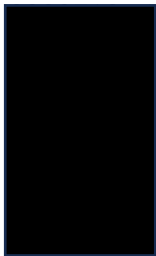









CEG 4166: Real-Time Systems Design

Winter 2023

Group #6

Name	STUDENT NUMBER
Ashair Imran	
Edward 	
Ulysses 	
Jai 	
Wesley 	
Jathushan 	

## Contents

3.2 Cyclic Scheduler Module Design .....	3
3.3 User Interface Module Design .....	6
3.5 Acquisition Module Design .....	8
3.5.1 Acquisition Module Overview .....	8
3.5.2 Acquisition Module Logic.....	9
3.8 ADC Module Design .....	11
3.7.1 ADC Hardware Module Overview .....	11
3.7.2 Sampling.....	14
3.7.3 Design.....	14
3.10 GPIO Module Design .....	20
3.10.1 GPIO Hardware Module Overview.....	20
3.10.2 GPIO Design .....	21
Implementation.....	23
GPIO Implementation.....	23

## 3.2 Cyclic Scheduler Module Design

The cyclic scheduler acts as the scheduler for three tasks, user interface, acquisition management, and analysis tasks. The cyclic scheduler needs to ensure that the tasks are executed in accordance with the schedule table, that ensures that all tasks are run to completion, and that no error occurs between the execution of the different tasks. The main concern with the cyclic scheduler is to ensure that no data overflow occurs with the double data buffer used by the data acquisition task. The data in the double data buffer is used whenever the analysis task is run, therefore, the analysis task needs to be run within a strict timeframe to ensure that no data overflow occurs.

### Analysis Task

The period of the analysis task must be below 16.67ms in accordance with the performance requirements of the task. The performance requirement is listed in the release document, which specifies that the analysis task must be able to process data from 3 ports every 16.7ms.

### User Interface Task

The execution time for the user interface task is assumed to be less than 1 ms because there is minimal computation involved. The period of execution for the user interface task will be bounded by a minimum time that allows for the data to be fully transmitted over UART to avoid flickering, and a maximum time that allows for a fast response time to user input. The minimum time is calculated by dividing the amount of data transmitted per frame by the data rate of the UART peripheral.

The amount of data per line is 45 characters for the title, 6 characters for the source subheading, 5 characters for the load and grid subheadings, 47 characters for the column headings, 61 characters for each port data, 63 characters for the user prompt, and 45 characters for the error message. Totalled up, this is 643 characters. Each character has 10 bits due to the start and stop bit of UART. Rounding up to 650 characters, the minimum period is  $(650 \times 10) \text{ bits} / 115200 \text{ bps} = 0.0564 \text{ seconds}$ . Thus, the minimum period is 56.4 ms. The maximum period is determined by the maximum amount of time that would give a good response time to user input. We will select a maximum period of 250 ms. Thus, the chosen period for this task in the cyclic scheduler must be from 56.4 ms to 250 ms. We have chosen a period of 120 ms because this matches the chosen period for the acquisition module and is within the required range.

### Check Acquisition Management Task

The check acquisition management task is to ensure that the acquisition is running on all closed ports, including when a port is opened or closed via the user interface task. Since this task is dependent on the state of the ports that can be closed or opened by the user interface task, this task will only need to be run the same number of times per second as the user interface task.

### The Schedule Table for Cyclic Scheduler Module

To design the cyclic scheduler, the first thing that is done is to determine the period and execution time of the different tasks.

The analysis task will have the shortest period, as it has an upper limit of 16.7ms between executions. By setting the period to 10ms, the period will fit well within 1 second, ideally executing 100 times within 1 second. It is known that it takes approximately 1ms to analyse 1 port worth of data, since the task needs to analyse 3 ports, then the execution time will be equal to 3ms.

The user interface task has a non-functional requirement to reduce the amount of flickering shown to the user. In conjunction with the maximum number of times the task can execute at 22 tasks per second, selecting a longer period will help reduce the flickering while adhering to the requirement. In this case, setting the period to 120ms will result in a tasks per second value of 8.33, which is well within the limit, and will result in little flickering. The execution time of the user interface task is very low, and a value from the lectures is chosen for the execution time at 0.4ms.

The check acquisition management task's period is identical to the user interface task as explained above. Therefore, it will have a period of 120ms. The execution time is negligible, taking a value from the lecture at 0.1ms.

The values of the periods are shown in Table 2, with Table 1 serving as a legend for the tasks in the tables.

$\tau_1$	Analysis Task
$\tau_2$	User Interface Task
$\tau_3$	Check Acquisition Management Task

Table 1. Task Assignment

$\tau_n$	Pi (ms)	ei (ms)	Di = Pi (ms)
$\tau_1$	10	3	10
$\tau_2$	120	0.4	120
$\tau_3$	120	0.1	120

Table 2. Task Period and Execution Time

To calculate the major cycle, the least common multiple is calculated from the periods. The results are shown in Table 3.

Major Cycle					
LCM(10,120,120) = $2 \times 2 \times 2 \times 3 \times 5 = 120$					
Period	2	2	2	3	5
10	5	5	5	5	1
120	60	30	15	5	1
120	60	30	15	5	1

Table 3. Major Cycle Calculation

The frame size is calculated in Table 4. From the table, it is shown that frame sizes, 3, 4, 5, 6, and 10 are all valid based on the periods. For this implementation, we have chosen 5ms as the frame size.

Frame Size	D1 = 10	D2 = 120	D3 = 120		
3	$2*3\text{-gcd}(10,3) = 5 \leq 10$	$2*3\text{-gcd}(120,3) = 3 \leq 120$	$2*3\text{-gcd}(120,3) = 3 \leq 120$		
4	$2*4\text{-gcd}(10,4) = 6 \leq 10$	$2*4\text{-gcd}(120,4) = 4 \leq 120$	$2*4\text{-gcd}(120,4) = 4 \leq 120$		
5	$2*5\text{-gcd}(10,5) = 5 \leq 10$	$2*5\text{-gcd}(120,5) = 5 \leq 120$	$2*5\text{-gcd}(120,5) = 5 \leq 120$		
6	$2*6\text{-gcd}(10,6) = 10 \leq 10$	$2*6\text{-gcd}(120,6) = 6 \leq 120$	$2*6\text{-gcd}(120,6) = 6 \leq 120$		
8	$2*8\text{-gcd}(10,8) = 14$	$2*8\text{-gcd}(120,8) = 18 \leq 120$	$2*8\text{-gcd}(120,8) = 18 \leq 120$		
10	$2*10\text{-gcd}(10,10) = 10 \leq 10$	$2*10\text{-gcd}(120,10) = 10 \leq 120$	$2*10\text{-gcd}(120,10) = 10 \leq 120$		
12	$2*12\text{-gcd}(10,12) = 22$	$2*12\text{-gcd}(120,12) = 112 \leq 120$	$2*12\text{-gcd}(120,12) = 112 \leq 120$		
15	$2*5\text{-gcd}(10,15) = 25$	$2*5\text{-gcd}(120,15) = 15 \leq 120$	$2*5\text{-gcd}(120,15) = 15 \leq 120$		

Table 4. Frame Size Calculation

Major Cycle: 120ms
Minor Cycle: 5ms
24 frames per major cycle

Table 5. Major and Minor Cycle Assignment

Figure 1 shows the assignment of the tasks within the major cycle, with table 6 serving as a legend for task assignment.

