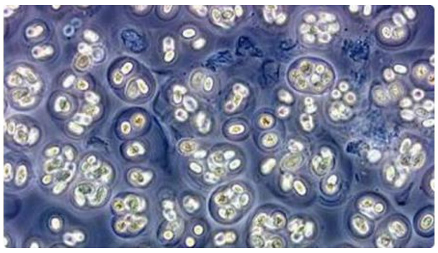
Diseño Respirómetro

De la Rosa F [(*)].; Navarro R. R [(**)].; Vázquez E. E. [(*)]; Moya N. S.; Murillo, M. D.; Ronveaux M. T [(* **)].
PID “Calidad Ambiental de Agua y Aire”
CEDIA-FRT-UTN
(*) Integrantes del PID – (*) Colaborador del PID – (* **) Integrante del PID y Colaborador en documentación.

Introducción



En este documento se describe el diseño de un Respirómetro, para la medición del consumo biológico de oxigeno por una población microbiana. La medición se realiza en forma directa en fase liquida. El aporte de oxigeno es discontinuo y está controlado por el usuario mediante una serie de parámetros para el cálculo automático de la VECO[1] y el CAO[2].

En el diseño realizado la concentración de oxígeno en la fase liquida es realizada a través de un sensor de oxígeno disuelto y no se produce alimentación ni de gas ni de líquido durante la medición, este tipo de instrumentos se clasifica como LSS.

Debido a los múltiples procesos físicos químicos involucrados es necesaria una lógica de control compleja control que determine las concentraciones máximas y mínimas de oxígeno disuelto y cuando la muestra está en condiciones de ser medida, evitando los errores producidos por mediciones durante la etapa de inicio del proceso.

Además, el sistema de control debe verificar que la temperatura y velocidad de agitación del sistema sean constantes. Los datos relevados son almacenados para calcular la pendiente de la curva oxigeno – tiempo[3] que corresponde a dicho ciclo.

Estos valores son de importancia fundamental para el cálculo del VECO y del CAO que son los parámetros que el sistema calcula y muestra en forma automática sin embargo se pueden generar datos adicionales y curvas a partir de la información anterior y un análisis estadístico de la misma.

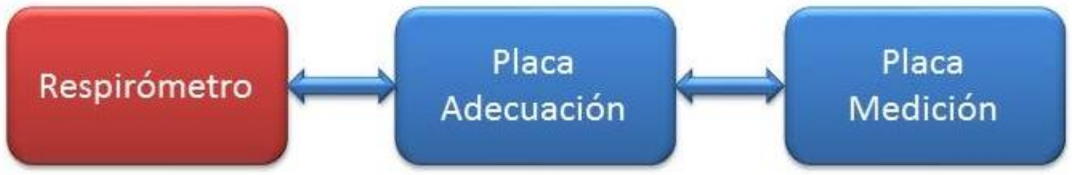


Figura n° 1 – Diagrama de bloques del sistema

La figura n°1 muestra el diagrama en bloques del sistema, describiéndose a continuación cada uno de los elementos:

[1] Velocidad Especifica de Consumo de Oxigeno.
[2] Consumo Acumulado de Oxigeno.
[3] Zona de toma de datos

Respirómetro

Este bloque es la parte física o planta del sistema donde se producen las reacciones biológicas, está constituida por:

- **Matraces** de reacción que son los recipientes donde se producen las reacciones biológicas
- **Depósito** de rebose constituido por cámaras de expansión de vidrio
- Baño como la actividad microbiana depende de la temperatura se ha implementado este con depósito y un termostato de inmersión.
- **Agitadores magnéticos** que se utilizan para homogeneizar la disolución y favorecer la transferencia de oxigeno
- **Aireadores** que se utilizan para la inyección de aire en los matraces de reacción
- **Sensores** que es un sensor de oxígeno disuelto conectado a la placa de adecuación.

Placa de Adecuación

La placa de adecuación es controlada por la placa de control, esta controla las electroválvulas que se encuentran en el respirómetro y los motores utilizando un sistema de control por semiciclo de alterna. En el caso de las electroválvulas para implementar un control on / off y en el de los motores para controlar la velocidad de funcionamiento de los mismos.

Placa de control

La placa de control ha sido implementada alrededor de un circuito integrado diseñado por este equipo de investigación específicamente para esta aplicación, este ASIC[4] se encarga de manejar la placa de potencia, procesar las señales recibidas y manejar la interfaz de usuario del sistema.

Cronograma de Actividades

Durante la ejecución de las tareas en el periodo 2015 se han realizado las siguientes actividades.

Actividades	Meses											
	1	2	3	4	5	6	7	8	9	10	11	12
1. Análisis y diseño de la placa de control	🟦											
2. Realización de esquemas eléctricos		🟦										
3. Simulación de circuitos			🟦									
4. Diseño de PCB				🟦								
5. Montaje de componentes y Prueba					🟦							
6. Diseño de detalle				🟦	🟦	🟦						
7. Simulación y prueba del código							🟦					
8. Análisis y diseño de la placa de potencia								🟦				
9. Simulación de circuitos									🟦			
10. Diseño de PCB										🟦	🟦	
11. Montaje de componentes y Prueba											🟦	
12. Pruebas Generales												🟦

Cronograma de actividades

[4] Circuito Integrado de Aplicación Especifica

Descripción de actividades

1. En esta etapa se recopilo la información correspondiente a las distintas alternativas para la implementación de una placa de control basada en microcontroladores. Se analizó los requisitos tantos de hardware como de firmware del dispositivo a implementar, decidiéndose por la utilización de un ASIC de diseño propio el cual se implementaría en un PSOC ya que permitía una alta escala de integración y gran fiabilidad en las medidas.
2. Se realizaron el diseño del ASIC y los esquemas eléctricos de la placa de control en función del análisis realizado en la etapa 1. Para ello se utilizó el software EDA Altium Designer 2013. Para la implementación del ASIC se utilizó el software Psoc Creator.
3. Se simuló el comportamiento de los circuitos para verificar su correcto funcionamiento. Utilizándose las herramientas de software proporcionadas por el EDA Altium Designer que se basan en Pspice. El comportamiento verificado de los circuitos se encontró dentro de los parámetros especificados por lo que no fue necesario realizar modificaciones en los mismos.
4. Se diseñó el PCB correspondiente a los esquemas eléctricos de la Etapa 3, para ello se utilizó un PCB de tipo FR35, con dos layer. El diseño corresponde a un circuito clase V, El pcb fue implementado con las siguientes características:
   - Metalizado de 25 µm según IPC clase 3
   - Laminados con revestimiento de cobre según IPC4101, clase B/L
   - Mascara de soldadura según IPC SM-840 Clase T
   - Mascara Peters SD2595
5. Montaje de componentes y prueba, en esta etapa se procedió al montaje de los componentes y la prueba de la placa de control en función de las especificaciones del sistema. Las pruebas fueron realizadas tanto en simulación como en modo real. Los resultados obtenidos fueron satisfactorios dándose por terminada esta etapa.
6. Diseño de detalle, en función de las especificaciones y el análisis de la etapa 1 se procedió a diseñar los algoritmos para la implementación del firmware de la placa de control. Estos algoritmos utilizan un sistema de lógica PID para el control del sistema.
7. Simulación y prueba de código, con el firmware desarrollado en la etapa 6 se utilizó un sistema de simulación del respirómetro dado que no está disponible hasta el momento el sensor de oxigeno disuelto
8. En esta etapa se recopilo la información correspondiente a las distintas alternativas para la implementación de una placa de adecuación controlada por ASIC. Se analizó los requisitos de hardware del dispositivo a implementar, decidiéndose por la utilización de una placa con control aislado ópticamente y conmutación en cruce por 0.
9. Se simuló el comportamiento de los circuitos de la placa de control para verificar su correcto funcionamiento. Utilizándose las herramientas de software proporcionadas por el EDA Altium Designer que se basan en Pspice. El comportamiento verificado de los circuitos se encontró dentro de los parámetros especificados por lo que no fue necesario realizar modificaciones en los mismos.
10. Se diseñó el PCB correspondiente el diseño de la etapa 8, para ello se utilizó un PCB de tipo FR35, con dos layer. El diseño corresponde a un circuito clase III, El pcb fue implementado con las siguientes características:
    - Metalizado de 25 µm según IPC clase 3
    - Laminados con revestimiento de cobre según IPC4101, clase B/L
    - Mascara de soldadura según IPC SM-840 Clase T
    - Mascara Peters SD2595

11. Montaje de componentes y prueba, en esta etapa se procedió al montaje de los componentes y la prueba de la placa de adecuación en función de las especificaciones del sistema. Las pruebas fueron realizadas tanto en simulación como en modo real. Los resultados obtenidos fueron satisfactorios dándose por terminada esta etapa.
12. Pruebas generales del sistema en esta etapa se procedió a la prueba de la actuación de la placa de control y la placa de adecuación en conjunto en función de las especificaciones del sistema. Las pruebas fueron realizadas tanto en simulación como en modo real. Los resultados obtenidos fueron satisfactorios dándose por terminada el desarrollo.

PLACA DE ADECUACIÓN

La placa de adecuación es la encargada de manejar el control de electroválvulas y los motores del respirómetro. Esta placa implementa un sistema de control que tiene dos modos de funcionamiento. Para el caso de las electroválvulas se implementa un control on-off y para el caso de los motores se hace un control de velocidad manejando la potencia entregada por semiciclo cuyo principio, lo podemos explicar utilizando la gráfica siguiente:

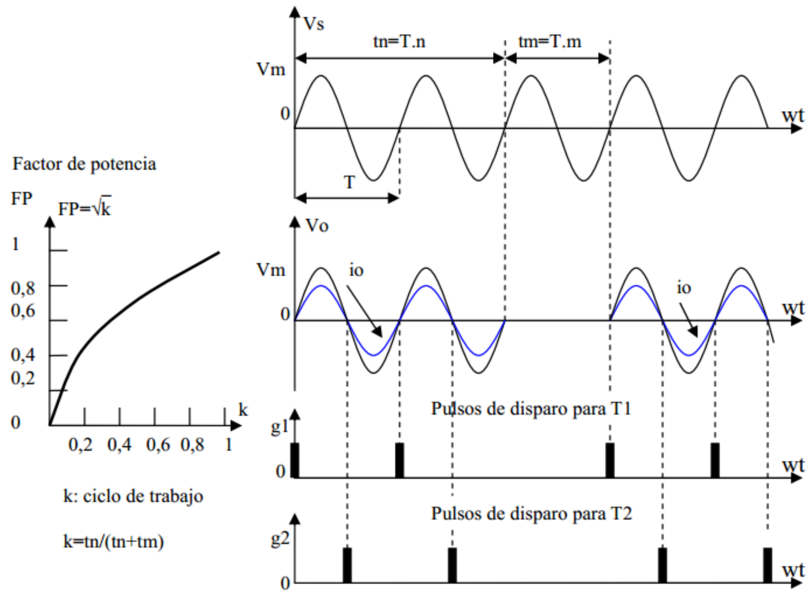


Figura n° 2

En figura n°2 se observa que los TRIAC conectan la carga durante un tiempo "t_n" y desconectan la carga, durante un tiempo "t_m".
El tiempo de activación "t_n" consiste en una cantidad entera de semiciclos. Los tiristores se activan en "sincronismo" con los cruces por cero del voltaje de entrada de corriente alterna. Este tipo de control se usa en aplicaciones con una gran inercia mecánica y alta constante de tiempo térmico (control de velocidad de motores y calefacción industrial).
Debido a la conmutación a voltaje y corriente cero de los tiristores, las armónicas generadas por la conmutación, se reducen notablemente. Para un voltaje senoidal de entrada dada por:

$$V_S = V_m \times \sin \omega t = \sqrt{2} \times V_S \times \sin \omega t$$

Si la placa de control conecta la carga una cantidad de "n" ciclos y desconexión de “m” ciclos, el voltaje eficaz (rms) sobre la carga lo podemos determinar cómo:

$$V_0 = \left[\frac{n}{2\pi} \times (n+m) \int_0^{2\pi} 2 \times V_S^2 \times \sin \omega t^2 \times d(\omega t)\right]$$

$$V_0 = V_S \times \sqrt{\frac{n}{(n+m)}} = V_S \times \sqrt{K}$$

El valor de k se llama “ciclo de trabajo”. En función de la variación del ciclo de trabajo se establece un consumo de energía para cada una de las cargas preciso y estable.
Que es el objetivo central del sistema, los tanques de agua incorporan además un sistema de medición de temperatura para determinar el punto de corte de la energía hacia sus resistencias.

El control de la energía entregada a la carga es realizado mediante un optotriac MOC3040. Estos dispositivos entregan la corriente necesaria para disparar los tiristores de alta tensión y entregan una aislación eléctrica entre el circuito de control y la línea de energía eléctrica de 7,5 kV. Al tener integrado un detector de cruce por 0 se elimina los picos de corriente de conmutación y se disminuye la generación de ruido EMI hacia la línea.

Por otro lado, la alta inmunidad a los transitorios de 5000V/µS combinado con sus características de baja capacidad de acoplamiento de salida, alta resistencia de aislación y hasta 800V de V_{DRM} hace a este dispositivo el enlace ideal entre el circuito de control y las cargas a conmutar en la línea de energía eléctrica.

Circuito integrado de detección

El circuito integrado de detección de cruce por cero, consiste en un diodo emisor de luz infrarrojo que activa ópticamente la etapa de detección.

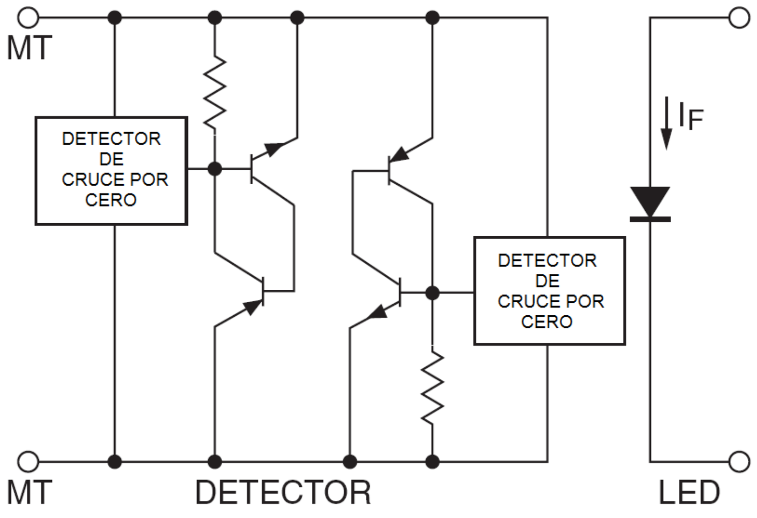


Figura n° 3 – Representación esquemática circuito de control TRIAC

La Figura n° 3 muestra por una representación esquemática del circuito de control del TRIAC, Todos los elementos se encuentran en un pequeño encapsulado DIP de 6 pines que asegura la integridad mecánica y protección contra las impurezas externas.

Los chips están aislados a través del medio de transmisión infrarroja que aísla de forma fiable la entrada del LED para poder controlarlos circuitos de carga alimentados por CA. Este sistema de aislamiento cumple con los estrictos requisitos establecidos por las agencias reguladoras tales como UL y VDE.

Un esquema simplificado del circuito integrado se muestra en la figura donde se puede ver que está aislado ópticamente. Este modelo es suficiente para describir todas las características importantes de funcionamiento del mismo.

