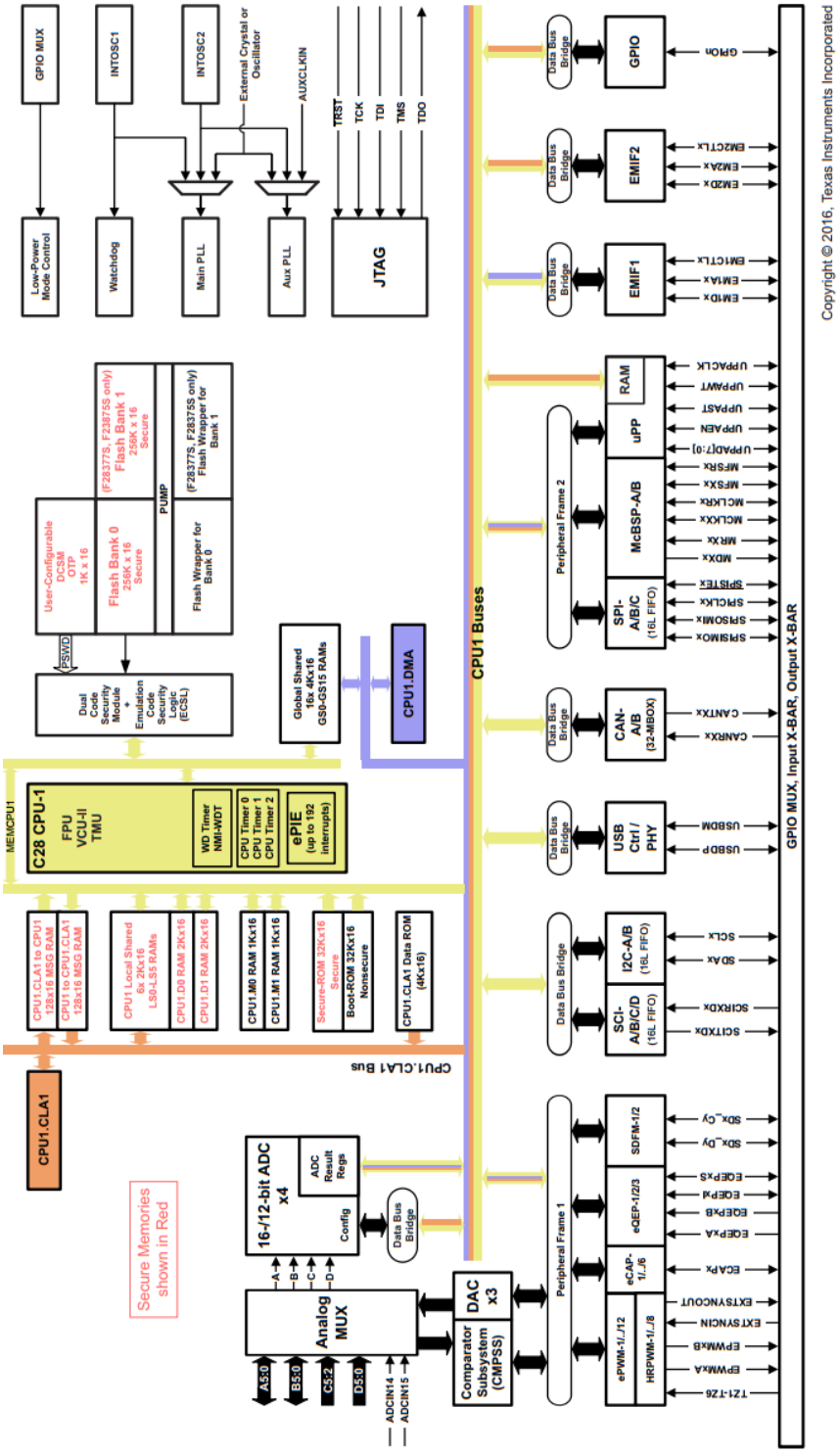

Figure A.1.: Functional Block Diagram of the TMSF28377S Controller. [6, p.4]