$\tau_1$	$\tau_2$	$\tau_1$	$\tau_3$	$\tau_1$		$\tau_1$		$\tau_1$		$\tau_1$		$\tau_1$		$\tau_1$		$\tau_1$		$\tau_1$		$\tau_1$	
----------	----------	----------	----------	----------	--	----------	--	----------	--	----------	--	----------	--	----------	--	----------	--	----------	--	----------	--

Figure 1. Major Cycle Task Assignment

$\tau_1$	Analysis Task
$\tau_2$	User Interface Task
$\tau_3$	Check Acquisition Management Task

Table 6. Major Cycle Task Assignment Legend

To implement the cyclic scheduler, one global variable, frameNum will be used to keep track of the current frame within the major cycle. frameNum increments from 1 to 24, representing the number of frames in a major cycle. The implementation will invoke the corresponding task according to the value of frameNum.

### Pseudocode

```

cyclicScheduler(UArg args)
{
    frameNum = 0;
    stringText[700]

    if(frameNum is 0){
        uart_print(-----)
    }
    if(schedule of frameNum is NULL){
        sprintf(stringText,"%d Null Task\n",frameNum);
        uart_print(stringText);
    }if(schedule of frameNum is not NULL and frameNum is not 0){
        sprintf(stringText, frameNum)
        uart_print(stringText)
        schedule of frameNum
    }
    frameNum = (frameNum+1)%majorCYCLE
}

```

*Figure 2. Cyclic Scheduler Pseudocode*

In this code whenever frameNum is zero it would not detect any thread therefore would print a new sequence of the output of the cyclic scheduler. Once the variable is null it would display the null task and whenever it is neither null nor zero it would display the results of the outputs of the cyclic scheduler. Where stringText consists of 700 characters to display the message of the analysis task, user interface task, and check acquisition management once the frameNum is not null and not zero.

Since the cyclic scheduler is invoked periodically by the clock thread, it is important to use a global variable to keep track of the current frame number.

### **3.3 User Interface Module Design**

The user interface module will use the Model-View-Controller (MVC) pattern to generate a display for the technician to view data and control the IDP system. The MVC pattern will be used to ensure proper separation of concerns and allow for easier debugging. The MVC pattern also makes it easier to implement the design, as the parts can be coded separately and then integrated.

#### **Model**

The model will be represented by the PORT\_STATE structures for each port, which will be updated with the data pertaining to each port. These structures are in the port module, and will be modified directly by the UI module using the getPortStateRef function. This function, along with the port module functions isPortEnabled and isRelayOpen will also be used to reference the model when constructing the view. The model will also consist of the error messages, headings, and port data templates that will be stored as strings.

#### **View**

The view will be the terminal which is updated with the data from the model. An example view is shown in Figure 3 below. Data from each source and load port is only printed if the port is enabled. When a source relay is open, only Vrms and Freq will be printed, while no data will be printed for an open load port. For a closed port, all data will be printed. The grid cannot be enabled or disabled, opened or closed, and only Vrms and Freq will be displayed. The UART module is used to communicate with the Tera Term terminal using the UART (Universal Asynchronous Receiver-Transmitter) protocol.

To print to the terminal, the command `uart_print` will be used, with a character array being passed. The view will be updated by first populating a 2D character array called `line[7][60]` with the data for each port, using the `\t` escape code for tabs and `\n` escape code for newlines. The data for each port will have to be formatted using the `sprintf` function to insert relevant data. Next, the screen will be cleared and the cursor moved to the upper left corner using the VT100 escape codes `clearscreen ED2` (`^[[2J`) and `cursorhome` (`^[[H`). Next, the user interface task will print all the column headings. The data for each enabled port stored in the line 2D array will be printed using the `uart_print` function of the `uart` module. Finally, the prompt for the technician to control the system will be printed along with any error messages. The prompt will either be to enter a command or choose a port number to apply the command. Error messages will persist until the user enters another command or chooses a valid port.

```

COM5 - Tera Term VT
File Edit Setup Control Window Help

          Intelligent Distribution Panel Simulator
State      Vrms      Irms      Freq      Pf      Real P      Reactive P
Source
ENABLE OPEN      0.00
ENABLE CLOSED    0.00  0.00000  0.00  0.000  0.000  0.000
DISABLED
Load
ENABLED OPEN
DISABLED
ENABLED CLOSED    0.00  0.00000  0.00  0.000  0.000  0.000
Grid
      0.00      0.00

Enter command (e - enable, d - disable, o - open, c - close):

**** Invalid UI Command (Input: b) ****

```

Figure 1: Desired UI

## Controller

The controller will control the prompt for the technician to enable and disable or close and open relays. The state machine shown in Figure 4 will be used to determine which prompt should be printed for the user at any given time. The states will be `ENTER_COMMAND`, `ENABLE`, `DISABLE`, `OPEN`, and `CLOSE`. The state will transition from `ENTER_COMMAND` to one of the other states when the user enters the corresponding letter. In the other states, the user will be prompted to enter a port number that will be enabled, disabled, opened, or closed. The state will then transition back to `ENTER_COMMAND`. For error handling, variables will be created to signal to the view whether an error message should be displayed, and which error message to display. Errors will consist of the user

entering an invalid command (not the letters e, d, c, or o) or the user entering an invalid port number when prompted.