Cuando el led se encuentra polarizado en forma directa emite radiación infrarroja que hace que se dispare el detector. Esta corriente de disparo a través del LED denominada $I_{FT,}$ es la corriente máxima que garantiza que el TRIAC pase al estado de conducción y está en el rango de 5 mA para el MOC3063 a 15 mA para el MOC3061. La caída de tensión en el LED con una corriente directa de I_F = 30 mA es de 1,5 V como máximo, la curva tensión / corriente del triac se muestra a continuación:

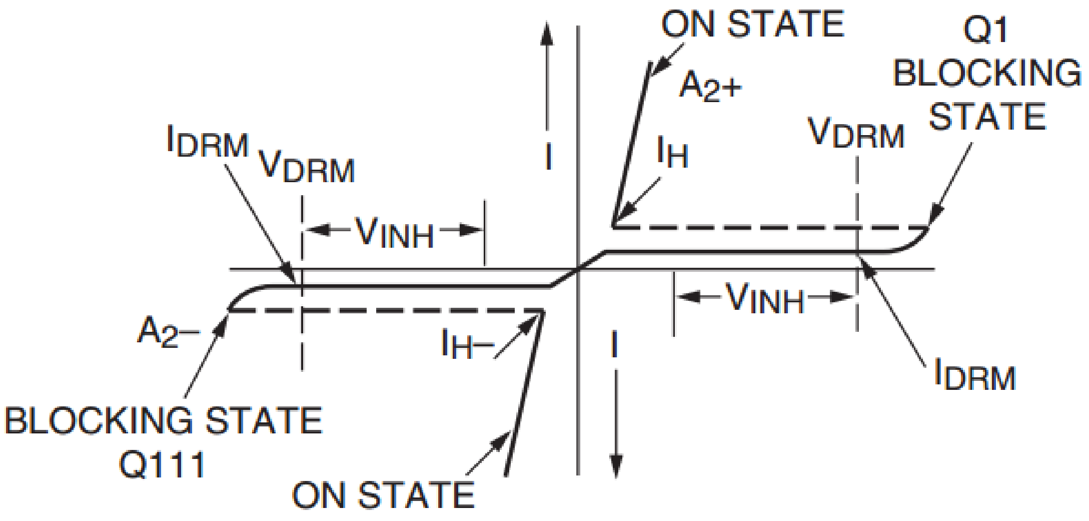
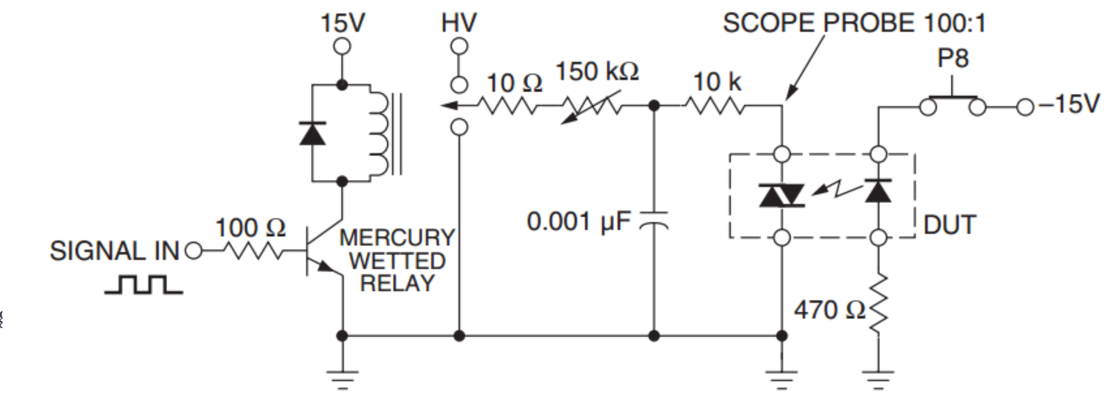


Figura n° 4

Como se ve en la gráfica figura n°4 una vez activado el TRIAC este permanece en ese estado hasta que la corriente a través del detector cae por debajo del valor de la corriente I_H, Que se encuentra en el orden de los 100 µA. En este momento, el detector conmuta el TRIAC al estado off y el mismo solo podrá ser activado por la corriente $I_{FT.}$

Circuito Básico de disparo

Suponiendo que el circuito mostrado a continuación está en estado de bloqueo o estado "apagado" (lo que significa que SI es cero), la tensión total de la línea de energía eléctrica aparece en terminales principales tanto del TRIAC como del controlador TRIAC.



Cuando la corriente de LED I_{FT} es suficiente y la tensión de la línea de energía eléctrica está por debajo de la tensión de inhibición V_{INH} el TRIAC es disparado pasando al estado "on".

Este circuito figura n°5 es utilizado como base en la implementación de la placa de control como se ve en los esquemas eléctricos que se encuentran a continuación.