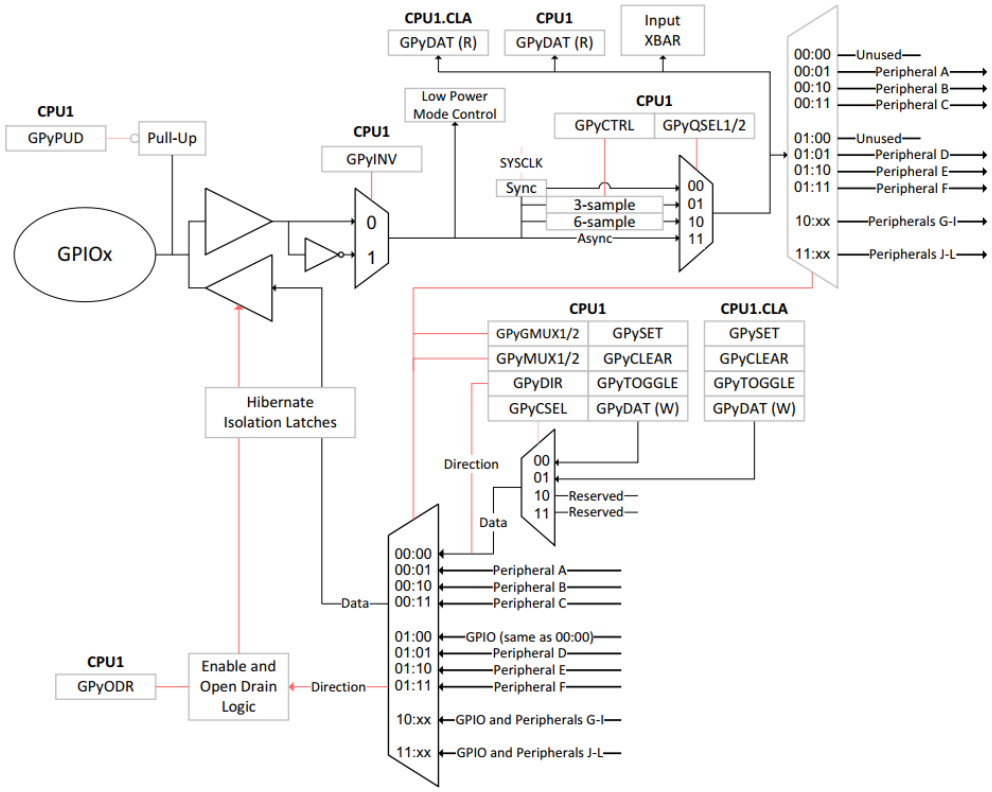

Figure A.2.: GPIO Logic for one pin [5, Figure 6-1]

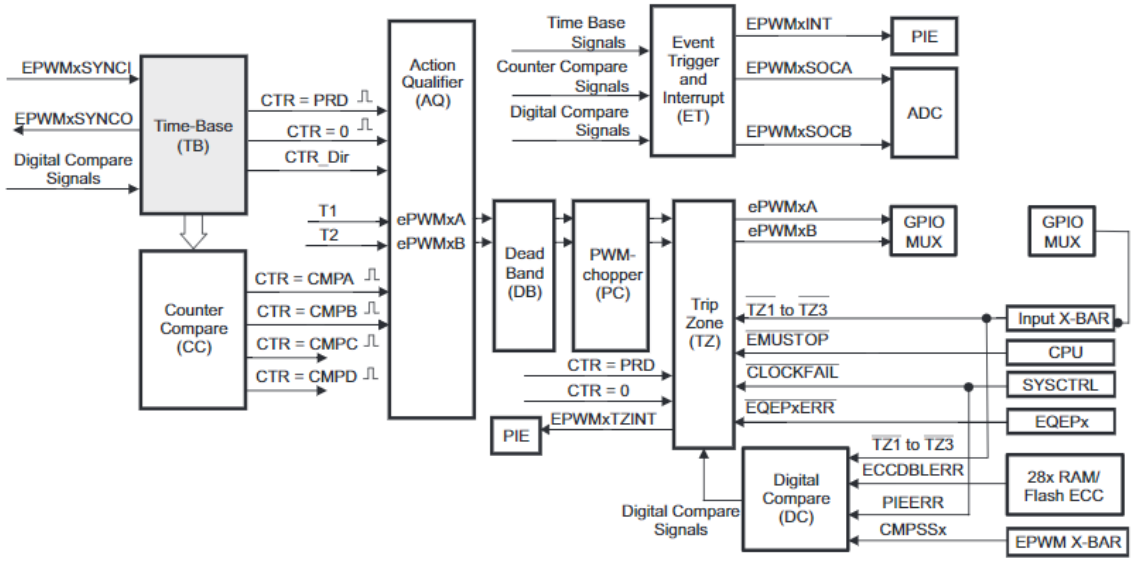

Figure A.3.: EPWM Time-Base Submodule architecture [5, Figure 13-4]