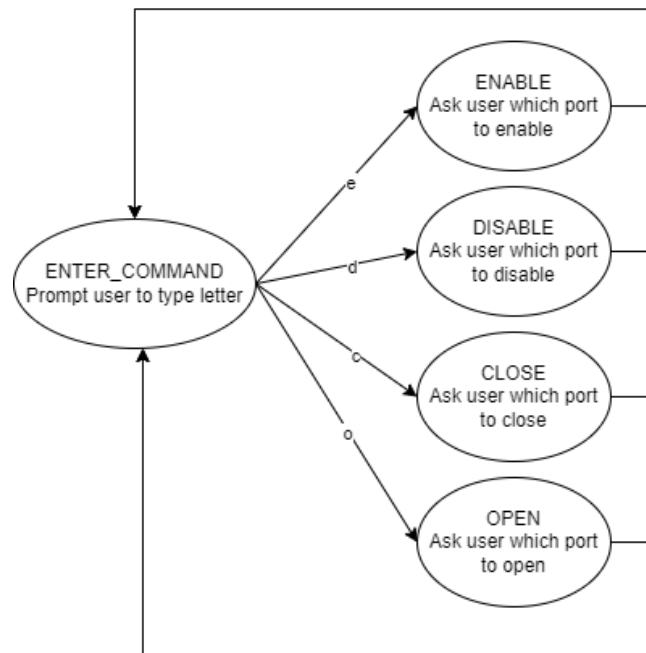


Figure 2: UI Controller FSM

## 3.5 Acquisition Module Design

### 3.5.1 Acquisition Module Overview

As per the Sprint 2 requirements, the acquisition module must be extended to collect data selectively and continuously on enabled ports as well as the grid. This will require an almost total revision of the module design. The Data acquisition module provides the functionality to control process of acquiring data from the hardware. The new module will use Swi thread functionality to control data collection, with separate Swi threads configured for the ADC, SPI0, and SPI1. This data collection will also invoke Hwi threads. For enabled ports, the Swi thread task will manage data collection, meaning that data collection will be continuous.

There are three system tasks that are implemented in the Acquisition module, shown in the table below:

Task	Thread	Invoked By	Other Related Modules
Source Acquisition Management	Swi1	SPI (Collect Source Port Data)	Port, SPI, GPIO
Load Acquisition Management	Swi2	SPI (Collect Load Port Data)	Port, SPI, GPIO
Grid Acquisition Management	Swi0	ADC (Collect Grid Data)	Port, ADC

Table 7: Tasks Implemented in Acquisition Module

The purpose of these three tasks is to ensure that there is always continuous data collection from the source/load/grid relays. These tasks will run when their respective invoker is called, as shown in Table 7. The grid port is always enabled, and there is only one such port. The



module will wait for the SPI or the ADC to call the collection tasks. The respective management task will run and continuously update its respective PORT\_DATA structure.

### **3.5.2 Acquisition Module Logic**

Acquisition is now initiated by the cyclic scheduler which starts the Check Acquisition Management task, implemented by the startAcquisition() function in this module. This function will check to see if we are currently acquiring data for each port group, and will only begin data collection for it if we are not. This is necessary when the program runs for the first time, and when the system enters a new configuration after all ports have been disabled. Data is collected by invoking HWI threads for the source, load, and grid ports.

After acquisition has been started by the Check Acquisition Management task, the remaining acquisition will be handled by a SWI thread for each group. When data collection is completed for a port, the respective ISR calls Swi\_post for the given handle (source, load, or grid), and the SYS/BIOS is configured to call acquire\_swi() in this module with a UArg parameter (unsigned int) specific to the calling port group - 1 for ADC, 2 for SPI0, and 3 for SPI1. These parameters are defined as arg0 for each SWI in the SYS/BIOS configuration file.

Because of this, we will be using a single function to implement all SWI tasks that will use this parameter in a switch statement. The function will then determine the next port to collect data from and call its respective HWI, and this process will repeat as per the following format:

**Grid Acquisition Management:** Since grid collection is continuous, this task will automatically invoke a Hwi thread to begin acquiring more grid data when it is called.

**Source Acquisition Management:** We select the next enabled port to gather source data from, determine whether we should also collect current data, and invoke a Hwi thread to collect data from it. If there are no enabled ports, we indicate we are not collecting source data anymore and return, waiting for the Check Acquisition Management task to begin data collection again.

**Load Acquisition Management:** We select the next port with a closed relay to gather source data from and invoke a Hwi thread to collect data from it (including current data). If there are no ports with closed relays, we indicate we are not collecting load data anymore and return, waiting for the Check Acquisition Management task to begin data collection again.

### **Changes from Sprint 1**

In this Sprint, separate structures have been created for the ports such that data that is in the process of being collected and analyzed is stored in a PORT\_DATA structure, and the results of the analysis as well as other information are stored in a PORT\_STATE structure.

An array of PORT\_DATA structures has been created for each group (source, load, and grid) to allow the analysis module to analyze data from a given group while we collect more data from it. The Port module contains functions that can be used to find a PORT\_DATA structure that has an empty buffer, which we will use to find one that is safe to fill with data.

However, since there are 3 source and load ports and only 2 spaces for each group in the array, we will keep track of which ports we have already collected data from and set the respective PORT\_ID in the PORT\_DATA array element to each new port we are going to collect from. This will be done using a set of global variables provided in the module that indicate whether we are currently collecting data from each group, as well as the IDs of the

most recent source and load ports we have collected from. We will collect data in a cyclic manner, going from the first to last ports in the source and load groups and then starting from the beginning again.

Acquisition is initiated as before by calling the startSpiAcq() and startAdcAcq() on a given port. However, in this case we must also first determine if a port is enabled to begin collection, and we must also examine a port's relayOpenStatus flag in the PORT\_STATE structure to determine if we will also analyze current in the case of a source port, or if we will analyze the port at all in the case of a load port. As the design document states:

- Collection of data for signals and their analysis (calculations) shall be done on a continuous basis.
- Data collection/analysis shall be done only for enabled ports.
  - When a source port relay is open, only the voltage signals are sampled.
  - When a source port relay is closed, both voltage and current signals are sampled.
  - When a load relay is open, no signals are sampled.
  - When a load relay is closed, both voltage and current signals are sampled.
  - Data collection/analysis is always done for the grid (cannot be disabled).

The Port module provides functions that allow us to determine each port's state and whether we should collect current. Load ports with open relays will be skipped, as per the instructions.

The analysis module is responsible for freeing data buffers in the PORT\_DATA structures and setting their bufFull variables to False after it is finished with them, thereby enabling us to use them to collect more data.

The cyclic scheduler is configured in such a way that there will be enough time for both the analysis and acquisition task to run, ensuring that we don't enter into a scenario where we are trying to acquire data from a given port group and the bufFull variables in both of its PORT\_DATA structures is set to True.

Additionally, the system is designed to support a continuous data collection, so the only time we need to stop is when all ports have been disabled so as to avoid being stuck in a busy-waiting loop checking for enabled ports. The startAcquisition function will then resume data collection when at least one port becomes available again.

## Functions

### 1. acquire\_swi(UArg uarg)

This function is called when data collection has been completed for one of the ports. The SYS/BIOS is configured to call this function with a UArg parameter (unsigned int) specific to the calling port group - 1 for ADC, 2 for SPI0, and 3 for SPI1.

When we begin collecting data from a new port/the grid, we must update the respective global variables to point to the corresponding PORT\_ID and PORT\_DATA structures.

Data collection from the grid is continuous, so if the calling group is the grid we must continue to call startAdcAcq() on it.