CONTROL ELECTROVALVULA I

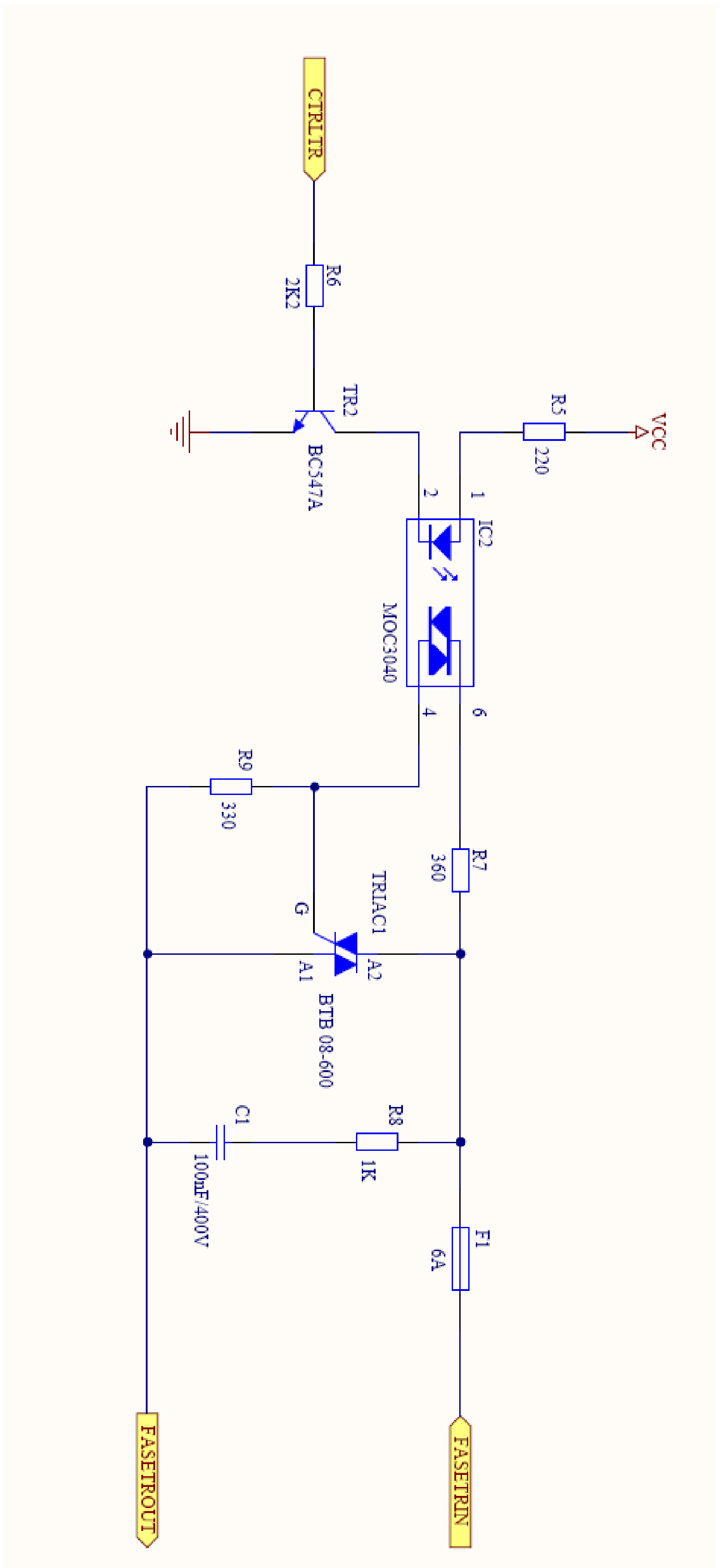


Figura n° 6 – Control Electroválvula I

CONTROL ELECTROVALVULA II

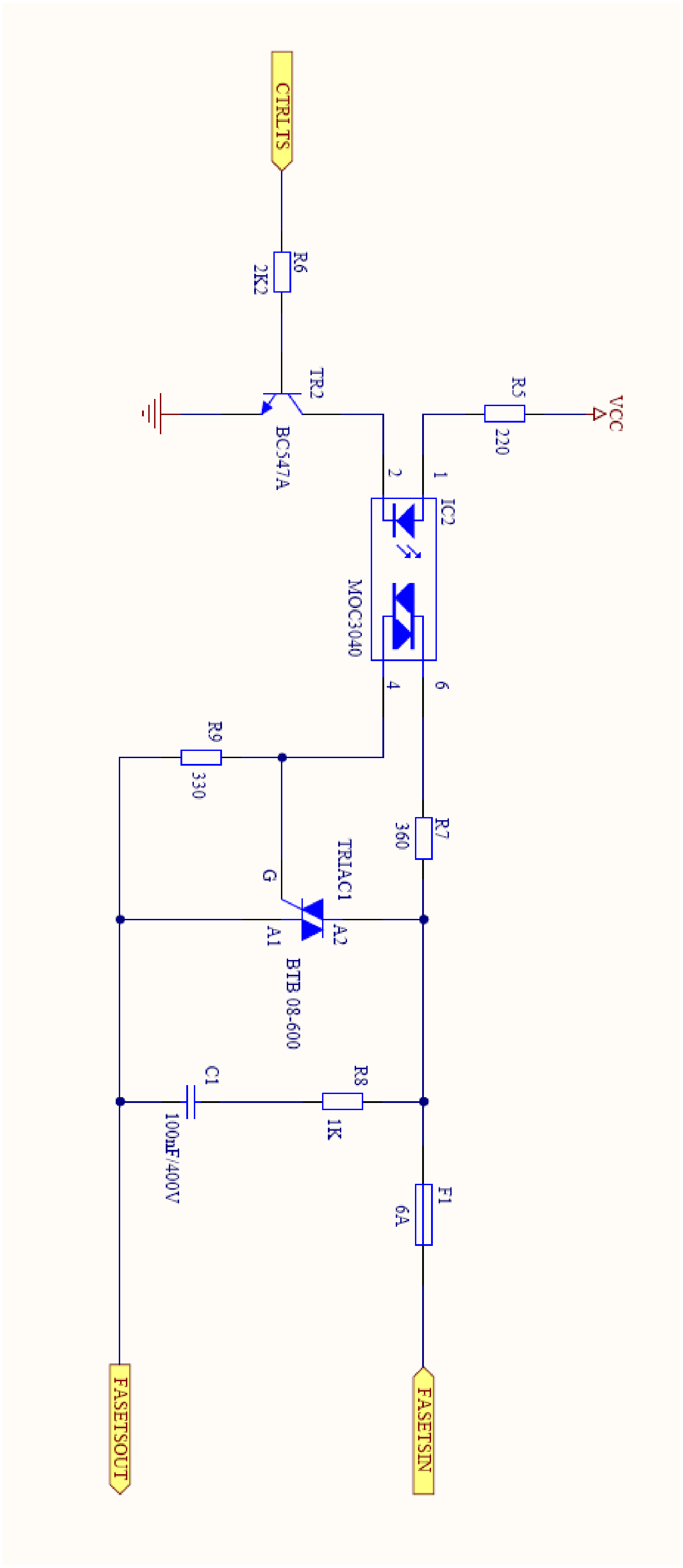


Figura n° 7 – Control Electroválvula II

CONTROL ELECTROVALVULA III

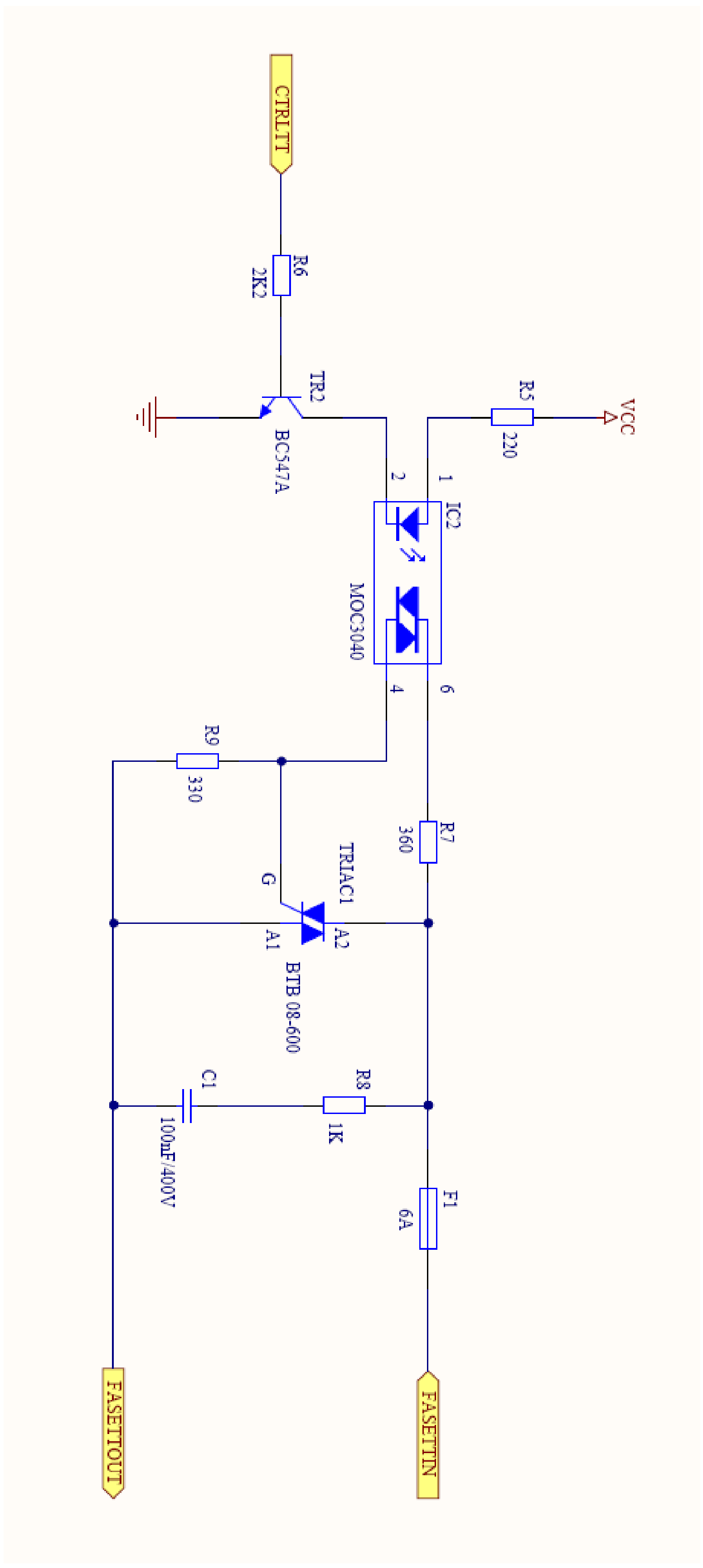


Figura n° 8 – Control Electroválvula III

CONTROL AGITADOR I

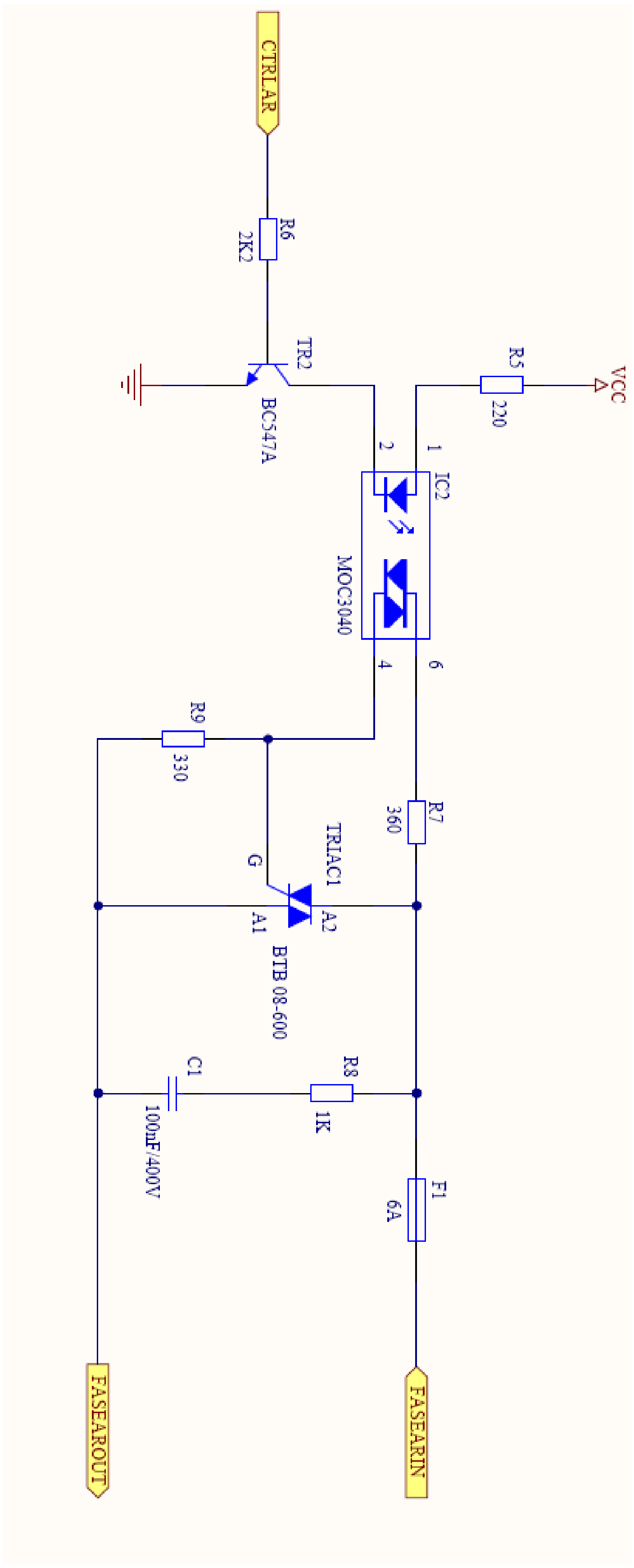


Figura n° 9 – Control Agitador I

CONTROL AGITADOR II

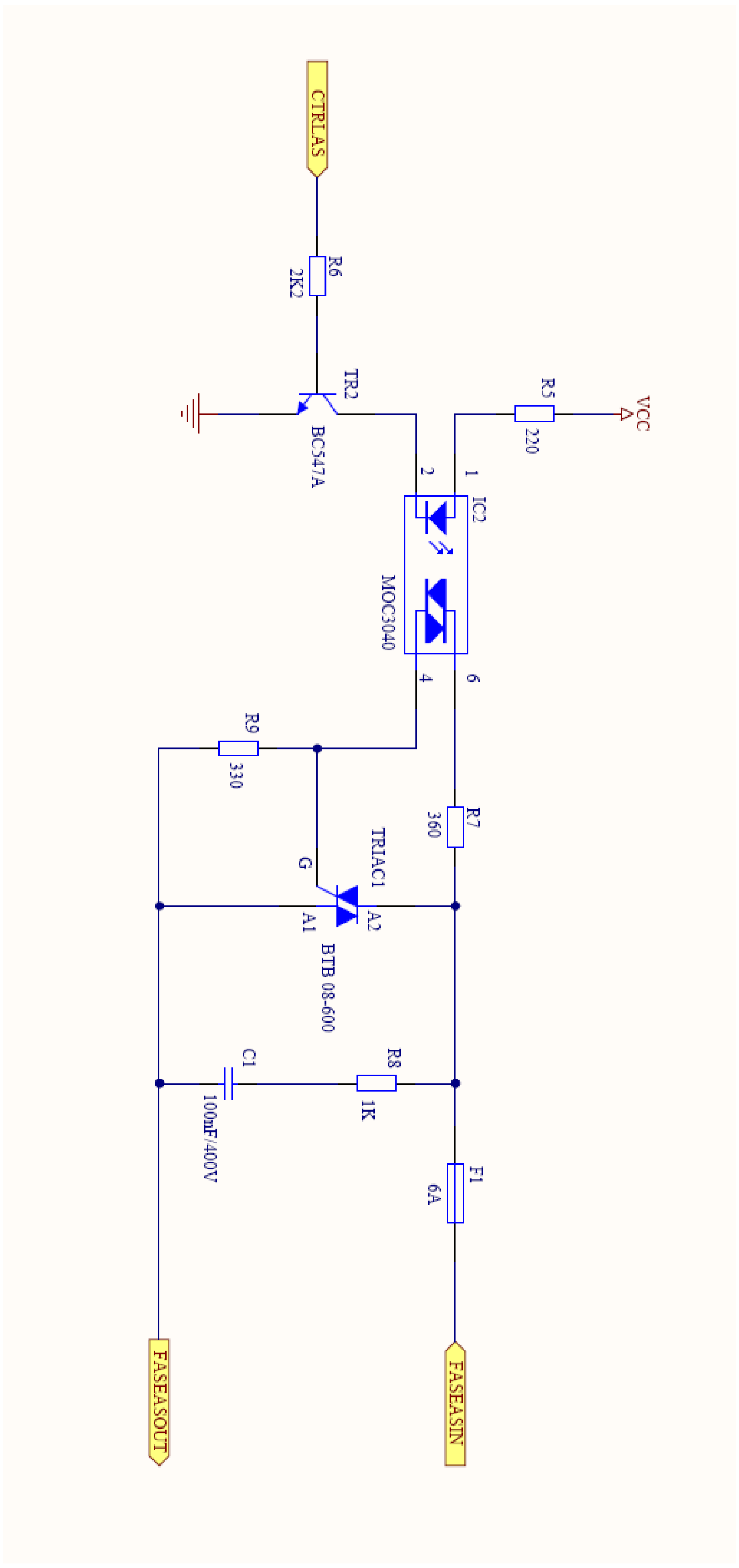


Figura n° 10 – Control Agitador II

CONTROL AIREADOR

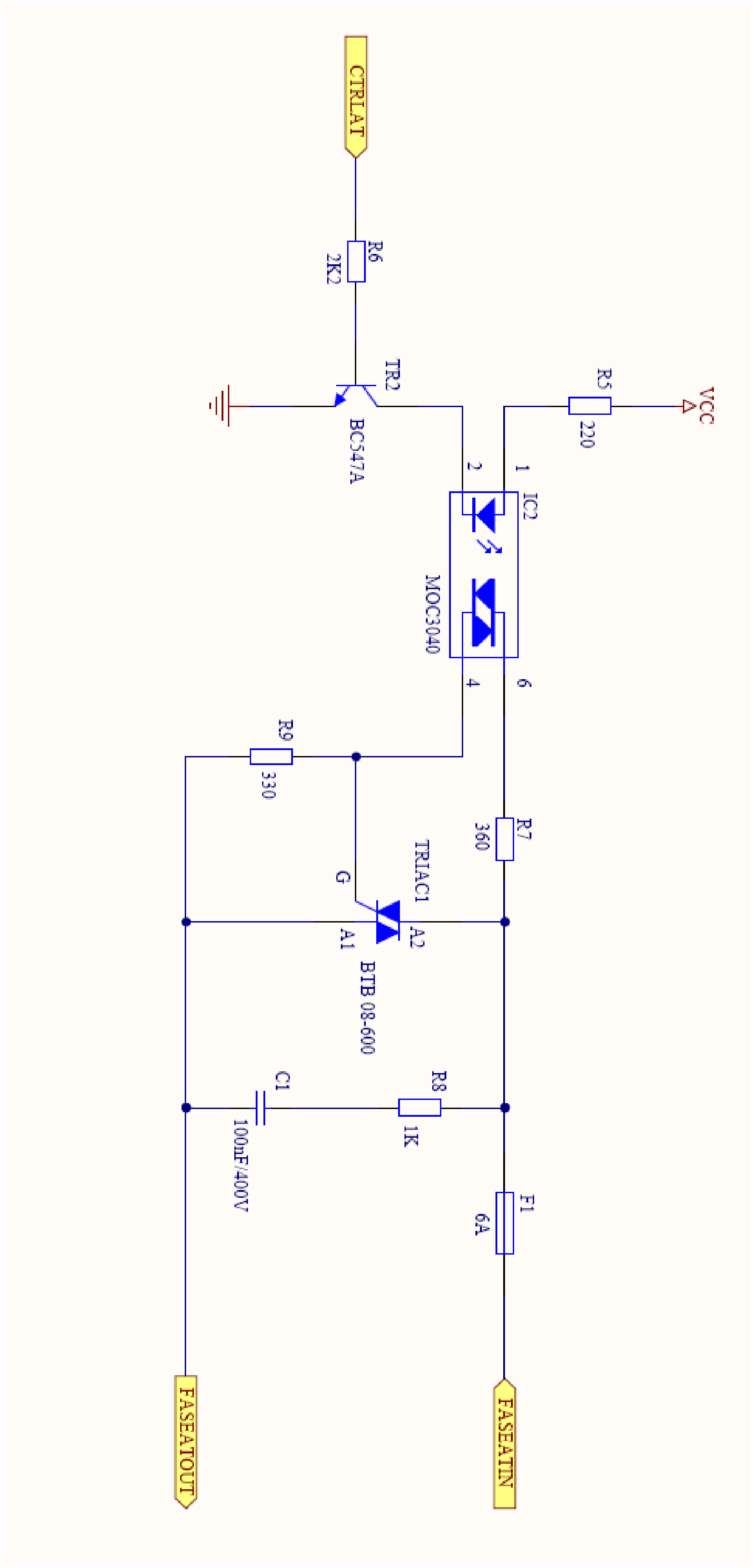


Figura n° 11 – Control Aireador

Placa de control

Para la implementación de la placa de control se ha procedido al desarrollo de un ASIC propio el cual se encarga de implementar la mayoría de las funciones del sistema con un mínimo de componentes externos.

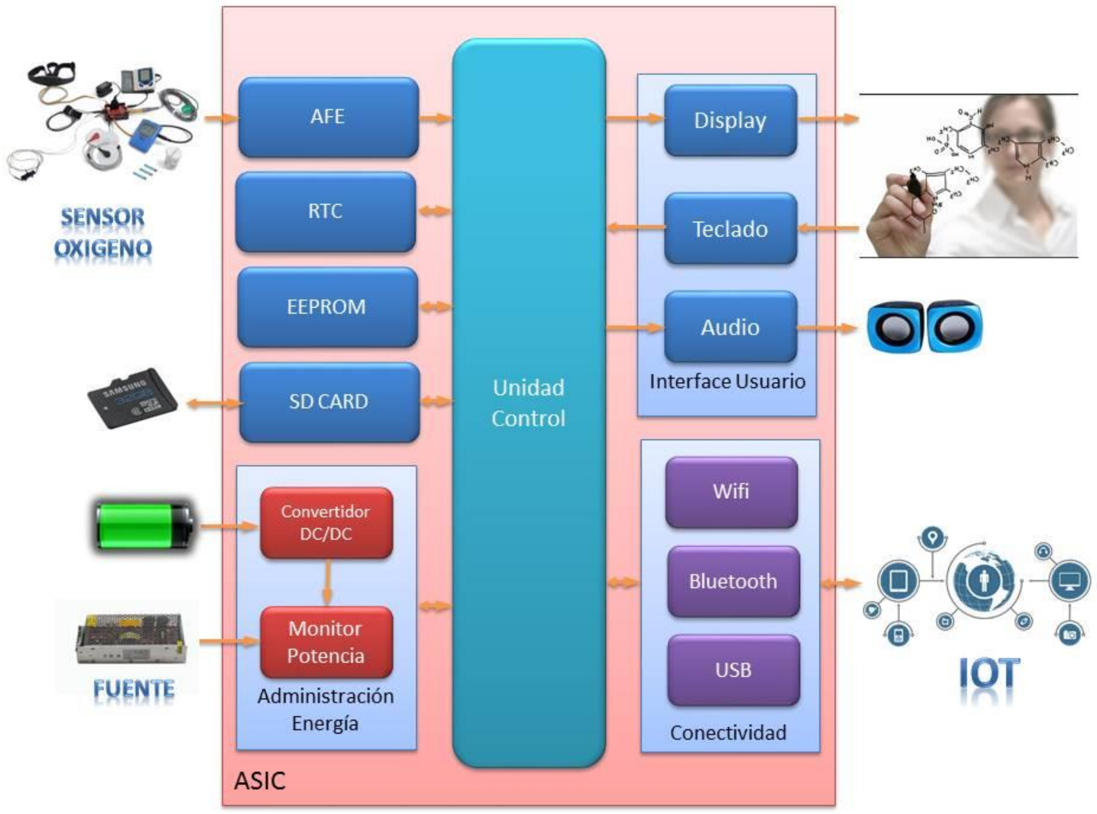


Figura n° 12 – Placa de Control ASIC

El integrado diseñado figura n° 12 está compuesto por los siguientes módulos:

Unidad de control

Este módulo se basa en un procesador ARM Cortex- M3 segmentado en tres etapas de 32 bits que se ejecuta a velocidades de hasta 80 MHz. El Cortex- M3 incluye un controlador totalmente integrado anidado con vectores de interrupción (CNTV) y varios módulos de depuración y traza. El subsistema de CPU en general incluye un controlador de DMA, caché flash y RAM. El NVIC ofrece baja latencia, las interrupciones anidadas, y la cola en cadena de interrupciones y otras características para aumentar la eficiencia de la gestión de interrupciones.

Display Driver

Este módulo proporciona la interfaz para un controlador de LCD y controlador de dispositivo gráfico. Estos dispositivos están integrados habitualmente en un panel LCD. La interfaz de estos dispositivos se conoce comúnmente como una interfaz i8080. Esta es una referencia al protocolo de interfaz de bus paralelo histórico del microprocesador Intel 8080.

- interfaz de bits 8 a gráfico controlador LCD

- Compatible con muchos dispositivos controladores de gráficos
- Realiza la lectura y escritura de transacción
- 2-255 ciclos de Read Ancho pulso bajo
- 1-255 ciclos de Read alto ancho de pulso
- Implementa la interfaz típica i8080

Driver Teclado

Este módulo utilizando un modulador sigma - delta es una forma versátil y eficiente para medir la capacitancia en aplicaciones tales como botones de sentido táctil, controles deslizantes, panel táctil y detección de proximidad, tiene las siguientes características:

- 6 sensores capacitivos Apoyo para las combinaciones definidas por el usuario de botón.
- Alta inmunidad al ruido de CA línea de energía, el ruido de EMC , y los cambios de tensión de alimentación .
- Controlador de teclado capaz de detectar pulsación de tecla en el modo de hibernación

Driver Audio

Este módulo ofrece una solución sencilla y rápida para añadir salida de audio digital para el diseño. Se da formato a los datos de audio de entrada y metadatos para crear el flujo de bits apropiada S / PDIF para audio digital óptica o coaxial. Soporta audio intercalado y separada, tiene las siguientes características:

- Cumple con la norma IEC - 60958 , AES / EBU , normas AES3 para la transmisión de audio PCM lineal
- Apoyo a los tipos de muestra para el reloj / 128 (hasta 192 kHz)
- Longitud de la muestra de audio configurable (8/16/24)
- FIFO Independiente de la izquierda y la derecha o FIFO del canal estéreo entrelazados

Conectividad

La conectividad está dada por tres módulos WIFI, Bluetooth y USB esto permite conectarse a cualquier red de tipo IoT disponible y transferir a posteriori los datos a la nube.

Driver SD card

Este módulo proporciona una interfaz para tarjetas SD formateadas con un sistema de archivos FAT. La especificación de la tarjeta SD incluye múltiples opciones de interfaz de hardware para la comunicación con una tarjeta SD. Este componente utiliza el método de interfaz SPI para la comunicación. Hasta cuatro interfaces SPI independientes se pueden

utilizar para la comunicación con una tarjeta SD cada uno. Ambos formatos FAT 32 / 16 son compatibles.
• Hasta cuatro tarjetas Secure Digital (SD) en modo SPI
• Formato FAT 32 / 16

Convertidor DC/DC

Este módulo permite tensiones de entrada que son inferiores a la tensión de funcionamiento del microcontrolador adecuando dicho nivel.
El convertidor utiliza un inductor externo para convertir la tensión de entrada a la tensión de salida deseada. Este está activado por defecto en el arranque del chip con una tensión de salida de 1,9 V. Esto permite el arranque en situaciones donde el voltaje de entrada al circuito integrado está por debajo de la tensión mínima permisible para alimentar el chip.