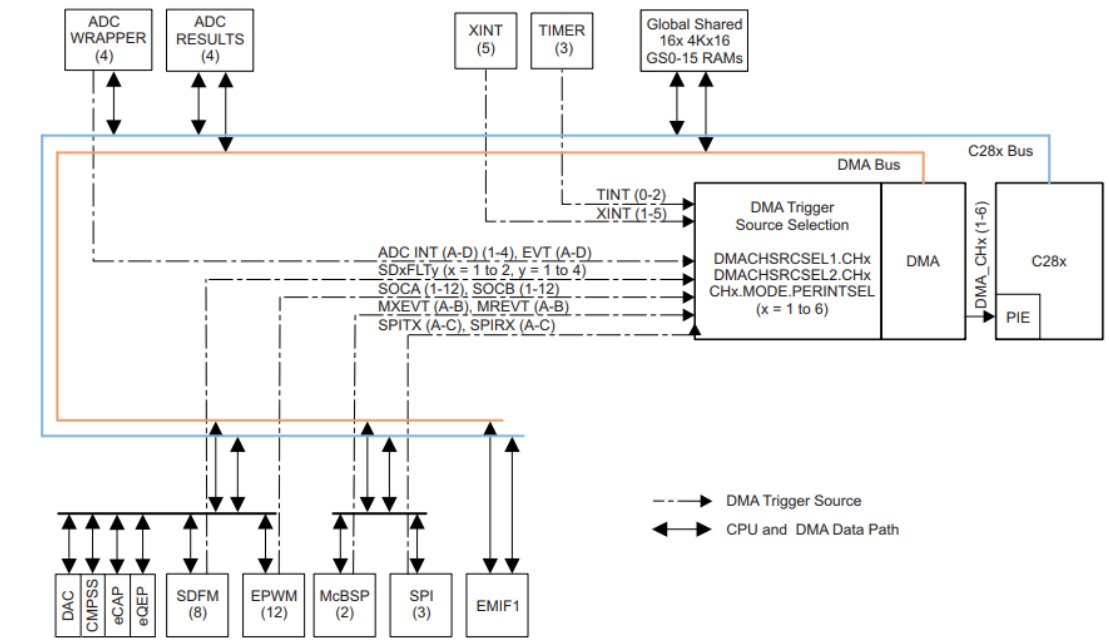

Figure A.4.: DMA architecture [5, Figure 4-1]

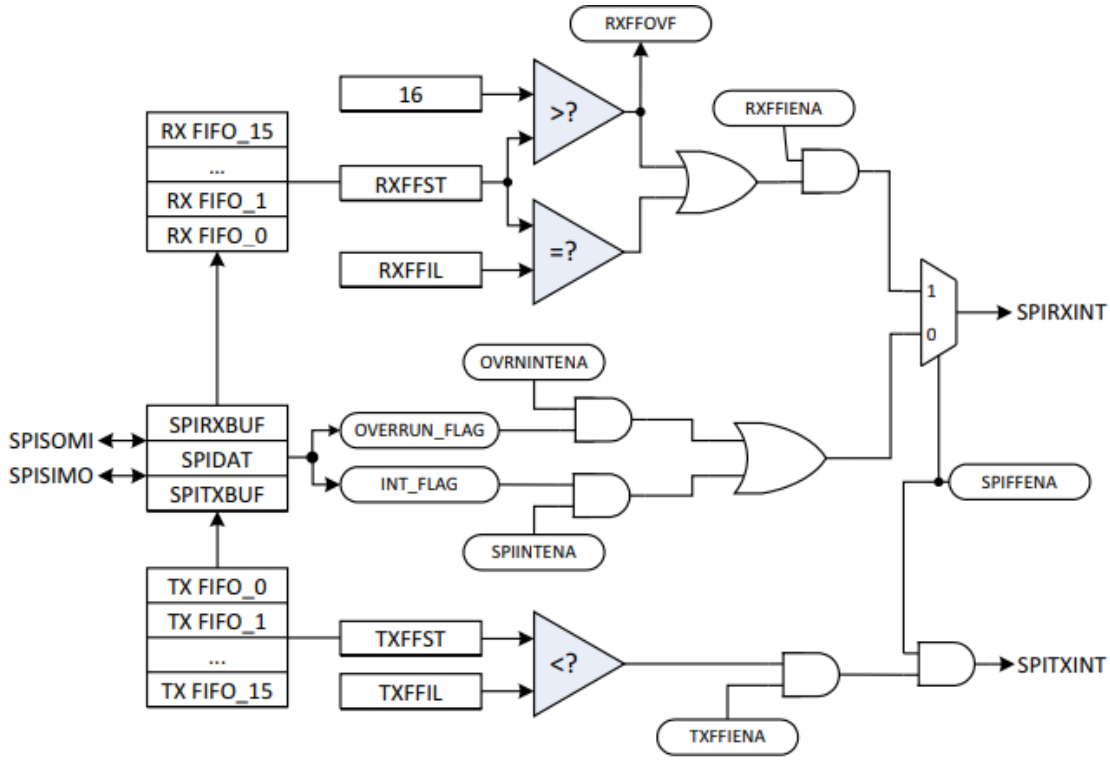

Figure A.5.: SPI interrupt generation logic [5, Figure 17-2]