Otherwise, we must use the currentSourcePort or CurrentLoadPort variables along with the enable status in the PORT\_STATE structures to determine which port to collect data from next by calling startSpiAcq(). The open and closed state of the port is used to determine if we should analyze current for source ports, and we must also skip load ports with open relays.

Additional global variables have been created to facilitate the checking of the next available source or load port, with the assumption that we only have 3 available ports each for source and load.

If all ports for the group are disabled, we indicate we are not collecting data for this group anymore and return, waiting for startAcquisition() to begin again in the next cycle.

## 2. startAcquisition()

This function is called by the cyclic scheduler and is used to begin acquiring data if acquisition is stopped. It begins acquisition for all 3 port groups, and its manner of deciding which port to analyze next for each group is the same as the above function.

In each case, we check if we are already collecting data for the given port group and only begin acquisition if we are not, as the acquire\_swi function will handle this if we are. Additionally, if all ports for the group are unavailable, we do nothing.

## 3. initAcquisition(int timerHandle)

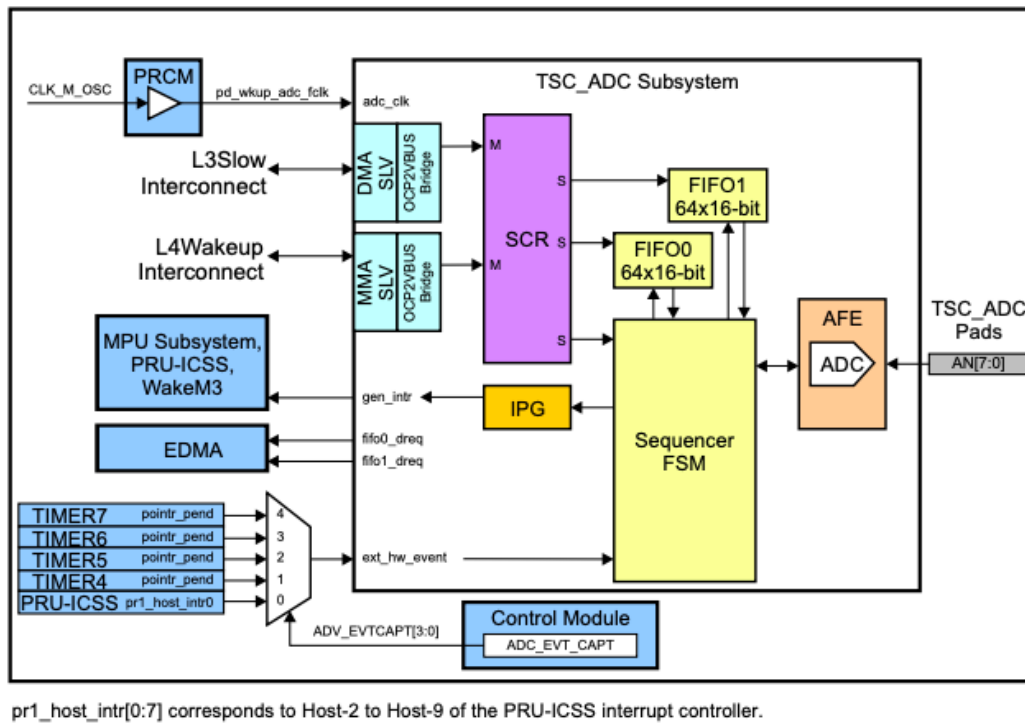
This function initializes SPI0, SPI1, and the ADC modules so they can begin collecting data.

# 3.8 ADC Module Design

## 3.7.1 ADC Hardware Module Overview

The ADC interface is a general-purpose analog-to-digital converter with eight channels for a 4-wire, 5-wire, or 8-wire resistive panel. In the context of the IDP, the ADC will monitor the voltage across the relay and the current that goes through it using four channels. The ADC digitises the sampled input by collecting signal data from the grid over 12 clock cycles. The converted A/D sample will be 12-bits.

**Figure 12-1. TSC\_ADC Integration**



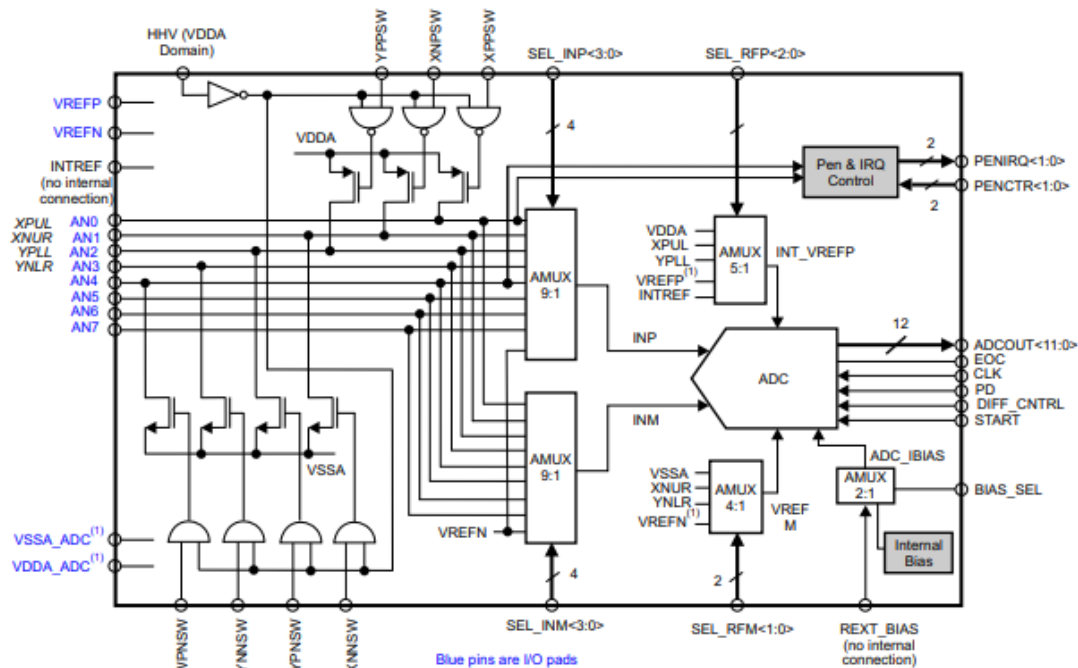
*Figure 3: Shows the integration of the TSC\_ADC module in the device*

The 'Sequencer FSM' is responsible for turning on or off the switches in the AFE (Analog Front End) and the FIFO.

There are 2 FIFO in this module. These registers are 16 bit wide and can be configured by the CPU with the help of DMAENABLE\_SET register. Programmable threshold is also present in the FIFO registers. The flags FIFO0THRESHOLD/ FIFO1THRESHOLD are set once the threshold is reached. The CPU can read the number of entries in FIFO with FIFO0COUNT/ FIFO1COUNT.