- Produce una tensión de salida seleccionable que es más alta que la tensión de entrada
- Tensión de entrada entre 0,5 V y 3,6 V
- Rango de voltaje de salida incrementada entre 1,8 V y 5,25 V
- Fuente de hasta 75 mA en función de los valores de los parámetros de entrada y voltaje de salida seleccionado
- Dos modos de funcionamiento : Activo y sleep

Real Time Clock

El componente de reloj de tiempo real (RTC) proporciona la hora exacta y la fecha para el sistema. La hora y la fecha se actualizan cada segundo basado en un pulso por segundo de interrupción de un cristal de 32.768 kHz.

La precisión del reloj se basa en el cristal disponible y es típicamente 20 ppm. El RTC permite obtener segundo, minuto, hora, día de la semana, día del mes, día del año, mes y año, de inicio y fin.

La alarmas facilitan la detección de eventos por segundo, minuto, hora , día de la semana , día del mes , día del año, el mes y el año. Las características principales son:
• Múltiples opciones de alarma
• Múltiples opciones de desbordamiento
• La opción de horario de verano (DST)

Esquema de pines:

La figura muestra el esquema de pines del ASIC diseñado, así como su correspondiente encapsulado. Este ASIC ha sido montado en un PSOC de la empresa Cypress.

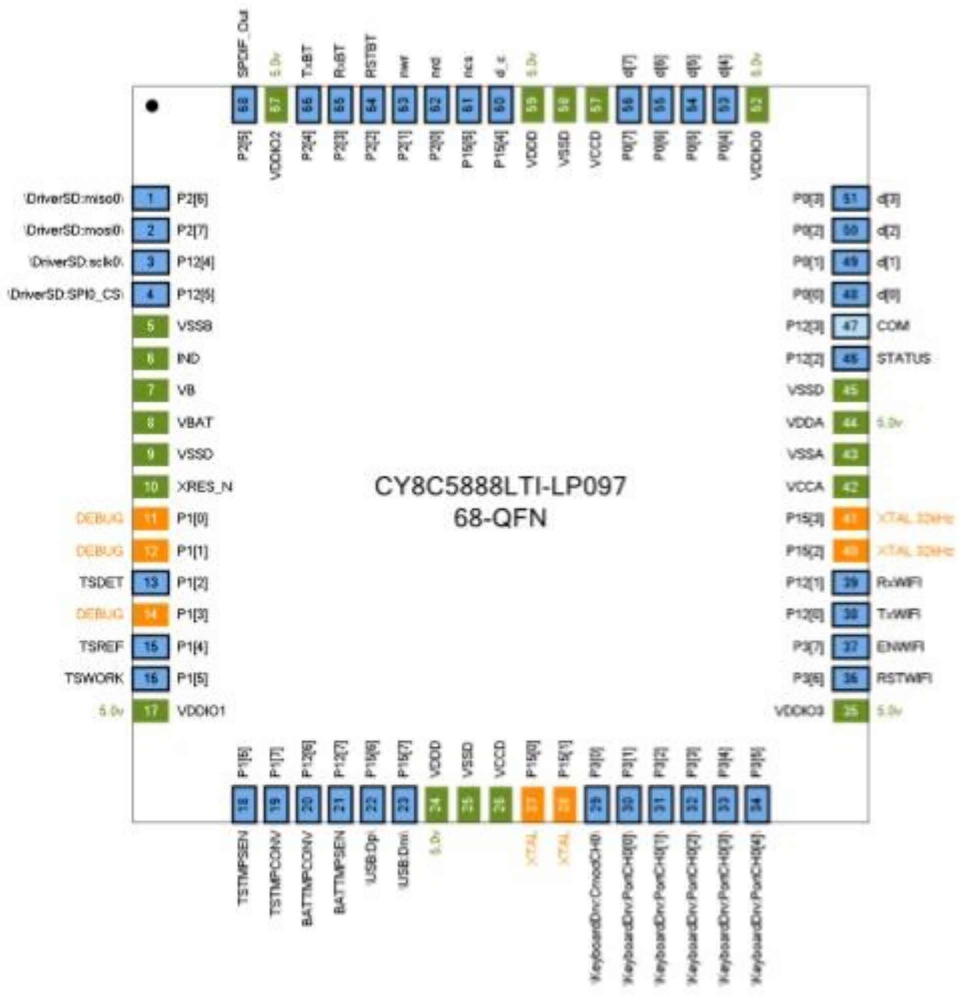


Figura n° 13 – Esquema pines ASIC

```
/*******************************************************************************
*****
* File Name: ADC.c
* Version 3.20
*
* Description:
*  This file provides the source code to the API for the Delta-Sigma ADC
*  Component.
*
* Note:
*
********************************************************************************
*****
* Copyright 2008-2015, Cypress Semiconductor Corporation.  All rights
reserved.
* You may use this file only in accordance with the license, terms,
conditions,
* disclaimers, and limitations in the end user license agreement
accompanying
* the software package with which this file was provided.
********************************************************************************
****/

#include "ADC.h"

#if(ADC_DEFAULT_INTERNAL_CLK)
    #include "ADC_theACLK.h"
#endif /* ADC_DEFAULT_INTERNAL_CLK */

#include "ADC_Ext_CP_Clk.h"

#if(ADC_DEFAULT_INPUT_MODE)
    #include "ADC_AMux.h"
#endif /* ADC_DEFAULT_INPUT_MODE */


/***************************************
* Global data allocation
***************************************/

/* Software flag for checking conversion completed or not */
volatile uint8 ADC_convDone = 0u;

/* Software flag to stop conversion for single sample conversion mode
*   with resolution above 16 bits
*/
volatile uint8 ADC_stopConversion = 0u;

/* To run the initialization block only at the start up */
uint8 ADC_initVar = 0u;

/* To check whether ADC started or not before switching the configuration
*/
volatile uint8 ADC_started = 0u;

/* Flag to hold ADC config number. By default active config is 1. */
volatile uint8 ADC_Config = 1u;

volatile int32 ADC_Offset;
volatile int32 ADC_CountsPerVolt;


/***************************************
* Local data allocation
***************************************/

/* The array with precalculated gain compensation coefficients */
static ADC_GCOR_STRUCT ADC_gcor[ADC_DEFAULT_NUM_CONFIGS];



/***************************************
* Forward function references
***************************************/
static void ADC_InitConfig(uint8 config) ;
static  void  ADC_GainCompensation(uint8  inputRange,  uint16  idealDecGain,
uint16 idealOddDecGain,
                            uint8 resolution, uint8 config) ;
static void ADC_SetDSMRef0Reg(uint8 value) ;


/*****************************************************************************
****
* Function Name: ADC_Init
*****************************************************************************
****
*
* Summary:
*  Initialize component's parameters to the parameters set by user in the
*  customizer of the component placed onto schematic. Usually called in
* ADC_Start().
*
*
* Parameters:
*  None
*
* Return:
*  None
*
*****************************************************************************
****/
void ADC_Init(void)
{

    ADC_Config = 1u;
    ADC_convDone = 0u;

    ADC_Ext_CP_Clk_SetMode(CYCLK_DUTY);

    /* This is only valid if there is an internal clock */
    #if(ADC_DEFAULT_INTERNAL_CLK)
        ADC_theACLK_SetMode(CYCLK_DUTY);
    #endif /* ADC_DEFAULT_INTERNAL_CLK */

    #if(ADC_IRQ_REMOVE == 0u)
        /* Set interrupt priority */
        CyIntSetPriority(ADC_INTC_NUMBER, ADC_INTC_PRIOR_NUMBER);
    #endif   /* End ADC_IRQ_REMOVE */

    /* Init static registers with common configuration */
    ADC_DSM_DEM0_REG     = ADC_CFG1_DSM_DEM0;
    ADC_DSM_DEM1_REG     = ADC_CFG1_DSM_DEM1;
    ADC_DSM_MISC_REG     = ADC_CFG1_DSM_MISC;
    ADC_DSM_CLK_REG     |= ADC_CFG1_DSM_CLK;
    ADC_DSM_REF1_REG     = ADC_CFG1_DSM_REF1;

    ADC_DSM_OUT0_REG     = ADC_CFG1_DSM_OUT0;
    ADC_DSM_OUT1_REG     = ADC_CFG1_DSM_OUT1;

    ADC_DSM_CR0_REG      = ADC_CFG1_DSM_CR0;
    ADC_DSM_CR1_REG      = ADC_CFG1_DSM_CR1;
#if(ADC_MI_ENABLE != 0u) /* Enable Modulator Input */
    ADC_DSM_CR3_REG      |= ADC_DSM_MODBIT_EN;
#else
    ADC_DSM_CR3_REG      = ADC_CFG1_DSM_CR3;
#endif /* ADC_MI_ENABLE != 0u*/
    ADC_DSM_CR8_REG      = ADC_CFG1_DSM_CR8;
    ADC_DSM_CR9_REG      = ADC_CFG1_DSM_CR9;
    ADC_DSM_CR13_REG     = ADC_CFG1_DSM_CR13;

    ADC_DEC_SR_REG       = ADC_CFG1_DEC_SR;

    /* Calculate Gain compensation coefficients for all configurations */
    ADC_GainCompensation(ADC_CFG1_INPUT_RANGE,
                                  ADC_CFG1_IDEAL_DEC_GAIN,
                                  ADC_CFG1_IDEAL_ODDDEC_GAIN,
                                  ADC_CFG1_RESOLUTION,
                                  ADC_CFG1);
    #if(ADC_DEFAULT_NUM_CONFIGS > 1)
        ADC_GainCompensation(ADC_CFG2_INPUT_RANGE,
                                  ADC_CFG2_IDEAL_DEC_GAIN,
                                  ADC_CFG2_IDEAL_ODDDEC_GAIN,
                                  ADC_CFG2_RESOLUTION,
                                  ADC_CFG2);
    #endif /* ADC_DEFAULT_NUM_CONFIGS > 1 */
    #if(ADC_DEFAULT_NUM_CONFIGS > 2)
        ADC_GainCompensation(ADC_CFG3_INPUT_RANGE,
                                  ADC_CFG3_IDEAL_DEC_GAIN,
                                  ADC_CFG3_IDEAL_ODDDEC_GAIN,
                                  ADC_CFG3_RESOLUTION,
                                  ADC_CFG3);
    #endif /* ADC_DEFAULT_NUM_CONFIGS > 2 */
    #if(ADC_DEFAULT_NUM_CONFIGS > 3)
        ADC_GainCompensation(ADC_CFG4_INPUT_RANGE,
                                  ADC_CFG4_IDEAL_DEC_GAIN,
                                  ADC_CFG4_IDEAL_ODDDEC_GAIN,
                                  ADC_CFG4_RESOLUTION,
                                  ADC_CFG4);
    #endif /* ADC_DEFAULT_NUM_CONFIGS > 3 */

    /* Set GCOR register for config1 */
    ADC_DEC_GVAL_REG = ADC_gcor[ADC_Config - 1u].gval;
    CY_SET_REG16(ADC_DEC_GCOR_16B_PTR, ADC_gcor[ADC_Config - 1u].gcor);

    /*  Initialize  the  registers  with  default  customizer  settings  for
config1 */
    ADC_InitConfig(ADC_Config);
}
```

```
/***************************************************************************
****
* Function Name: ADC_Enable
****************************************************************************
****
*
* Summary:
*  Enables the ADC DelSig block operation.
*
*
* Parameters:
*  None
*
* Return:
*  None
*
****************************************************************************
****/
void ADC_Enable(void)
{
      uint8 config;
    uint8 enableInterrupts;
    enableInterrupts = CyEnterCriticalSection();

    /* Read volatile variable to the local variable */
    config = ADC_Config;

    /* Enable active mode power for ADC */
    ADC_PWRMGR_DEC_REG |= ADC_ACT_PWR_DEC_EN;
    ADC_PWRMGR_DSM_REG |= ADC_ACT_PWR_DSM_EN;

     /* Enable alternative active mode power for ADC */
    ADC_STBY_PWRMGR_DEC_REG |= ADC_STBY_PWR_DEC_EN;
    ADC_STBY_PWRMGR_DSM_REG |= ADC_STBY_PWR_DSM_EN;

    /* Disable PRES, Enable power to VCMBUF0, REFBUF0 and REFBUF1, enable
PRES */
    ADC_RESET_CR4_REG |= ADC_IGNORE_PRESA1;
    ADC_RESET_CR5_REG |= ADC_IGNORE_PRESA2;

    ADC_DSM_CR17_REG |= (ADC_DSM_EN_BUF_VREF | ADC_DSM_EN_BUF_VCM);

    /* Code to disable the REFBUF0 if reference chosen is External ref */
    #if (((ADC_CFG1_REFERENCE == ADC_EXT_REF_ON_P03) || \
          (ADC_CFG1_REFERENCE == ADC_EXT_REF_ON_P32)) || \
         ((ADC_DEFAULT_NUM_CONFIGS > 1) && \
          ((ADC_CFG2_REFERENCE == ADC_EXT_REF_ON_P03) ||  \
           (ADC_CFG2_REFERENCE == ADC_EXT_REF_ON_P32))) || \
         ((ADC_DEFAULT_NUM_CONFIGS > 2) && \
          ((ADC_CFG3_REFERENCE == ADC_EXT_REF_ON_P03) ||  \
           (ADC_CFG3_REFERENCE == ADC_EXT_REF_ON_P32))) || \
         ((ADC_DEFAULT_NUM_CONFIGS > 3) && \
          ((ADC_CFG4_REFERENCE == ADC_EXT_REF_ON_P03) || \
           (ADC_CFG4_REFERENCE == ADC_EXT_REF_ON_P32))))
        if (((config == 1u) &&
            ((ADC_CFG1_REFERENCE == ADC_EXT_REF_ON_P03) ||
             (ADC_CFG1_REFERENCE == ADC_EXT_REF_ON_P32))) ||
            ((config == 2u) &&
            ((ADC_CFG2_REFERENCE == ADC_EXT_REF_ON_P03) ||
             (ADC_CFG2_REFERENCE == ADC_EXT_REF_ON_P32))) ||
            ((config == 3u) &&
            ((ADC_CFG3_REFERENCE == ADC_EXT_REF_ON_P03) ||
             (ADC_CFG3_REFERENCE == ADC_EXT_REF_ON_P32))) ||
            ((config == 4u) &&
            ((ADC_CFG4_REFERENCE == ADC_EXT_REF_ON_P03) ||
             (ADC_CFG4_REFERENCE == ADC_EXT_REF_ON_P32))))
        {
            /* Disable the REFBUF0 */
            ADC_DSM_CR17_REG &= (uint8)~ADC_DSM_EN_BUF_VREF;
        }
    #endif /* External ref */

    #if (((ADC_CFG1_INPUT_RANGE == ADC_IR_VSSA_TO_2VREF) || \
         ((ADC_DEFAULT_NUM_CONFIGS > 1) && \
          (ADC_CFG2_INPUT_RANGE == ADC_IR_VSSA_TO_2VREF)) || \
         ((ADC_DEFAULT_NUM_CONFIGS > 2) && \
          (ADC_CFG3_INPUT_RANGE == ADC_IR_VSSA_TO_2VREF)) || \
         ((ADC_DEFAULT_NUM_CONFIGS > 3) && \
          (ADC_CFG4_INPUT_RANGE == ADC_IR_VSSA_TO_2VREF))))
        if(((config == 1u) &&
            (ADC_CFG1_INPUT_RANGE == ADC_IR_VSSA_TO_2VREF) &&
            ((ADC_CFG1_REFERENCE != ADC_EXT_REF_ON_P03) &&
             (ADC_CFG1_REFERENCE != ADC_EXT_REF_ON_P32))) ||
             ((config == 2u) &&
              (ADC_CFG2_INPUT_RANGE == ADC_IR_VSSA_TO_2VREF) &&
             ((ADC_CFG2_REFERENCE != ADC_EXT_REF_ON_P03) &&
             (ADC_CFG2_REFERENCE != ADC_EXT_REF_ON_P32))) ||
             ((config == 3u) &&
              (ADC_CFG3_INPUT_RANGE == ADC_IR_VSSA_TO_2VREF) &&
             ((ADC_CFG3_REFERENCE != ADC_EXT_REF_ON_P03) &&
             (ADC_CFG3_REFERENCE != ADC_EXT_REF_ON_P32))) ||
             ((config == 4u) &&
              (ADC_CFG4_INPUT_RANGE == ADC_IR_VSSA_TO_2VREF) &&
             ((ADC_CFG4_REFERENCE != ADC_EXT_REF_ON_P03) &&
             (ADC_CFG4_REFERENCE != ADC_EXT_REF_ON_P32))))
        {
            /* Enable the REFBUF1 */
            ADC_DSM_REF0_REG |= ADC_DSM_EN_BUF_VREF_INN;
        }
    #endif /* VSSA_TO_2VREF */
    if(config != 0u)
    {
        /* Suppress compiler warning */
    }

    /* Wait for 3 microseconds */
    CyDelayUs(ADC_PRES_DELAY_TIME);

    /* Enable the press circuit */
    ADC_RESET_CR4_REG &= (uint8)~ADC_IGNORE_PRESA1;
    ADC_RESET_CR5_REG &= (uint8)~ADC_IGNORE_PRESA2;

    /* Enable negative pumps for DSM  */
    ADC_PUMP_CR1_REG  |= ( ADC_PUMP_CR1_CLKSEL | ADC_PUMP_CR1_FORCE );

    /* Enable Modulator Chopping if required */
    ADC_DSM_CR2_REG = ADC_CFG1_DSM_CR2;

    /* This is only valid if there is an internal clock */
    #if(ADC_DEFAULT_INTERNAL_CLK)
        ADC_PWRMGR_CLK_REG |= ADC_ACT_PWR_CLK_EN;
        ADC_STBY_PWRMGR_CLK_REG |= ADC_STBY_PWR_CLK_EN;
    #endif /* ADC_DEFAULT_INTERNAL_CLK */

    /* Enable the active and alternate active power for charge pump clock
*/
    ADC_PWRMGR_CHARGE_PUMP_CLK_REG |= ADC_ACT_PWR_CHARGE_PUMP_CLK_EN;
    ADC_STBY_PWRMGR_CHARGE_PUMP_CLK_REG |= ADC_STBY_PWR_CHARGE_PUMP_CLK_EN;

    #if(ADC_IRQ_REMOVE == 0u)
        /* Clear a pending interrupt */
        CyIntClearPending(ADC_INTC_NUMBER);
        /* Enable interrupt */
        CyIntEnable(ADC_INTC_NUMBER);
    #endif   /* End ADC_IRQ_REMOVE */

    CyExitCriticalSection(enableInterrupts);

}


/************************************************************************
*****
* Function Name: ADC_Start
*************************************************************************
*****
*
* Summary:
*  Performs all required initialization for this component and enables
*  the power. It configure all the register the first time it is called.
*  Subsequent calls of the Start function only enable the ADC and turn
*  on the power. If multiple configurations are selected, it will
*  configure the ADC for configuration 1 by default, unless the
*  ADC_SelectConfiguration( ) function has been called to change
*  the default setting.
*
* Parameters:
*  None
*
* Return:
*  None
*
* Global variables:
*  ADC_initVar:  Used to check the initial configuration,
*  modified when this function is called for the first time.
*
*************************************************************************
****/
void ADC_Start(void)
{
    if(ADC_initVar == 0u)
    {
        if(ADC_started == 0u)
        {
            ADC_Init();
        }
        ADC_initVar = 1u;
    }

    /* Enable the ADC */
    ADC_Enable();
}
```

```
/***************************************************************************
*****
* Function Name: ADC_Stop
****************************************************************************
*****
*