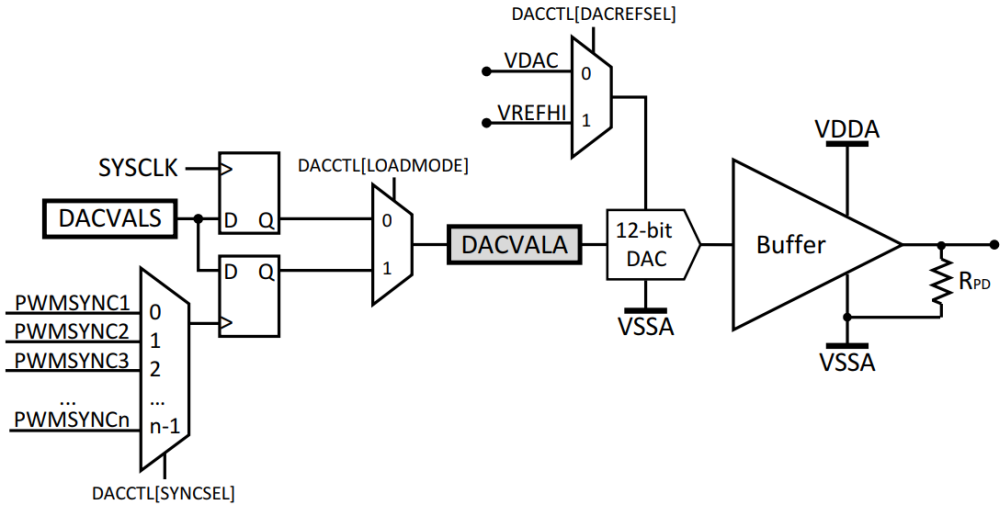

Figure A.6.: Buffered DAC architecture [5, Figure 10-1]

Acronyms

Table A.1.: Acronyms

Acronym	Definition
ADC	Analog to Digital Converter
CLA	Control Law Accelerator
CLK	Clock
CPHA	Clock Phase
CPOL	Clock Polarity
CPU	Central Processing Unit
CSS	Code Composer Studio
DAC	Digital to Analog Converter
DMA	Direct Memory Access
DSP	Digital Signal Processor
FIR	Finite Impulse Response
GPIO	General Purpose Input Output
GUI	Graphical User Interface
I2C	Inter Integrated Circuit
IDE	Integrated Development Environment
ISR	Interrupt Service Routine
LSPCLK	Low Speed Peripheral Clock
MISO	Master In Slave Out
MOSI	Master Out Slave In
MSb	Most Significant Bit
MSB	Most Significant Byte

Acronym	Definition
MSPS	Megasamples Per Second
PIE	Peripheral Interrupt Expansion
PRD	Period
RAM	Random Access Memory
SCI	Serial Communication Interface
SDI	Serial Data In
SDO	Serial Data Out
SOC	Start Of Conversion
SPI	Serial Peripheral Interface
SPIRXBUF	SPI Receive Buffer
SPITXBUF	SPI Transmit Buffer
$\overline{\text{SPISTE}}$	SPI Slave Transmit Enable
SS	Slave Select
SYSCLK	System Clock
TRM	Technical Reference Manual
UART	Universal Asynchronous Receiver Transmitter
UI	User Interface
USB	Universal Serial Bus

Bibliography

[1] David Bader. Process Control for a Large Area High Resolution Hall Scanner. Master's thesis, TU Wien, 2017.

[2] Linear Technology. *LTC2380-24 Datasheet*.

[3] Linear Technology. *LTC2400 Datasheet*.

[4] Texas Instruments. *LAUNCHXL-F28377S Overview User's Guide*, April 2017.

[5] Texas Instruments. *TMS320F2837xS Delfino Microcontrollers Technical Reference Manual (SPRUHX5C)*, September 2015.

[6] Texas Instruments. *TMS320F2837xS Datasheet*, May 2016.

[7] Motorola. *SPI Block Guide*, January 2000.

[8] Texas Instruments. Pragmas in c++. http://processors.wiki.ti.com/index.php/Pragmas_in_C%2B%2B, 2014. Accessed: 2017-10-10.

[9] Texas Instruments. C++ support in ti compilers. http://processors.wiki.ti.com/index.php/C%2B%2B_Support_in_TI_Compilers, 2014. Accessed: 2017-10-10.

[10] Texas Insturments. *TMS320C28x CPU and Instruction Set*, April 2015.

[11] Mario Hiti. *Development of a Universal Serial Bus Interface for a Microcontroller*, 2019.

[12] Texas Instruments. *TMS320x281x Serial Peripheral Interface Reference Guide (SPRU059E)*, 2009.