The CLK\_M\_OSC signal is the ADC clock operating at 24 MHz. The analog signal, AN[7:0] is converted to a data stream by the AFE. The data stream is then written to FIFO. The sequencer establishes the AFE's mode of operation.

**Figure 12-2. Functional Block Diagram**



(1) In the device-specific datasheet:

- VDDA\_ADC and VSSA\_ADC are referred to as "Internal References"
- VREFP and VREFN are referred to as "External References"

*Figure 4: Functional block diagram showing the ADC and its connections*

The above figure shows a clearer view of the AFE and its various inputs and outputs. AN[7:0] can be used as a 8-wire general purpose ADC. But for this project, we only have 2 signals:  $V_h$  and  $V_n$ , so only AN0 and AN1 will be used. The control signal is sent by the sequencer for the AFE to act as an ADC to convert the 2 signals. The 2 AN signals are then converted by the ADC. The ADC then digitizes the output on ADCOUT[11:0] bus.

According to the AM335x manual, a 'step' is a term used for sampling a channel input. The programmer defines it, and with the sequencer it will send values to AFE and determine how and when to sample a channel. The registers are programmed with the help of the STEPENABLE, the STEPCONFIGx and the STEPDELAYx registers ( $x = 1$  to 16). 16 programmable steps, a touchscreen step and an idle step are supported by the sequencer. The idle step will always be enabled, it does not have a delay register and does not sample on any channel. The sequencer will trigger a STEP once a STEPENABLE bit is turned on. After the completion of a step, an END\_OF\_SEQUENCE interrupt is generated. The sequencer then goes through the steps (1 – 16).

The steps can be set up to be averaged when sampled. This can be done over 1, 2, 4, 8 or 16 samples.

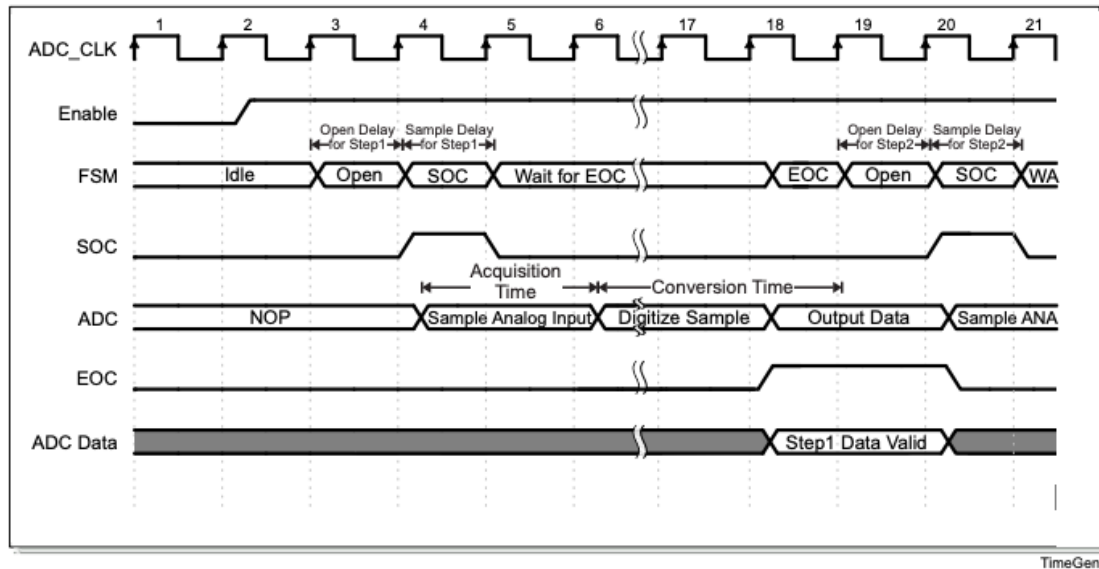


Figure 5: Example timing diagram for Sequencer

According to the data provided in the AM335x Manual, the ADC is able to sample at 14 ADC clocks per sample with the slightest delays enabled.

### 3.7.2 Sampling

The data sampling will be done in a continuous mode because we want to sample 1024 samples. STEP1 and STEP2 will be enabled for us to sample 2 channels, AN0 and AN1, one immediately after the other. When the STEPENABLE bit is turned on, the step is enabled. If the mode is one-shot, then after the completion of the step, it will turn off the STEPENABLE bit. An END\_OF\_SEQUENCE interrupt will also be generated.

There is a difference between the signal frequency and the conversion time, and therefore an open delay is used. The length of ADC's clock cycle is 0.042 micro-seconds.

### 3.7.3 Design

The interrupt bits, IRQSTATUS\_RAW, for the ADC module are defined in the table below. Each bit will be of type R/W and the reset is set to 0h. The definitions of these bits can be found in the 'adcDefinitions.h' file.

Bit	Field	Description
7	FIFO1_Underflow	Write 0 = No action Write 1 = Set event (debug) Read 0 = No event pending Read 1 = Event pending
6	FIFO1_Overflow	Write 0 = No action Write 1 = Set event (debug) Read 0 = No event pending Read 1 = Event pending.
5	FIFO1_Threshold	Write 0 = No action Write 1 = Set event (debug) Read 0 = No event pending Read 1 = Event pending

4	FIFO0_Underflow	Write 0 = No action Write 1 = Set event (debug) Read 0 = No event pending Read 1 = Event pending.
3	FIFO0_Overflow	Write 0 = No action Write 1 = Set event (debug) Read 0 = No event pending Read 1 = Event pending.
2	FIFO0_Threshold	Write 0 = No action Write 1 = Set event (debug) Read 0 = No event pending Read 1 = Event pending.
1	End_of_Sequence	Write 0 = No action Write 1 = Set event (debug) Read 0 = No event pending Read 1 = Event pending.