* Summary:
*  This function stops and powers down the ADC component and the internal
*  clock if the external clock is not selected. If an external clock is
*  used, it is up to the designer to power down the external clock it
*  required.
*
* Parameters:
*  None
*
* Return:
*  None
*
****************************************************************************
****/
void ADC_Stop(void)
{
    uint8 enableInterrupts;
    enableInterrupts = CyEnterCriticalSection();

    /* Stop conversions */
    ADC_DEC_CR_REG &= (uint8)~ADC_DEC_START_CONV;
    ADC_DEC_SR_REG |=  ADC_DEC_INTR_CLEAR;

    /* Disable PRES, Disable power to VCMBUF0, REFBUF0 and REFBUF1,
       enable PRES */
    ADC_RESET_CR4_REG |= ADC_IGNORE_PRESA1;
    ADC_RESET_CR5_REG |= ADC_IGNORE_PRESA2;

    ADC_DSM_CR17_REG &= (uint8)~(ADC_DSM_EN_BUF_VREF | ADC_DSM_EN_BUF_VCM);
    ADC_DSM_REF0_REG &= (uint8)~ADC_DSM_EN_BUF_VREF_INN;

    /* Wait for 3 microseconds. */
    CyDelayUs(ADC_PRES_DELAY_TIME);

    /* Enable the press circuit */
    ADC_RESET_CR4_REG &= (uint8)~ADC_IGNORE_PRESA1;
    ADC_RESET_CR5_REG &= (uint8)~ADC_IGNORE_PRESA2;

    /* Disable power to the ADC */
    ADC_PWRMGR_DSM_REG &= (uint8)~ADC_ACT_PWR_DSM_EN;

    /* Disable power to Decimator block */
    ADC_PWRMGR_DEC_REG &= (uint8)~ADC_ACT_PWR_DEC_EN;

    /* Disable alternative active power to the ADC */
    ADC_STBY_PWRMGR_DEC_REG &= (uint8)~ADC_STBY_PWR_DEC_EN;
    ADC_STBY_PWRMGR_DSM_REG &= (uint8)~ADC_STBY_PWR_DSM_EN;

   /* Disable negative pumps for DSM  */
    ADC_PUMP_CR1_REG  &=  (uint8)~(ADC_PUMP_CR1_CLKSEL  |  ADC_PUMP_CR1_FORCE
);

    /* This is only valid if there is an internal clock */
    #if(ADC_DEFAULT_INTERNAL_CLK)
```

```
        ADC_PWRMGR_CLK_REG &= (uint8)~ADC_ACT_PWR_CLK_EN;
        ADC_STBY_PWRMGR_CLK_REG &= (uint8)~ADC_STBY_PWR_CLK_EN;
    #endif /* ADC_DEFAULT_INTERNAL_CLK */

    /* Disable Modulator Chopping */
    ADC_DSM_CR2_REG &= (uint8)~ADC_DSM_MOD_CHOP_EN;
    /* Disable power to charge pump clock */
    ADC_PWRMGR_CHARGE_PUMP_CLK_REG                                          &=
(uint8)~ADC_ACT_PWR_CHARGE_PUMP_CLK_EN;
    ADC_STBY_PWRMGR_CHARGE_PUMP_CLK_REG                                     &=
(uint8)~ADC_STBY_PWR_CHARGE_PUMP_CLK_EN;

    CyExitCriticalSection(enableInterrupts);
}


/***************************************************************************
*****
* Function Name: ADC_SetBufferGain
****************************************************************************
*****
*
* Summary:
*  Sets input buffer gain.
*
* Parameters:
*  gain:  Two bit value to select a gain of 1, 2, 4, or 8.
*
* Return:
*  None
*
****************************************************************************
****/
void ADC_SetBufferGain(uint8 gain)
{
    uint8 tmpReg;
    tmpReg = ADC_DSM_BUF1_REG & (uint8)~ADC_DSM_GAIN_MASK;
    tmpReg |= (uint8)(gain << ADC_DSM_GAIN_SHIFT) & ADC_DSM_GAIN_MASK;
    ADC_DSM_BUF1_REG = tmpReg;
}


/***************************************************************************
*****
* Function Name: ADC_SetCoherency
****************************************************************************
*****
*
* Summary:
*  This function allows the user to change which of the ADC's 3 word
*  result will trigger a coherency unlock. The ADC's result will not be
*  updated until the set byte is read either by the ADC or DMA.
*   By default the LSB is the coherency byte for right alignment data
format.
*  The middle or high byte is set automatically depend on left alignment
*  configuration for DMA data transfer.
*  If DMA or if a custom API requires different byte to be read the last,
*  this API should be used to set the last byte of the ADC result that is
read.
*  If a multibyte read is performed either by DMA or the ARM processor, the
*  coherency can be set to any byte in the last word read.
```

```
*
* Parameters:
*  coherency:  Two bit value to set the coherency bit.
*           00-Coherency checking off
*           01-low byte is key byte
*           02-middle byte is the key byte
*           03-high byte is the key byte
*
* Return:
*  None
*
***************************************************************************
****/
void ADC_SetCoherency(uint8 coherency)
{
    uint8 tmpReg;

    tmpReg = ADC_DEC_COHER_REG & (uint8)~ADC_DEC_SAMP_KEY_MASK;
    tmpReg |= coherency & ADC_DEC_SAMP_KEY_MASK;
    ADC_DEC_COHER_REG = tmpReg;
}


/**************************************************************************
*****
* Function Name: ADC_SetGCOR
***************************************************************************
*****
*
* Summary:
*  Calculates a new GCOR value and writes it into the GCOR register.
*  The GCOR value is a 16-bit value that represents a gain of 0 to 2.
*   The ADC result is multiplied by this value before it is placed in the
ADC
*  output registers. The numerical format for the GCOR value is:
*  0x0000 -> 0.000
*  0x8000 -> 1.000
*  0xFFFF -> 1.99997
*  When executing the function, the old GCOR value is multiplied by
*  gainAdjust and reloaded into the GCOR register.
*
* Parameters:
*  gainAdjust:  floating point value to set GCOR registers.
*
* Return:
*  uint8: 0 - if GCOR value is within the expected range.
*         1 - the correction value is outside GCOR value range of
*             0.00 to 1.9999.
*
* Side Effects:   The GVAL register is set to the amount of valid bits in
the
*                   GCOR  register minus one. If GVAL is 15 (0x0F), all 16
bits
*                  of the GCOR registers will be valid. If for example GVAL
is
*                   11 (0x0B) only 12 bits will be valid. The least 4 bits
will
*                   be lost when the GCOR value is shifted 4 places to the
right.
*
```

```
****************************************************************************
***/
uint8 ADC_SetGCOR(float32 gainAdjust)
{
    uint16 tmpReg;
    uint8 status;
    float32 tmpValue;

    tmpReg = ADC_gcor[ADC_Config - 1u].gcor;
    tmpValue = ((float32)tmpReg / (float32)ADC_IDEAL_GAIN_CONST);
    tmpValue = tmpValue * gainAdjust;

    if (tmpValue > 1.9999)
    {
        status = 1u;
    }
    else
    {
        tmpValue *= (float32)ADC_IDEAL_GAIN_CONST;
            tmpReg = (uint16)tmpValue;
        CY_SET_REG16(ADC_DEC_GCOR_16B_PTR, tmpReg);
        /* Update gain array to be used by SelectConfiguration() API */
       ADC_gcor[ADC_Config - 1u].gcor = tmpReg;

        status = 0u;

    }
    return(status);
}


/***************************************************************************
****
* Function Name: ADC_ReadGCOR
****************************************************************************
****
*
* Summary:
*   This API returns the current GCOR register value, normalized based on
the
*  GVAL register settings.
*  For example, if the GCOR value is 0x0812 and the GVAL register is set to
*   11 (0x0B) then the returned value will be shifted by for bits to the
left.
*   (Actual GCOR value = 0x0812, returned value = 0x8120)
*
* Parameters:
*  None
*
* Return:
*  uint16:  Normalized GCOR value.
*
****************************************************************************
****/
uint16 ADC_ReadGCOR(void)
{
    uint8 gValue;
    uint16 gcorValue;

    gValue = ADC_DEC_GVAL_REG;
    gcorValue = CY_GET_REG16(ADC_DEC_GCOR_16B_PTR);
    if (gValue < ADC_MAX_GVAL)
    {
        gcorValue <<= ADC_MAX_GVAL - gValue;
    }

    return gcorValue;
}


/*************************************************************************
*****
* Function Name: ADC_StartConvert
*************************************************************************
*****
*
* Summary:
*  Forces the ADC to initiate a conversion. If in the "Single Sample"
*  mode, one conversion will be performed then the ADC will halt. If in
*  one of the other three conversion modes, the ADC will run
*  continuously until the ADC_Stop() or ADC_StopConvert() is called.
*
* Parameters:
*  None
*
* Return:
*  None
*
*************************************************************************
****/
void ADC_StartConvert(void)
{
    /* Start the conversion */
    ADC_DEC_CR_REG |= ADC_DEC_START_CONV;
}


/*************************************************************************
*****
* Function Name: ADC_StopConvert
*************************************************************************
*****
*
* Summary:
*  Forces the ADC to stop all conversions. If the ADC is in the middle of a
*   conversion, the ADC will be reset and not provide a result for that
partial
*  conversion.
*
* Parameters:
*  None
*
* Return:
*  None
*
*************************************************************************
****/
void ADC_StopConvert(void)
{
    /* Stop all conversions */
    ADC_DEC_CR_REG &= (uint8)~ADC_DEC_START_CONV;
```

```
}


/***************************************************************************
*****
* Function Name: ADC_IsEndConversion
****************************************************************************
*****
*
* Summary:
*    Checks  the  status  that  the  most  recently  started  conversion  has
completed.
*  The status is cleared by any of ADC_GetResult8(), ADC_GetResult16() or
*  ADC_GetResult32() API.
*  This function provides the programmer with two options. In one mode this
*   function immediately returns with the conversion status. In the other
mode,
*    the  function  does  not  return  (blocking)  until  the  conversion  has
completed.
*
* Parameters:
*    retMode:  Check  conversion  return  mode.  See  the  following  table  for
options.
*   ADC_RETURN_STATUS -   Immediately returns conversion result
*                                        status.
*   ADC_WAIT_FOR_RESULT - Does not return until ADC conversion
*                                        is complete.
*
* Return:
*  If a nonzero value is returned, the last conversion has completed.
*   If  the  returned  value  is  zero,  the  ADC  is  still  calculating  the  last
result.
*
* Global variables:
*  ADC_convDone:  Used to check whether conversion is complete
*  or not for single sample mode with resolution is above 16
*
***************************************************************************
****/
uint8 ADC_IsEndConversion(uint8 retMode)
{
    uint8 status;

    do
    {
        /* Check for stop convert if conversion mode is Single Sample with
        *    resolution above 16 bit
        */
        if(ADC_stopConversion != 0u)
        {
            status = ADC_convDone;
        }
        else
        {
            status = ADC_DEC_SR_REG & ADC_DEC_CONV_DONE;
        }
    }while((status      !=      ADC_DEC_CONV_DONE)      &&      (retMode      ==
ADC_WAIT_FOR_RESULT));

    return(status);
}
```

```
/***************************************************************************
*****
* Function Name: ADC_GetResult8
****************************************************************************
*****
*
* Summary:
*  This function returns the result of an 8-bit conversion. If the
*  resolution is set greater than 8-bits, the LSB of the result will be
*  returned. When the ADC is configured for 8-bit single ended mode,
*  the ADC_GetResult16() function should be used instead. This
*  function returns only signed 8-bit values. The maximum positive
*  signed 8-bit value is 127, but in singled ended 8-bit mode, the
*  maximum positive value is 255.
*
* Parameters:
*  None
*
* Return:
*  int8: The LSB of the last ADC conversion.
*
* Global variables:
*  ADC_convDone:  Cleared in single sample mode with resolution
*                               above 16 bits
*
****************************************************************************
****/
int8 ADC_GetResult8( void )
{
    int8 result;
    uint8 coherency;

    /* Read active coherency configuration */
    coherency = ADC_DEC_COHER_REG & ADC_DEC_SAMP_KEY_MASK;

    result = (int8)ADC_DEC_SAMP_REG;

    if(coherency == ADC_DEC_SAMP_KEY_MID)
    {   /* Dummy read of the middle byte to unlock the coherency */
        (void)ADC_DEC_SAMPM_REG;
    }
    else  if(coherency == ADC_DEC_SAMP_KEY_HIGH)
    {   /* Dummy read of the MSB byte to unlock the coherency */
        (void)ADC_DEC_SAMPH_REG;
    }
    else /*No action required for other coherency */
    {
    }
    /*  Clear  conversion  complete  status  in  Single  Sample  mode  with
resolution above 16 bit */
    if(ADC_stopConversion != 0u)
    {
        ADC_convDone = 0u;
    }
    return (result);
}
```

```
/***************************************************************************
*****
* Function Name: ADC_GetResult16
****************************************************************************
*****
*
* Summary:
*  Returns a 16-bit result for a conversion with a result that has a
*  resolution of 8 to 16 bits. If the resolution is set greater than 16-
bits,
*  it will return the 16 least significant bits of the result. When the ADC
*  is configured for 16-bit single ended mode, the ADC_GetResult32()
*  function should be used instead. This function returns only signed
*  16-bit result, which allows a maximum positive value of 32767, not
65535.
*  This function supports different coherency settings.
*
* Parameters:
*   void
*
* Return:
*  int16:  ADC result.
*
* Global variables:
*  ADC_convDone:  Cleared in single sample mode with resolution
*                               above 16 bits
*
****************************************************************************
****/
int16 ADC_GetResult16(void)
{
    uint16 result;
    uint8 coherency;

    /* Read active coherency configuration */
    coherency = ADC_DEC_COHER_REG & ADC_DEC_SAMP_KEY_MASK;

    if(coherency <= ADC_DEC_SAMP_KEY_LOW)
    {   /*  Use default method to read result registers i.e. LSB byte read
at the end*/
        #if (CY_PSOC3)
            result = ADC_DEC_SAMPM_REG;
            result = (result << 8u) | ADC_DEC_SAMP_REG;
        #else
            result = (CY_GET_REG16(ADC_DEC_SAMP_16B_PTR));
        #endif /* CY_PSOC3 */
    }
    else /* MID or HIGH */
    {   /* Read middle byte at the end */
        #if (CY_PSOC3)
            result = (CY_GET_REG16(ADC_DEC_SAMP_16B_PTR));
        #else
            result = ADC_DEC_SAMP_REG;
            result |=  (uint16)((uint16)ADC_DEC_SAMPM_REG << 8u);
        #endif /* CY_PSOC3 */
        if(coherency == ADC_DEC_SAMP_KEY_HIGH)
        {   /* Dummy read of the MSB byte to unlock the coherency */
            (void)ADC_DEC_SAMPH_REG;
        }
    }
    /*  Clear  conversion  complete  status  in  Single  Sample  mode  with
resolution above 16 bit */
    if(ADC_stopConversion != 0u)
    {
        ADC_convDone = 0u;
    }

    return ((int16)result);
}


/***************************************************************************
*****
* Function Name: ADC_GetResult32
****************************************************************************
*****
*
* Summary:
*  Returns a 32-bit result for a conversion with a result that has a
*  resolution of 8 to 20 bits.
*  This function supports different coherency settings.
*
* Parameters:
*  None
*
* Return:
*  int32: Result of the last ADC conversion.
*
* Global variables:
*  ADC_convDone:  Cleared in single sample mode with resolution
*                              above 16 bits
*
****************************************************************************
****/
int32 ADC_GetResult32(void)
{
    uint32 result;
    uint8 coherency;
    #if (CY_PSOC3)
          uint16 tmp;
    #endif /* CY_PSOC3 */

    /* Read active coherency configuration */
    coherency = ADC_DEC_COHER_REG & ADC_DEC_SAMP_KEY_MASK;

    if(coherency <= ADC_DEC_SAMP_KEY_LOW)
    {    /*  Use default method to read result registers i.e. LSB byte read
at the end*/
        #if (CY_PSOC3)
            result = ADC_DEC_SAMPH_REG;
            if((result & 0x80u) != 0u)
            {    /* Sign extend */
                result |= 0xFF00u;
            }
            result = (result << 8u) | ADC_DEC_SAMPM_REG;
            result = (result << 8u) | ADC_DEC_SAMP_REG;
        #else
            result = CY_GET_REG16(ADC_DEC_SAMPH_16B_PTR);
            result         =         (result        <<        16u)        |
(CY_GET_REG16(ADC_DEC_SAMP_16B_PTR));
        #endif /* CY_PSOC3 */
    }
    else if(coherency == ADC_DEC_SAMP_KEY_MID)
    {   /* Read middle byte at the end */
        #if (CY_PSOC3)
            result = ADC_DEC_SAMPH_REG;
            if((result & 0x80u) != 0u)
            {   /* Sign extend */
                result |= 0xFF00u;
            }
            result          =          (result         <<         16u)          |
(CY_GET_REG16(ADC_DEC_SAMP_16B_PTR));
        #else
            result = CY_GET_REG16(ADC_DEC_SAMPH_16B_PTR);
            result = (result << 16u) | ADC_DEC_SAMP_REG;
            result |=  (uint32)((uint32)ADC_DEC_SAMPM_REG << 8u);
        #endif /* CY_PSOC3 */
    }
    else /*ADC_DEC_SAMP_KEY_HIGH */
    {
        /* Read MSB byte at the end */
        #if (CY_PSOC3)
            result = CY_GET_REG16(ADC_DEC_SAMP_16B_PTR);
                  tmp = ADC_DEC_SAMPH_REG;
            if((tmp & 0x80u) != 0u)
            {   /* Sign extend */
                tmp |= 0xFF00u;
            }
            result |= (uint32)tmp << 16u;
        #else
            result = CY_GET_REG16(ADC_DEC_SAMP_16B_PTR);
            result  |=  (uint32)((uint32)CY_GET_REG16(ADC_DEC_SAMPH_16B_PTR)
<< 16u);
        #endif /* CY_PSOC3 */
    }
    /*  Clear  conversion  complete  status  in  Single  Sample  mode  with
resolution above 16 bit */
    if(ADC_stopConversion != 0u)
    {
        ADC_convDone = 0u;
    }

    return ((int32)result);
}


/*******************************************************************************
*****
* Function Name: ADC_SetOffset
********************************************************************************
*****
*
* Summary:
*  Sets the ADC offset which is used by the functions ADC_CountsTo_uVolts,
*  ADC_CountsTo_mVolts, and ADC_CountsTo_Volts to subtract the offset from
the
*  given reading before calculating the voltage conversion.
*
* Parameters:
*  int32:  This value is a measured value when the inputs are shorted or
*          connected to the same input voltage.
*
```

```
* Return:
*  None
*
* Global variables:
*  ADC_Offset:  Modified to set the user provided offset. This
*  variable is used for offset calibration purpose.
*
* Side Effects:
*  Affects the ADC_CountsTo_Volts,
*  ADC_CountsTo_mVolts, ADC_CountsTo_uVolts functions
*  by subtracting the given offset.
*
****************************************************************************
****/
void ADC_SetOffset(int32 offset)
{

    ADC_Offset = offset;
}


/***************************************************************************
*****
* Function Name: ADC_SetGain
****************************************************************************
*****
*
* Summary:
*  Sets the ADC gain in counts per volt for the voltage conversion
*  functions below. This value is set by default by the reference and
*  input range settings. It should only be used to further calibrate the
*  ADC with a known input or if an external reference is used. This
*  function may also be used to calibrate an entire signal chain, not
*  just the ADC.
*
* Parameters:
*  int32: ADC gain in counts per volt.
*
* Return:
*  None
*
* Global variables:
*  ADC_CountsPerVolt:  modified to set the ADC gain in counts
*   per volt.
*
* Side Effects:
*  Affects the ADC_CountsTo_Volts,
*  ADC_CountsTo_mVolts, ADC_CountsTo_uVolts functions
*  supplying the correct conversion between ADC counts and voltage.
*
****************************************************************************
****/
void ADC_SetGain(int32 adcGain)
{
    ADC_CountsPerVolt = adcGain;
}


/***************************************************************************
*****
* Function Name: ADC_CountsTo_mVolts
```

```
*****************************************************************************
*****
*
* Summary:
*  Converts the ADC counts output to mVolts as a 16-bit integer. For
*  example, if the ADC measured 0.534 volts, the return value would
*  be 534 mVolts.
*
* Parameters:
*  int32: adcCounts Result from the ADC conversion.
*
* Return:
*  int16:  Result in mVolts
*
* Global variables:
*  ADC_CountsPerVolt:  used to convert ADC counts to mVolts.
*  ADC_Offset:  Used as the offset while converting ADC counts
*   to mVolts.
*
*****************************************************************************
****/
int16 ADC_CountsTo_mVolts(int32 adcCounts)
{

    int16 mVolts;

    /* Convert adcCounts to the right align if left option selected */
    #if(ADC_CFG1_DEC_DIV != 0)
        if(ADC_Config == ADC_CFG1)
        {
            adcCounts /= ADC_CFG1_DEC_DIV;
        }
    #endif /* ADC_CFG1_DEC_DIV */
    #if((ADC_CFG2_DEC_DIV != 0) && (ADC_DEFAULT_NUM_CONFIGS > 1))
        if(ADC_Config == ADC_CFG2)
        {
            adcCounts /= ADC_CFG2_DEC_DIV;
        }
    #endif /* ADC_CFG2_DEC_DIV */
    #if((ADC_CFG3_DEC_DIV != 0) && (ADC_DEFAULT_NUM_CONFIGS > 2))
        if(ADC_Config == ADC_CFG3)
        {
            adcCounts /= ADC_CFG3_DEC_DIV;
        }
    #endif /* ADC_CFG2_DEC_DIV */
    #if((ADC_CFG4_DEC_DIV != 0) && (ADC_DEFAULT_NUM_CONFIGS > 3))
        if(ADC_Config == ADC_CFG4)
        {
            adcCounts /= ADC_CFG4_DEC_DIV;
        }
    #endif /* ADC_CFG2_DEC_DIV */

    /* Subtract ADC offset */
    adcCounts -= ADC_Offset;

    mVolts = (int16)(( adcCounts * ADC_1MV_COUNTS ) / ADC_CountsPerVolt) ;

    return(mVolts);
}
```

```
/***************************************************************************
*****
* Function Name: ADC_CountsTo_Volts
****************************************************************************
*****
*
* Summary:
*  Converts the ADC output to Volts as a floating point number. For
*  example, if the ADC measure a voltage of 1.2345 Volts, the
*  returned result would be +1.2345 Volts.
*
* Parameters:
*  int32 adcCounts:  Result from the ADC conversion.
*
* Return:
*  float32: Result in Volts
*
* Global variables:
*  ADC_CountsPerVolt:  used to convert to Volts.
*  ADC_Offset:  Used as the offset while converting ADC counts
*   to Volts.
*
****************************************************************************
****/
float32 ADC_CountsTo_Volts(int32 adcCounts)
{

    float32 Volts;

    /* Convert adcCounts to the right align if left option selected */
    #if(ADC_CFG1_DEC_DIV != 0)
        if(ADC_Config == ADC_CFG1)
        {
            adcCounts /= ADC_CFG1_DEC_DIV;
        }
    #endif /* ADC_CFG1_DEC_DIV */
    #if((ADC_CFG2_DEC_DIV != 0) && (ADC_DEFAULT_NUM_CONFIGS > 1))
        if(ADC_Config == ADC_CFG2)
        {
            adcCounts /= ADC_CFG2_DEC_DIV;
        }
    #endif /* ADC_CFG2_DEC_DIV */
    #if((ADC_CFG3_DEC_DIV != 0) && (ADC_DEFAULT_NUM_CONFIGS > 2))
        if(ADC_Config == ADC_CFG3)
        {
            adcCounts /= ADC_CFG3_DEC_DIV;
        }
    #endif /* ADC_CFG2_DEC_DIV */
    #if((ADC_CFG4_DEC_DIV != 0) && (ADC_DEFAULT_NUM_CONFIGS > 3))
        if(ADC_Config == ADC_CFG4)
        {
            adcCounts /= ADC_CFG4_DEC_DIV;
        }
    #endif /* ADC_CFG2_DEC_DIV */

    /* Subtract ADC offset */
    adcCounts -= ADC_Offset;

    Volts = (float32)adcCounts / (float32)ADC_CountsPerVolt;

    return( Volts );
```

```
}


/*************************************************************************
*****
* Function Name: ADC_CountsTo_uVolts
**************************************************************************
*****
*
* Summary:
*  Converts the ADC output to uVolts as a 32-bit integer. For example,
*   if the ADC measured -0.02345 Volts, the return value would be -23450
uVolts.
*
* Parameters:
*  int32 adcCounts: Result from the ADC conversion.
*
* Return:
*  int32:  Result in uVolts
*
* Global variables:
*  ADC_CountsPerVolt:  used to convert ADC counts to mVolts.
*  ADC_Offset:  Used as the offset while converting ADC counts
*   to mVolts.
*
* Theory:
*  Care must be taken to not exceed the maximum value for a 31 bit signed
*   number in the conversion to uVolts and at the same time not lose
resolution.
*
*  uVolts = ((A * adcCounts) / ((int32)ADC_CountsPerVolt / B));
*
**************************************************************************
****/
int32 ADC_CountsTo_uVolts(int32 adcCounts)
{

    int32 uVolts;
    int32 coefA;
    int32 coefB;
    uint8 resolution;

    /* Set the resolution based on the configuration */
    /* Convert adcCounts to the right align if left option selected */
    if (ADC_Config == ADC_CFG1)
    {
        resolution = ADC_CFG1_RESOLUTION;
        #if(ADC_CFG1_DEC_DIV != 0)
            adcCounts /= ADC_CFG1_DEC_DIV;
        #endif /* ADC_CFG1_DEC_DIV */
    }
    else if (ADC_Config == ADC_CFG2)
    {
        resolution = ADC_CFG2_RESOLUTION;
        #if(ADC_CFG2_DEC_DIV != 0)
            adcCounts /= ADC_CFG2_DEC_DIV;
        #endif /* ADC_CFG2_DEC_DIV */
    }
    else if (ADC_Config == ADC_CFG3)
    {
        resolution = ADC_CFG3_RESOLUTION;
        #if(ADC_CFG3_DEC_DIV != 0)
            adcCounts /= ADC_CFG3_DEC_DIV;
        #endif /* ADC_CFG3_DEC_DIV */
    }
    else
    {
        resolution = ADC_CFG4_RESOLUTION;
        #if(ADC_CFG4_DEC_DIV != 0)
            adcCounts /= ADC_CFG4_DEC_DIV;
        #endif /* ADC_CFG4_DEC_DIV */
    }

    switch (resolution)
    {
        #if( (ADC_CFG1_RESOLUTION == ADC__BITS_12) || \
             (ADC_CFG2_RESOLUTION == ADC__BITS_12) || \
             (ADC_CFG3_RESOLUTION == ADC__BITS_12) || \
             (ADC_CFG4_RESOLUTION == ADC__BITS_12) )
            case (uint8)ADC__BITS_12:
                coefA = ADC_1UV_COUNTS / ADC_DIVISOR_2;
                coefB = ADC_DIVISOR_2;
                break;
        #endif /* ADC__BITS_12 */
        #if( (ADC_CFG1_RESOLUTION == ADC__BITS_13) || \
             (ADC_CFG2_RESOLUTION == ADC__BITS_13) || \
             (ADC_CFG3_RESOLUTION == ADC__BITS_13) || \
             (ADC_CFG4_RESOLUTION == ADC__BITS_13) )
            case (uint8)ADC__BITS_13:
                coefA = ADC_1UV_COUNTS / ADC_DIVISOR_4;
                coefB = ADC_DIVISOR_4;
                break;
        #endif /* ADC__BITS_13 */
        #if( (ADC_CFG1_RESOLUTION == ADC__BITS_14) || \
             (ADC_CFG2_RESOLUTION == ADC__BITS_14) || \
             (ADC_CFG3_RESOLUTION == ADC__BITS_14) || \
             (ADC_CFG4_RESOLUTION == ADC__BITS_14) )
            case (uint8)ADC__BITS_14:
                coefA = ADC_1UV_COUNTS / ADC_DIVISOR_8;
                coefB = ADC_DIVISOR_8;
                break;
        #endif /* ADC__BITS_14 */
        #if( (ADC_CFG1_RESOLUTION == ADC__BITS_15) || \
             (ADC_CFG2_RESOLUTION == ADC__BITS_15) || \
             (ADC_CFG3_RESOLUTION == ADC__BITS_15) || \
             (ADC_CFG4_RESOLUTION == ADC__BITS_15) )
            case (uint8)ADC__BITS_15:
                coefA = ADC_1UV_COUNTS / ADC_DIVISOR_16;
                coefB = ADC_DIVISOR_16;
                break;
        #endif /* ADC__BITS_15 */
        #if( (ADC_CFG1_RESOLUTION == ADC__BITS_16) || \
             (ADC_CFG2_RESOLUTION == ADC__BITS_16) || \
             (ADC_CFG3_RESOLUTION == ADC__BITS_16) || \
             (ADC_CFG4_RESOLUTION == ADC__BITS_16) )
            case (uint8)ADC__BITS_16:
                coefA = ADC_1UV_COUNTS / ADC_DIVISOR_32;
                coefB = ADC_DIVISOR_32;
                break;
        #endif /* ADC__BITS_16 */
        #if( (ADC_CFG1_RESOLUTION == ADC__BITS_17) || \
             (ADC_CFG2_RESOLUTION == ADC__BITS_17) || \
             (ADC_CFG3_RESOLUTION == ADC__BITS_17) || \
             (ADC_CFG4_RESOLUTION == ADC__BITS_17) )
            case (uint8)ADC__BITS_17:
                coefA = ADC_1UV_COUNTS / ADC_DIVISOR_64;
                coefB = ADC_DIVISOR_64;
                break;
        #endif /* ADC__BITS_17 */
        #if( (ADC_CFG1_RESOLUTION == ADC__BITS_18) || \
             (ADC_CFG2_RESOLUTION == ADC__BITS_18) || \
             (ADC_CFG3_RESOLUTION == ADC__BITS_18) || \
             (ADC_CFG4_RESOLUTION == ADC__BITS_18) )
            case (uint8)ADC__BITS_18:
                coefA = ADC_1UV_COUNTS / ADC_DIVISOR_125;
                coefB = ADC_DIVISOR_125;
                break;
        #endif /* ADC__BITS_18 */
        #if( (ADC_CFG1_RESOLUTION == ADC__BITS_19) || \
             (ADC_CFG2_RESOLUTION == ADC__BITS_19) || \
             (ADC_CFG3_RESOLUTION == ADC__BITS_19) || \
             (ADC_CFG4_RESOLUTION == ADC__BITS_19) )
            case (uint8)ADC__BITS_19:
                coefA = ADC_1UV_COUNTS / ADC_DIVISOR_250;
                coefB = ADC_DIVISOR_250;
                break;
        #endif /* ADC__BITS_19 */
        #if( (ADC_CFG1_RESOLUTION == ADC__BITS_20) || \
             (ADC_CFG2_RESOLUTION == ADC__BITS_20) || \
             (ADC_CFG3_RESOLUTION == ADC__BITS_20) || \
             (ADC_CFG4_RESOLUTION == ADC__BITS_20) )
            case (uint8)ADC__BITS_20:
                coefA = ADC_1UV_COUNTS / ADC_DIVISOR_500;
                coefB = ADC_DIVISOR_500;
                break;
        #endif /* ADC__BITS_20 */
        default:    /* resolution < 12 */
            /* 11 bits ADC + 2^20(1048576) = 31 bits */
            coefA = ADC_1UV_COUNTS;
            coefB = ADC_DIVISOR_1;
            break;
    }
    coefB = ADC_CountsPerVolt / coefB;
    uVolts  =  ((coefA  *  adcCounts)  /  coefB)  -  ((coefA  *  ADC_Offset)  /
coefB);

    return( uVolts );
}


/***************************************************************************
*****
* Function Name: ADC_InitConfig(uint8 config)
****************************************************************************
*****
*
* Summary:
*  Initializes all registers based on customizer settings
*
* Parameters:
*   void
*
* Return:
```

```
*  None
*
* Global variables:
*  ADC_CountsPerVolt:  Used to set the default counts per volt.
*
* Side Effects: Rewrites the coherency set by ADC_SetCoherency()
*   API to the default value.
*
****************************************************************************
****/
static void ADC_InitConfig(uint8 config)
{
    ADC_stopConversion = 0u;

    if (config == 1u)
    {
        /* Default Config */
        ADC_DEC_CR_REG      = ADC_CFG1_DEC_CR;
        ADC_DEC_SHIFT1_REG  = ADC_CFG1_DEC_SHIFT1;
        ADC_DEC_SHIFT2_REG  = ADC_CFG1_DEC_SHIFT2;
        ADC_DEC_DR2_REG     = ADC_CFG1_DEC_DR2;
        ADC_DEC_DR2H_REG    = ADC_CFG1_DEC_DR2H;
        ADC_DEC_DR1_REG     = ADC_CFG1_DEC_DR1;
        ADC_DEC_OCOR_REG    = ADC_CFG1_DEC_OCOR;
        ADC_DEC_OCORM_REG   = ADC_CFG1_DEC_OCORM;
        ADC_DEC_OCORH_REG   = ADC_CFG1_DEC_OCORH;
        ADC_DEC_COHER_REG   = ADC_CFG1_DEC_COHER;

        ADC_DSM_CR4_REG     = ADC_CFG1_DSM_CR4;
        ADC_DSM_CR5_REG     = ADC_CFG1_DSM_CR5;
        ADC_DSM_CR6_REG     = ADC_CFG1_DSM_CR6;
        ADC_DSM_CR7_REG     = ADC_CFG1_DSM_CR7;
        ADC_DSM_CR10_REG    = ADC_CFG1_DSM_CR10;
        ADC_DSM_CR11_REG    = ADC_CFG1_DSM_CR11;
        ADC_DSM_CR12_REG    = ADC_CFG1_DSM_CR12;
        ADC_DSM_CR14_REG    = ADC_CFG1_DSM_CR14;
        ADC_DSM_CR15_REG    = ADC_CFG1_DSM_CR15;
        ADC_DSM_CR16_REG    = ADC_CFG1_DSM_CR16;
        ADC_DSM_CR17_REG    = ADC_CFG1_DSM_CR17;
        /* Set DSM_REF0_REG by disabling and enabling the PRESS circuit */
        ADC_SetDSMRef0Reg(ADC_CFG1_DSM_REF0);
        ADC_DSM_REF2_REG    = ADC_CFG1_DSM_REF2;
        ADC_DSM_REF3_REG    = ADC_CFG1_DSM_REF3;

        ADC_DSM_BUF0_REG    = ADC_CFG1_DSM_BUF0;
        ADC_DSM_BUF1_REG    = ADC_CFG1_DSM_BUF1;
        ADC_DSM_BUF2_REG    = ADC_CFG1_DSM_BUF2;
        ADC_DSM_BUF3_REG    = ADC_CFG1_DSM_BUF3;

        /* To select either Vssa or Vref to -ve input of DSM depending on
        *  the input  range selected.
        */
        #if(ADC_DEFAULT_INPUT_MODE)
            #if (ADC_CFG1_INPUT_RANGE == ADC_IR_VSSA_TO_2VREF)
                ADC_AMux_Select(1u);
            #else
                ADC_AMux_Select(0u);
            #endif /* ADC_IR_VSSA_TO_2VREF) */
        #endif /* ADC_DEFAULT_INPUT_MODE */
        /*  Set  the  Conversion  stop  if  resolution  is  above  16  bit  and
conversion
        *   mode is Single sample
        */
        #if(ADC_CFG1_RESOLUTION > 16 && \
            ADC_CFG1_CONV_MODE == ADC_MODE_SINGLE_SAMPLE)
            ADC_stopConversion = 1u;
        #endif /* Single sample with resolution above 16 bits. */

        ADC_CountsPerVolt = (int32)ADC_CFG1_COUNTS_PER_VOLT;

        ADC_Ext_CP_Clk_SetDividerRegister(ADC_CFG1_CP_CLK_DIVIDER, 1u);

        /* This is only valid if there is an internal clock */
        #if(ADC_DEFAULT_INTERNAL_CLK)
            ADC_theACLK_SetDividerRegister(ADC_CFG1_ADC_CLK_DIVIDER, 1u);
        #endif /* ADC_DEFAULT_INTERNAL_CLK */

        #if(ADC_IRQ_REMOVE == 0u)
            /* Set interrupt vector */
            (void)CyIntSetVector(ADC_INTC_NUMBER, &ADC_ISR1);
        #endif   /* End ADC_IRQ_REMOVE */
    }

    #if(ADC_DEFAULT_NUM_CONFIGS > 1)
        if(config == 2u)
        {
            /* Second Config */
            ADC_DEC_CR_REG      = ADC_CFG2_DEC_CR;
            ADC_DEC_SHIFT1_REG  = ADC_CFG2_DEC_SHIFT1;
            ADC_DEC_SHIFT2_REG  = ADC_CFG2_DEC_SHIFT2;
            ADC_DEC_DR2_REG     = ADC_CFG2_DEC_DR2;
            ADC_DEC_DR2H_REG    = ADC_CFG2_DEC_DR2H;
            ADC_DEC_DR1_REG     = ADC_CFG2_DEC_DR1;
            ADC_DEC_OCOR_REG    = ADC_CFG2_DEC_OCOR;
            ADC_DEC_OCORM_REG   = ADC_CFG2_DEC_OCORM;
            ADC_DEC_OCORH_REG   = ADC_CFG2_DEC_OCORH;
            ADC_DEC_COHER_REG   = ADC_CFG2_DEC_COHER;

            ADC_DSM_CR4_REG     = ADC_CFG2_DSM_CR4;
            ADC_DSM_CR5_REG     = ADC_CFG2_DSM_CR5;
            ADC_DSM_CR6_REG     = ADC_CFG2_DSM_CR6;
            ADC_DSM_CR7_REG     = ADC_CFG2_DSM_CR7;
            ADC_DSM_CR10_REG    = ADC_CFG2_DSM_CR10;
            ADC_DSM_CR11_REG    = ADC_CFG2_DSM_CR11;
            ADC_DSM_CR12_REG    = ADC_CFG2_DSM_CR12;
            ADC_DSM_CR14_REG    = ADC_CFG2_DSM_CR14;
            ADC_DSM_CR15_REG    = ADC_CFG2_DSM_CR15;
            ADC_DSM_CR16_REG    = ADC_CFG2_DSM_CR16;
            ADC_DSM_CR17_REG    = ADC_CFG2_DSM_CR17;
            /* Set DSM_REF0_REG by disabling and enabling the PRESS cirucit
*/
            ADC_SetDSMRef0Reg(ADC_CFG2_DSM_REF0);
            ADC_DSM_REF2_REG    = ADC_CFG2_DSM_REF2;
            ADC_DSM_REF3_REG    = ADC_CFG2_DSM_REF3;

            ADC_DSM_BUF0_REG    = ADC_CFG2_DSM_BUF0;
            ADC_DSM_BUF1_REG    = ADC_CFG2_DSM_BUF1;
            ADC_DSM_BUF2_REG    = ADC_CFG2_DSM_BUF2;
            ADC_DSM_BUF3_REG    = ADC_CFG2_DSM_BUF3;
            /* To select either Vssa or Vref to -ve input of DSM depending
on
            *  the input range selected.
            */

            #if(ADC_DEFAULT_INPUT_MODE)
                #if (ADC_CFG2_INPUT_RANGE == ADC_IR_VSSA_TO_2VREF)
                    ADC_AMux_Select(1u);
                #else
                    ADC_AMux_Select(0u);
                #endif /* ADC_IR_VSSA_TO_2VREF) */
            #endif /* ADC_DEFAULT_INPUT_MODE */

            /* Set the Conversion stop if resolution is above 16 bit and
            *   conversion mode is Single sample
            */
            #if(ADC_CFG2_RESOLUTION > 16 && \
                ADC_CFG2_CONV_MODE == ADC_MODE_SINGLE_SAMPLE)
                ADC_stopConversion = 1u;
            #endif /* Single sample with resolution above 16 bits. */

            ADC_CountsPerVolt = (int32)ADC_CFG2_COUNTS_PER_VOLT;

            ADC_Ext_CP_Clk_SetDividerRegister(ADC_CFG2_CP_CLK_DIVIDER, 1u);

            /* This is only valid if there is an internal clock */
            #if(ADC_DEFAULT_INTERNAL_CLK)
                ADC_theACLK_SetDividerRegister(ADC_CFG2_ADC_CLK_DIVIDER,
1u);
            #endif /* ADC_DEFAULT_INTERNAL_CLK */

            #if(ADC_IRQ_REMOVE == 0u)
                /* Set interrupt vector */
                (void)CyIntSetVector(ADC_INTC_NUMBER, &ADC_ISR2);
            #endif   /* End ADC_IRQ_REMOVE */
        }
    #endif /* ADC_DEFAULT_NUM_CONFIGS > 1 */

    #if(ADC_DEFAULT_NUM_CONFIGS > 2)
        if(config == 3u)
        {
            /* Third Config */
            ADC_DEC_CR_REG      = ADC_CFG3_DEC_CR;
            ADC_DEC_SHIFT1_REG  = ADC_CFG3_DEC_SHIFT1;
            ADC_DEC_SHIFT2_REG  = ADC_CFG3_DEC_SHIFT2;
            ADC_DEC_DR2_REG     = ADC_CFG3_DEC_DR2;
            ADC_DEC_DR2H_REG    = ADC_CFG3_DEC_DR2H;
            ADC_DEC_DR1_REG     = ADC_CFG3_DEC_DR1;
            ADC_DEC_OCOR_REG    = ADC_CFG3_DEC_OCOR;
            ADC_DEC_OCORM_REG   = ADC_CFG3_DEC_OCORM;
            ADC_DEC_OCORH_REG   = ADC_CFG3_DEC_OCORH;
            ADC_DEC_COHER_REG   = ADC_CFG3_DEC_COHER;

            ADC_DSM_CR4_REG     = ADC_CFG3_DSM_CR4;
            ADC_DSM_CR5_REG     = ADC_CFG3_DSM_CR5;
            ADC_DSM_CR6_REG     = ADC_CFG3_DSM_CR6;
            ADC_DSM_CR7_REG     = ADC_CFG3_DSM_CR7;
            ADC_DSM_CR10_REG    = ADC_CFG3_DSM_CR10;
            ADC_DSM_CR11_REG    = ADC_CFG3_DSM_CR11;
            ADC_DSM_CR12_REG    = ADC_CFG3_DSM_CR12;
            ADC_DSM_CR14_REG    = ADC_CFG3_DSM_CR14;
            ADC_DSM_CR15_REG     = ADC_CFG3_DSM_CR15;
            ADC_DSM_CR16_REG     = ADC_CFG3_DSM_CR16;
            ADC_DSM_CR17_REG     = ADC_CFG3_DSM_CR17;
            /* Set DSM_REF0_REG by disabling and enabling the PRESS circuit 
*/
            ADC_SetDSMRef0Reg(ADC_CFG3_DSM_REF0);
            ADC_DSM_REF2_REG     = ADC_CFG3_DSM_REF2;
            ADC_DSM_REF3_REG     = ADC_CFG3_DSM_REF3;

            ADC_DSM_BUF0_REG     = ADC_CFG3_DSM_BUF0;
            ADC_DSM_BUF1_REG     = ADC_CFG3_DSM_BUF1;
            ADC_DSM_BUF2_REG     = ADC_CFG3_DSM_BUF2;
            ADC_DSM_BUF3_REG     = ADC_CFG3_DSM_BUF3;

            /* To select either Vssa or Vref to -ve input of DSM depending 
on
            *  the input range selected.
            */
            #if(ADC_DEFAULT_INPUT_MODE)
                #if (ADC_CFG3_INPUT_RANGE == ADC_IR_VSSA_TO_2VREF)
                    ADC_AMux_Select(1u);
                #else
                    ADC_AMux_Select(0u);
                #endif /* ADC_IR_VSSA_TO_2VREF) */
            #endif /* ADC_DEFAULT_INPUT_MODE */

            /* Set the Conversion stop if resolution is above 16 bit and
               conversion  mode is Single sample */
            #if(ADC_CFG3_RESOLUTION > 16 && \
                ADC_CFG3_CONV_MODE == ADC_MODE_SINGLE_SAMPLE)
                ADC_stopConversion = 1u;
            #endif /* Single sample with resolution above 16 bits */

            ADC_CountsPerVolt = (int32)ADC_CFG3_COUNTS_PER_VOLT;

            ADC_Ext_CP_Clk_SetDividerRegister(ADC_CFG3_CP_CLK_DIVIDER, 1u);

            /* This is only valid if there is an internal clock */
            #if(ADC_DEFAULT_INTERNAL_CLK)
                ADC_theACLK_SetDividerRegister(ADC_CFG3_ADC_CLK_DIVIDER, 
1u);
            #endif /* ADC_DEFAULT_INTERNAL_CLK */

            #if(ADC_IRQ_REMOVE == 0u)
                /* Set interrupt vector */
                (void)CyIntSetVector(ADC_INTC_NUMBER, &ADC_ISR3);
            #endif   /* End ADC_IRQ_REMOVE */
        }
    #endif /* ADC_DEFAULT_NUM_CONFIGS > 2 */

    #if(ADC_DEFAULT_NUM_CONFIGS > 3)
        if (config == 4u)
        {
            /* Fourth Config */
            ADC_DEC_CR_REG       = ADC_CFG4_DEC_CR;
            ADC_DEC_SHIFT1_REG   = ADC_CFG4_DEC_SHIFT1;
            ADC_DEC_SHIFT2_REG   = ADC_CFG4_DEC_SHIFT2;
            ADC_DEC_DR2_REG      = ADC_CFG4_DEC_DR2;
            ADC_DEC_DR2H_REG     = ADC_CFG4_DEC_DR2H;
            ADC_DEC_DR1_REG      = ADC_CFG4_DEC_DR1;
            ADC_DEC_OCOR_REG     = ADC_CFG4_DEC_OCOR;
            ADC_DEC_OCORM_REG   = ADC_CFG4_DEC_OCORM;
            ADC_DEC_OCORH_REG   = ADC_CFG4_DEC_OCORH;
            ADC_DEC_COHER_REG   = ADC_CFG4_DEC_COHER;

            ADC_DSM_CR4_REG     = ADC_CFG4_DSM_CR4;
            ADC_DSM_CR5_REG     = ADC_CFG4_DSM_CR5;
            ADC_DSM_CR6_REG     = ADC_CFG4_DSM_CR6;
            ADC_DSM_CR7_REG     = ADC_CFG4_DSM_CR7;
            ADC_DSM_CR10_REG    = ADC_CFG4_DSM_CR10;
            ADC_DSM_CR11_REG    = ADC_CFG4_DSM_CR11;
            ADC_DSM_CR12_REG    = ADC_CFG4_DSM_CR12;
            ADC_DSM_CR14_REG    = ADC_CFG4_DSM_CR14;
            ADC_DSM_CR15_REG    = ADC_CFG4_DSM_CR15;
            ADC_DSM_CR16_REG    = ADC_CFG4_DSM_CR16;
            ADC_DSM_CR17_REG    = ADC_CFG4_DSM_CR17;
            /* Set DSM_REF0_REG by disabling and enabling the PRESS circuit
*/
            ADC_SetDSMRef0Reg(ADC_CFG4_DSM_REF0);
            ADC_DSM_REF2_REG    = ADC_CFG4_DSM_REF2;
            ADC_DSM_REF3_REG    = ADC_CFG4_DSM_REF3;

            ADC_DSM_BUF0_REG    = ADC_CFG4_DSM_BUF0;
            ADC_DSM_BUF1_REG    = ADC_CFG4_DSM_BUF1;
            ADC_DSM_BUF2_REG    = ADC_CFG4_DSM_BUF2;
            ADC_DSM_BUF3_REG    = ADC_CFG4_DSM_BUF3;

            /* To select either Vssa or Vref to -ve input of DSM depending