Table 6: Interrupt Bits, their fields and descriptions

The register structure for the ADC module is shown below. This structure can be found in the ‘adcDefinitions.h’ file. It should be noted that each one is of type uint32\_t, which is used to guarantee a 32-bit integer:

Name	Offset	Purpose
revision	0x0	ADC revision, configures the system
resvd1[3]	0x4 - 0xFF	Reserved bytes
sysconfig	0x10	ADC sysconfig, configures the system
rsvd2[4]	0x14 - 0x23	Reserved bytes
irqstatus_raw	0x24	IRQ status unmasked
irqstatus	0x28	IRQ status register
irqenable_set	0x2C	IRQ enable set register
irqenable_clr	0x30	IRQ enable clear register
irqwakeupt	0x34	IRQ wakeup register
dmaenable_set	0x38	DMA enable clear register
dmenable_clr	0x3C	DMA enable clear register
cntrl	0x40	ADC control register
stat	0x44	ADC status register
range	0x48	ADC range register
clkdiv	0x4C	ADC clock divider register
misc	0x50	ADC miscellaneous register

stepenable	0x54	ADC step enable register
idleconfig	0x58	ADC idle config register
ts_charge_stepconfig ts_charge_delay	0x5C 0x60	ADC sampling time charge step config register ADC charge delay register
step[NUM_STEPS][2]	0x64 - 0xE3	configuration and step registers 1 to 16 (indexed from 0-15)
fifo0count fifo0threshold	0xE4 0xE8	ADC FIFO 0 count register ADC FIFO 0 Threshold register
dma0req	0xEC	ADC DMA 0 request register
fifo1count	0xF0	ADC FIFO 1 count register
fifo1threshold	0xF4	ADC FIFO 1 Threshold register
dma1req	0xF8	ADC DMA 1 request register
rsvd3	0xFC	reserved bytes
fifo0data	0x100	ADC fif0 data register
rsvd4[63]	0x104 - 1FF	reserved bytes
fifo1data	0x200	ADC fif1 data register

*Table 7: Register structure of ADC and offset values*

Registers that need to be initialised:

- sysconfig
- irqenable\_clr
- cntrl
- clkdiv
- stepenable
- idleconfig
- stepconfig
- fifo0threshold



- fifo1threshold

26	FIFO_select	R/W	0h	Sampled data will be stored in FIFO. 0 = FIFO. 1 = FIFO1.
22-19	SEL_INP_SWC_3_0	R/W	0h	SEL_INP pins SW configuration. 0000 = Channel 1. 0111 = Channel 8. 1xxx = VREFN.
4-2	Averaging	R/W	0h	Number of samples to average: 000 = No average. 001 = 2 samples average. 010 = 4 samples average. 011 = 8 samples average. 100 = 16 samples average.
1-0	Mode	R/W	0h	00 = SW enabled, one-shot. 01 = SW enabled, continuous. 10 = HW synchronized, one-shot. 11 = HW synchronized, continuous.

The STEPCONFIG registers have the FIFO\_select, which can be set to 1 or 0, the SEL\_INP\_SWC\_3\_0, from which we can select AN0 and AN1, the ‘averaging’ to select 16 samples, and the mode bits, to select continuous mode.

**Table 12-24. STEPDELAY1 Register Field Descriptions**

Bit	Field	Type	Reset	Description
31-24	SampleDelay	R/W	0h	This register will control the number of ADC clock cycles to sample (hold SOC high). Any value programmed here will be added to the minimum requirement of 1 clock cycle.
23-18	RESERVED	R/W	0h	
17-0	OpenDelay	R/W	0h	Program the number of ADC clock cycles to wait after applying the step configuration registers and before sending the start of ADC conversion

In the above table, we can see how OpenDelay is configured in STEPDELAY<sub>x</sub> (x = 1 to 16).

**Table 12-9. IRQENABLE\_SET Register Field Descriptions (continued)**

Bit	Field	Type	Reset	Description
2	FIFO0_Threshold	R/W	0h	Write 0 = No action. Read 0 = Interrupt disabled (masked). Read 1 = Interrupt enabled. Write 1 = Enable interrupt.

*Figure 6: FIFO0 Threshold Interrupt Fields*

In the above figure, we see how to enable the FIFO0\_Threshold interrupt.

**Table 12-19. STEPENABLE Register Field Descriptions**

Bit	Field	Type	Reset	Description
31-17	RESERVED	R	0h	RESERVED.
16	STEP16	R/W	0h	Enable step 16
15	STEP15	R/W	0h	Enable step 15
14	STEP14	R/W	0h	Enable step 14
13	STEP13	R/W	0h	Enable step 13
12	STEP12	R/W	0h	Enable step 12
11	STEP11	R/W	0h	Enable step 11
10	STEP10	R/W	0h	Enable step 10
9	STEP9	R/W	0h	Enable step 9
8	STEP8	R/W	0h	Enable step 8
7	STEP7	R/W	0h	Enable step 7
6	STEP6	R/W	0h	Enable step 6
5	STEP5	R/W	0h	Enable step 5
4	STEP4	R/W	0h	Enable step 4
3	STEP3	R/W	0h	Enable step 3
2	STEP2	R/W	0h	Enable step 2
1	STEP1	R/W	0h	Enable step 1
0	TS_Charge	R/W	0h	Enable TS Charge step

*Table 8: STEPENABLE Register Field Descriptions*

In the above table, we can see how to enable to STEP1 and STEP2 with the STEPENABLE register.

**Table 12-14. CTRL Register Field Descriptions**

Bit	Field	Type	Reset	Description
0	Enable	R/W	0h	TSC_ADC_SS module enable bit. After programming all the steps and configuration registers, write a 1 to this bit to turn on TSC_ADC_SS. Writing a 0 will disable the module (after the current conversion).

In the above table we see how to enable the TSC\_ADC\_DD module.

A C language software module, adc.c, is designed to:

- Convert  $V_h$  and  $V_n$  voltage signals to numerical values.
- Control the ADC device in the AM335x microcontroller.

Some of the variables and definitions in the adc.c file:

- ADC\_REGS – a pointer to the structure of registers.
- ADC – a structure variable type that contains all necessary variables for performing data collection:
  - A variable “done” of type *bool*.
  - “swiHndl” of type *Swi\_Handle*, for acquisition Swi. It is used to raise Swi interrupt when collection is done. The swi handler will also be used to invoke a Swi thread from the ISR (using Swi\_post function) or raise a SWI to indicate the end of data collection.
  - A variable for the number of samples to be collected, “numSamples” of type *uint32\_t*.

- A variable for the number of samples collected, “numCollected” of type *uint32\_t*.
- A pointer to buffers, “\*buffers” of type *uint16\_t*.
- A “timerHandle” of type *int* for time stamps.
- “\*endTime” of type *uint32\_t*.
- **Extern** Swi\_Handle acqSwi\_adc for Swi configuration.

The module consists of four functions.

adc.c functions:

1. *Void* initADC(*int* timerHandle, *Swi\_Handle* swiHndl)

This function initializes the ADC device. It also configures the ADC device for collecting data from the Grid. Samples are collected over approximately 16.7 ms which is about 1 cycle of a 60 Hz signal. No current values can be collected for the grid.

One of the two parameters used is an integer named “timerHandle” which is the handle of the wall clock timer that has been initialized and is counting. The other is “swiHndl”, a handle to the SYS/BIOS swi object used for raising the SWI interrupt of the grid acquisition management task.

2. *int* startAdcAcq(*uint32\_t* \*start\_time, *uint32\_t* \*end\_time, *uint16\_t* databufs[], *int* numsamples)

This function starts the acquisition of data into a buffer from the ADC. It also configures and enables the ADC device to collect “numSamples” samples and stores raw values into a data buffer. The total number of samples collected is “DATA\_SIZE” (1024); half for  $V_h$  and half for  $V_n$ . The values which are collected and stored alternate between  $V_h$  and  $V_n$ . Start time values and end time values of the data acquisition are also saved. A parameter named “start\_time” of type *uint32\_t* (4 byte unsigned integer) is set to the time when the collection starts. Another parameter named “end\_time” of type *uint32\_t* is set to the time when the collection ends. A parameter of type *uint16\_t* (4 byte unsigned integer) named “databufs” points to an array where the collected data is stored. “numSamples” (integer) is the number of samples to be collected. The function returns an integer; ADC\_STARTED if started or ADC\_BUSY if currently collecting data.

3. *void* init\_hw()

Configures the ADC device to meet the following requirements:

- Digitizes the signals on AN0 ( $V_h$ ) and AN1 ( $V_n$ ) in step 1 and step 2 respectively.
- Use the 24 MHz clock with no scaling (1/24 microsecond = 41 2/3 nano-seconds)
- Conversion sequence consists of Step 1 with 333 clock open delay follow by Step 2 with no Open delay. Both steps use single clock Sample Delay.
- Both steps store conversion data in FIFO0, such that data read will alternate between the signals, i.e. :  $V_h V_n V_h V_n \dots$
- Interrupts are generated on the FIFO0 threshold, configured at a threshold level of 32.

The conversions are started/terminated by enabling/disabling the ADC enable bit in the CNTRL register (BIT0).

‘sysconfig’ is set to 0x10, which configures the system. ‘stepenable’ is set to 0 (0x01 | 0x10) to enable STEP1 and STEP2.

When enough samples have been read from the FIFO0, there may be additional samples left in the FIFO. The number of samples left should be considered when determining the end time of the last sample read.

#### 4. *void* adc\_isr(UArg arg)

This function is an Interrupt Service Routine (ISR) function. *adc\_isr* is configured in SYS/BIOS to ADC interrupt which invokes the *adc\_isr* function with a reference to the appropriate ADC structure that contains all the data used for collection over the ADC interface. The Swi handler variable in the ADC structure will be used in this function to raise a SWI indicating the end of data collection.

The parameter ‘arg’ is invoked in SYS/BIOS.

## 3.10 GPIO Module Design

### 3.10.1 GPIO Hardware Module Overview

The GPIO interface is composed of four modules, each providing 32 general-purpose pins each. Only three of the four modules provided are used in the IDP, and therefore 96 dedicated pins are available for input and output configuration. Each module has 32 identical channels which may be configured for different applications such as:

- Data input/output
- Keyboard interface
- Synchronous interrupt generation
- Wake-up request generation
- Shared register access

In the case of the IDP, the GPIO will be primarily used for opening and closing relays as well as selecting ADC chips. A software module is designed in 3.9.2 to control the pins and fully support the functionality of the GPIO modules. GPIO3 is not used in IDP. Each GPIO module is attributed to its own register with a base address. The base addresses for each are listed below in table 9:

GPIO	Address
0	0x44E07000
1	0x4804C000
2	0x481AC000
3 (Not Used)	0x481AE000

*Table 9: Base Addresses for GPIO Modules*

The register structure for each GPIO module is shown below. Reserved locations in memory are not shown. Each one is of type *uint32\_t*, which is used to guarantee a 32-bit integer. The

four main registers that will be used will be in bold, these are output enable (oe), dataout, cleardataout, and setdataout. These are the primary registers that will be altered by the functions in gpio.c:

Name	Offset Address	Purpose
revision	0x00	Read-Only. Gives revision number
sysconfig	0x10	Controls parameters of L4 interconnect
eo	0x20	Supports DMA events
irqstatus_raw_0	0x24	Provides core status information for interrupts
irqstatus_raw_1	0x28	
irqstatus_0	0x2c	Provides core status information for enabled interrupts
irqstatus_1	0x30	
irqstatus_set_0	0x34	Enables specific interrupt event to trigger
irqstatus_set_1	0x38	
irqstatus_clr_0	0x3c	Clears specific interrupt event to trigger
irqstatus_clr_1	0x40	
irqawaken_0	0x44	Enables wakeup events on interrupt
irqawaken_1	0x48	
sysstatus	0x114	Provides reset status
ctrl	0x130	Controls clock gating functionality
<b>oe</b>	<b>0x134</b>	<b>Output enable. Clears bit to 0</b>
datain	0x138	Registers data input from GPIO pins
<b>dataout</b>	<b>0x13c</b>	<b>Sets value for GPIO output pins</b>
leveldetect0	0x140	Enable or disable low-level interrupts
leveldetect1	0x144	Enable or disable high-level interrupts
risingdetect	0x148	Enable or disable line rising-edge interrupt
fallingdetect	0x14c	Enable or disable line falling-edge interrupt
debouncingenable	0x150	Enable or disable debouncing
debouncingtime	0x154	Controls debouncing time for all ports
<b>cleardataout</b>	<b>0x190</b>	<b>Clears dataout to 0 bits</b>
<b>setdataout</b>	<b>0x194</b>	<b>Sets dataout to 1 bits</b>

Table 10: Register Structure for GPIO Modules

### 3.10.2 GPIO Design

In terms of design, the GPIO module for Sprint 2 does not have major modifications from its previous iteration. As per the requirements for Sprint 2, The goal of the GPIO is still the same, which is to allow the opening and closing of source and load relays. The GPIO must also be able to select the ADCs on the SRC's. For this purpose, a software module, gpio.c, is designed to control the pins and support the stated functionality. It is comprised of four functions that allow initialization and modification of the dataout register.

In order to access the registers, the header files gpio.h, and gpioDefinitions.h are imported. These provide the register structure and definitions necessary to allow the gpio.c file to access the registers. The GPIO out pins must be defined to the appropriate bits as shown below in figure 6.

```
#define GPIO0_OUT_PINS (BIT8 | BIT9 | BIT10 | BIT22 | BIT23 | BIT26 | BIT27)
#define GPIO1_OUT_PINS (BIT12 | BIT13 | BIT14 | BIT15 | BIT16 | BIT17 | BIT18 | BIT19)
#define GPIO2_OUT_PINS (BIT1 | BIT6 | BIT7 | BIT12 | BIT23 | BIT24 | BIT25)
```

Figure 6: GPIO bit definitions

gpio.c functions:

#### 1. *void initGPIO()*

This function enables GPIO0, GPIO1, and GPIO2. As mentioned before, GPIO3 is not used in the IDP. This function also configures the output pins. This allows the IDP to control the relays and address the ADCs on relays. The GPIO\_REGS structure, which is defined under gpioDefinitions.h, will be used to give each of the GPIO modules the correct structure and to apply its correct base address to it. For example, the void initGPIO() function will initialize GPIO0 with its base address 0x44E07000 and apply the register structure shown in Table 10 to GPIO0.

When the GPIO is first initialized, all the bits are set to 0. The variable GPIO0 will now be initialized, and the contents of the register will be accessible by the next functions. For this purpose, the *oe* register is used. As discussed in the previous section, the output enable register clears all the bits to 0.

#### 2. *void bitSet(int gpioNum, uint32\_t bits)*

The *bitSet* function uses the setdataout (Offset 0x194) register to set the selected GPIO module's dataout register. The parameter *gpioNum* selects which GPIO to perform the function on. Either 0,1, or 2 to select GPIO0, GPIO1, or GPIO2. The bits parameter is used to select which corresponding bits to set to 1 in the dataout register. The bits in this parameter that are set to 1 will be set to 1 in the corresponding GPIO's dataout register. If the bits parameter is set to 0, then the dataout register will not be affected.

#### 3. *void bitClear(int gpioNum, uint32\_t bits)*

The *bitClear* function uses cleardataout (Offset 0x190) to clear the selected GPIO module's dataout register. The *gpioNum* parameter selects which GPIO to perform the function on. The *bits* parameter is used to select which bits to clear. Bits in this parameter that are set to 1 will be set to 0 in the corresponding GPIO's dataout register. If the bits parameter is set to 0, then the dataout register will not be affected.

#### 4. *bool isBitSet(int gpioNum, uint32\_t bits)*

This function will check if bits are cleared in the dataout register. It will return a 0 if all the bits defined by *bits* are cleared. Otherwise, it returns a non-zero value. This function can use boolean operations to compare the two given parameters. The operation that will be used to conduct this comparison is the following:

```
if((gpio_regs[gpioNum]->dataout | ~bits) == ~bits)
```

*gpio\_regs[gpioNum]->dataout* represents the content of the register of *gpioNum*. This is being compared with the given *bits* variable. An OR operation is performed with the complement of the *bits* variable. If this operation returns the same complement of *bits*, then that means that all values that are '1' in 'bits' are a '0' in the data register.

As can be seen by the above functions, the GPIO software controller allows the modification of the dataout register. The last three functions share the parameters *gpioNum* and *bits* which allow the exact bits of a specific GPIO's dataout register to be modified. The *bitSet* function will set the necessary bits to 1, and then the *bitClear* function will set the necessary bits to 0, allowing all GPIO modules to freely modify their respective dataout registers. This allows the GPIO interface to target specific ADC chips and to open and close relays. Once again, the four registers of importance that were used in these functions are *oe*, *setdataout*, *cleardataout*, and *dataout*.

## Implementation

### Cyclic Scheduler Implementation

The cyclic scheduler's implementation evolved from the original design submitted in deliverable 1. The new changes are based around having a custom type Task array that represents the scheduling of the tasks instead of just calculating based off a global variable. This means that there is more flexibility when it comes to modifying the order of the tasks being scheduled, as well as better readability when it comes to the implementation of the scheduler.

### GPIO Implementation

The GPIO development involved implementing four functions in the *gpio.c* file. These functions, as defined in section 3.9.2, are used to initialize the GPIO, set bits, clear bits, and to check if bits are set. This file has a relatively straightforward implementation and each of the functions were implemented in a concise and simple manner. The initialization function sets up the three modules and sets all the pins to 0. This was done using the Output Enable (OE) register which clears all bits to 0. The OE register was applied to all three of the modules. The 'bitSet' and 'bitClear' functions are two similar functions using the *setdataout* and *cleardataout* registers respectively. One simple line is attributed to each function which simply sets the bits in the corresponding GPIO to the bits as defined by the 'bits' variable in the function.

The last function is 'isBitSet'. This function is slightly more complicated in that boolean operations needed to be used to compare the two given values. Writing this function required finding a simple set of Boolean operations that would successfully compare if the 1s in 'bits' were 0s in 'gpioNum'. To do this the following operation was used:

```
gpioNum | ~bits == ~bits
```

The `|` represents OR in Boolean, and the `~` represents a complement. So, if the *gpioNum* OR the complement of 'bits' equals to the complement of 'bits', then that means that all the 1s in 'bits' are 0s in the data register.

Many changes were made throughout the GPIO design before the implementation. As more research was done, the design was refined. The initial design that was submitted during deliverable 1 was relatively unused. The original pseudocode was discarded as the idea of using Boolean operations to compare the registers was not considered at the time. A lot more understanding was gained during the development. When developing, it is of utmost importance to come to clear, concise, and simple solutions. The original design, which used for loops to iterate through the bits, was simply inefficient and not nearly as effective as using

a Boolean operation. Overall, the design was refined, and the implementation was straightforward regarding the GPIO module.

The implementation of the GPIO module can be found in gpio.c.

## ADC Implementation

For the ADC module, three out of four functions had to be implemented in the adc.c file. These were:

- ‘initADC’, for this function the logic was simple:
  - Set the local timer handle from the adc struct to the one provided in the parameter.
  - Set the local SWI handle from the adc struct to the one provided in the parameter.
  - Activate the module, using activateModule() function.
  - Initialize the hardware using the local init\_hw() function.
  - Set ‘done’ from the adc struct as *true*.
- ‘startAdcAcq’, the implementation of this function was a bit tricky. The logic developed:
  - Initialise an integer variable ‘retval’ and set it to ADC\_BUSY
  - If ‘done’ from the ADC struct is *true*:
    - ‘done’ from the ADC struct is set to *false*.
    - getCounter(adc.timerHandle) is used to get ‘start\_time’
    - ‘buffers’, ‘end\_time’ and ‘numSamples’ from the ADC struct are set to ‘databufs’, ‘end\_time’ and ‘numsamples’ respectively (parameters of this function)
    - ‘numCollected’ from ADC struct is set to 0.
    - ADC is enabled by using ‘cntrl’ register from ‘adcRegs’
    - ‘retval’ is set to ADC\_STARTED
  - ‘retval’ is returned
- ‘adc\_isr’, this is an interrupt service routine. The implementation:
  - Start a while loop. While there are words to read (adcRegs->fifo0count > 0) && number of samples collected is less than the number of samples to be collected (adc.numCollected < adc.numSamples):
    - read data from ‘fifo0data’. Close while bracket.
  - If the number of samples collected is equal to the number of samples to be collected:
    - Set ‘done’ from ADC struct to *true*.
    - Use getCounter to get ‘end\_time’
    - Disable ADC
    - Use Swi\_post function with Swi handle as the parameter to invoke a Swi thread from this ISR function.
  - Clear the interrupt by setting ‘irqstatus’ from ‘adcRegs’ to ‘FIFO0\_THRESHOLD\_BIT’

The implementation for the ADC module can be found in adc.c