on
            *  the input range selected.
            */
            #if(ADC_DEFAULT_INPUT_MODE)
                #if (ADC_CFG4_INPUT_RANGE == ADC_IR_VSSA_TO_2VREF)
                    ADC_AMux_Select(1u);
                #else
                    ADC_AMux_Select(0u);
                #endif /* ADC_IR_VSSA_TO_2VREF)  */
            #endif /* ADC_DEFAULT_INPUT_MODE */

            /* Set the Conversion stop if resolution is above 16 bit and
               conversion mode is Single sample */
            #if(ADC_CFG4_RESOLUTION > 16 && \
                ADC_CFG4_CONV_MODE == ADC_MODE_SINGLE_SAMPLE)
                ADC_stopConversion = 1u;
            #endif /* Single sample with resolution above 16 bits */

            ADC_CountsPerVolt = (int32)ADC_CFG4_COUNTS_PER_VOLT;

            ADC_Ext_CP_Clk_SetDividerRegister(ADC_CFG4_CP_CLK_DIVIDER, 1u);

            /* This is only valid if there is an internal clock */
            #if(ADC_DEFAULT_INTERNAL_CLK)
                ADC_theACLK_SetDividerRegister(ADC_CFG4_ADC_CLK_DIVIDER,
1u);
            #endif /* ADC_DEFAULT_INTERNAL_CLK */

            #if(ADC_IRQ_REMOVE == 0u)
                /* Set interrupt vector */
                (void)CyIntSetVector(ADC_INTC_NUMBER, &ADC_ISR4);
            #endif   /* End ADC_IRQ_REMOVE */
        }
    #endif /* ADC_DEFAULT_NUM_CONFIGS > 3 */
```

```
}


/***************************************************************************
*****
* Function Name: ADC_SelectCofiguration
****************************************************************************
*****
*
* Summary:
*  Sets one of up to four ADC configurations. Before setting the new
*  configuration, the ADC is stopped and powered down. After setting
*  the new configuration, the ADC can be powered and conversion
*  can be restarted depending up on the value of second parameter
*  restart. If the value of this parameter is 1, then ADC will be
*  restarted. If this value is zero, then user must call ADC_Start
*  and ADC_StartConvert() to restart the conversion.
*
* Parameters:
*  config:  configuration user wants to select.
*           Valid range: 1..4
*   restart:   Restart  option.  1  means  start  the  ADC  and  restart  the
conversion.
*                             0 means do not start the ADC and conversion.
*
* Return:
*  None
*
****************************************************************************
****/
void ADC_SelectConfiguration(uint8 config, uint8 restart)

{
    /* Check whether the configuration number is valid or not */
    if((config > 0u) && (config <= ADC_DEFAULT_NUM_CONFIGS))
    {
        /* Set the flag to ensure Start() API doesn't override the
            *  selected configuration
            */
        if(ADC_initVar == 0u)
        {
            ADC_started = 1u;
        }

        /* Update the config flag */
        ADC_Config = config;

        /* Stop the ADC  */
        ADC_Stop();

        /* Set the  ADC registers based on the configuration */
        ADC_InitConfig(config);

        /* Compensate the gain */
        ADC_DEC_GVAL_REG = ADC_gcor[config - 1u].gval;
        CY_SET_REG16(ADC_DEC_GCOR_16B_PTR, ADC_gcor[config - 1u].gcor);

        if(restart == 1u)
        {
            /* Restart the ADC */
            ADC_Start();
            /* Restart the ADC conversion */
            ADC_StartConvert();
        }
    }
    else
    {
        /* Halt CPU in debug mode if config is out of valid range */
        CYASSERT(0u != 0u);
    }
}


/***************************************************************************
*****
* Function Name: ADC_GainCompensation
****************************************************************************
*****
*
* Summary:
*    This  API  calculates  the  trim  value  and  then  store  this  to  gcor
structure.
*
* Parameters:
*  inputRange:  input range for which trim value is to be calculated.
*  IdealDecGain:  Ideal Decimator gain for the selected resolution and
*               conversion  mode.
*   IdealOddDecGain:   Ideal  odd  decimation  gain  for  the  selected  resolution
and
                      conversion mode.
*   resolution:   Resolution  to  select  the  proper  flash  location  for  trim
value.
*  config:      Specifies the configuration number
*               Valid range: 1..4
*
* Return:
*  None
*
****************************************************************************
****/
static  void  ADC_GainCompensation(uint8  inputRange,  uint16  idealDecGain,
uint16 idealOddDecGain,
                              uint8 resolution, uint8 config)
{
    int8 flash;
      int32 normalised;
      uint16 gcorValue;
    uint32 gcorTmp;

    if((config > 0u) && (config <= ADC_DEFAULT_NUM_CONFIGS))
    {
        switch(inputRange)
        {
            case ADC_IR_VNEG_VREF_DIFF:
            case ADC_IR_VSSA_TO_2VREF:
                /* Normalize the flash Value */
                if(resolution > 15u)
                {
                    flash = ADC_DEC_TRIM_VREF_DIFF_16_20;
                }
                else
                {
                    flash = ADC_DEC_TRIM_VREF_DIFF_8_15;
                }
                break;

            case ADC_IR_VNEG_VREF_2_DIFF:
                /* Normalize the flash Value */
                if(resolution > 15u)
                {
                    flash = ADC_DEC_TRIM_VREF_2_DIFF_16_20;
                }
                else
                {
                    flash = ADC_DEC_TRIM_VREF_2_DIFF_8_15;
                }
                break;

            case ADC_IR_VNEG_VREF_4_DIFF:
                /* Normalize the flash Value */
                if(resolution > 15u)
                {
                    flash = ADC_DEC_TRIM_VREF_4_DIFF_16_20;
                }
                else
                {
                    flash = ADC_DEC_TRIM_VREF_4_DIFF_8_15;
                }
                break;

            case ADC_IR_VNEG_VREF_16_DIFF:
                /* Normalize the flash Value */
                if(resolution > 15u)
                {
                    flash = ADC_DEC_TRIM_VREF_16_DIFF_16_20;
                }
                else
                {
                    flash = ADC_DEC_TRIM_VREF_16_DIFF_8_15;
                }
                break;

            default:
                flash = 0;
                break;
        }

        /* Add two values */
            normalised = (int32)idealDecGain + ((int32)flash * 32);
        gcorTmp = (uint32)normalised * (uint32)idealOddDecGain;
        gcorValue = (uint16)(gcorTmp / ADC_IDEAL_GAIN_CONST);

        if (resolution < (ADC_MAX_GVAL - 1u))
        {
            gcorValue = (gcorValue >> (ADC_MAX_GVAL - (resolution + 1u)));
            ADC_gcor[config - 1u].gval = (resolution + 1u);
        }
        else
        {
            /* Use all 16 bits */
            ADC_gcor[config - 1u].gval = ADC_MAX_GVAL;
        }
```

```
        /* Save the gain correction register value */
        ADC_gcor[config - 1u].gcor = gcorValue;
    }
    else
    {
        /* Halt CPU in debug mode if config is out of valid range */
        CYASSERT(0u != 0u);
    }
}


/***************************************************************************
****
* Function Name: ADC_SetDSMRef0Reg(uint8)
****************************************************************************
***
*
* Summary:
*  This API sets the DSM_REF0 register. This is written for internal use.
*
* Parameters:
*  value:  Value to be written to DSM_REF0 register.
*
* Return:
*  None
*
****************************************************************************
***/
static void ADC_SetDSMRef0Reg(uint8 value)
{
    uint8 enableInterrupts;
    enableInterrupts = CyEnterCriticalSection();

    /* Disable PRES, Enable power to VCMBUF0, REFBUF0 and REFBUF1, enable
PRES */
    ADC_RESET_CR4_REG |= (ADC_IGNORE_PRESA1 | ADC_IGNORE_PRESD1);
    ADC_RESET_CR5_REG |= (ADC_IGNORE_PRESA2 | ADC_IGNORE_PRESD2);
    ADC_DSM_REF0_REG = value;

    /* Wait for 3 microseconds */
    CyDelayUs(ADC_PRES_DELAY_TIME);
    /* Enable the press circuit */
    ADC_RESET_CR4_REG &= (uint8)~(ADC_IGNORE_PRESA1 | ADC_IGNORE_PRESD1);
    ADC_RESET_CR5_REG &= (uint8)~(ADC_IGNORE_PRESA2 | ADC_IGNORE_PRESD2);

    CyExitCriticalSection(enableInterrupts);
}


/***************************************************************************
*****
* Function Name: ADC_Read8
****************************************************************************
*****
*
* Summary:
*  This function simplifies getting results from the ADC when only a
*  single reading is required. When called, it will start ADC
*  conversions, wait for the conversion to be complete, stop ADC
*  conversion and return the result. This is a blocking function and will
*  not return until the result is ready.
*
* Parameters:
*  None
*
* Return:
*  int8:  ADC result.
*
***************************************************************************
****/
int8 ADC_Read8(void)
{
    int8 result;

    /* Clear pending conversion done status */
    ADC_DEC_SR_REG |= ADC_DEC_INTR_CLEAR;
    ADC_StartConvert();
    (void)ADC_IsEndConversion(ADC_WAIT_FOR_RESULT);
     result = ADC_GetResult8();
    ADC_StopConvert();

     return(result);
}


/**************************************************************************
*****
* Function Name: ADC_Read16
***************************************************************************
*****
*
* Summary:
*  This function simplifies getting results from the ADC when only a
*  single reading is required. When called, it will start ADC
*  conversions, wait for the conversion to be complete, stop ADC
*  conversion and return the result. This is a blocking function and will
*  not return until the result is ready.
*
* Parameters:
*   void
*
* Return:
*  int16:  ADC result.
*
***************************************************************************
****/
int16 ADC_Read16(void)
{
    int16 result;

    /* Clear pending conversion done status */
    ADC_DEC_SR_REG |= ADC_DEC_INTR_CLEAR;
    ADC_StartConvert();
    (void)ADC_IsEndConversion(ADC_WAIT_FOR_RESULT);
     result = ADC_GetResult16();
    ADC_StopConvert();

     return(result);
}
```

```
/*************************************************************************
*****
* Function Name: ADC_Read32
*************************************************************************
*****
*
* Summary:
*  This function simplifies getting results from the ADC when only a
*  single reading is required. When called, it will start ADC
*  conversions, wait for the conversion to be complete, stop ADC
*  conversion and return the result. This is a blocking function and will
*  not return until the result is ready.
*
* Parameters:
*  None
*
* Return:
*  int32: ADC result.
*
*************************************************************************
****/
int32 ADC_Read32(void)
{
    int32 result;

    /* Clear pending conversion done status */
    ADC_DEC_SR_REG |= ADC_DEC_INTR_CLEAR;
    ADC_StartConvert();
    (void)ADC_IsEndConversion(ADC_WAIT_FOR_RESULT);
     result = ADC_GetResult32();
    ADC_StopConvert();

     return(result);
}


/* [] END OF FILE */
/*************************************************************************
*****
* File Name: ADC_AMux.c
* Version 1.80
*
*  Description:
*    This file contains all functions required for the analog multiplexer
*    AMux User Module.
*
*   Note:
*
*************************************************************************
****
* Copyright 2008-2010, Cypress Semiconductor Corporation.  All rights
reserved.
* You may use this file only in accordance with the license, terms,
conditions,
*  disclaimers, and limitations in the end user license agreement
accompanying
* the software package with which this file was provided.
*************************************************************************
*****/

#include "ADC_AMux.h"
```

```
static uint8 ADC_AMux_lastChannel = ADC_AMux_NULL_CHANNEL;


/*************************************************************************
*****
* Function Name: ADC_AMux_Start
**************************************************************************
*****
* Summary:
*  Disconnect all channels.
*
* Parameters:
*  void
*
* Return:
*  void
*
**************************************************************************
****/
void ADC_AMux_Start(void)
{
    uint8 chan;

    for(chan = 0u; chan < ADC_AMux_CHANNELS ; chan++)
    {
#if (ADC_AMux_MUXTYPE == ADC_AMux_MUX_SINGLE)
        ADC_AMux_Unset(chan);
#else
        ADC_AMux_CYAMUXSIDE_A_Unset(chan);
        ADC_AMux_CYAMUXSIDE_B_Unset(chan);
#endif
    }

    ADC_AMux_lastChannel = ADC_AMux_NULL_CHANNEL;
}


#if (!ADC_AMux_ATMOSTONE)
/*************************************************************************
*****
* Function Name: ADC_AMux_Select
**************************************************************************
*****
* Summary:
*  This functions first disconnects all channels then connects the given
*  channel.
*
* Parameters:
*  channel:  The channel to connect to the common terminal.
*
* Return:
*  void
*
**************************************************************************
****/
void ADC_AMux_Select(uint8 channel)
{
    ADC_AMux_DisconnectAll();        /* Disconnect all previous connections
*/
    ADC_AMux_Connect(channel);       /* Make the given selection */
    ADC_AMux_lastChannel = channel;  /* Update last channel */
}
#endif


/*******************************************************************************
* Function Name: ADC_AMux_FastSelect
********************************************************************************
* Summary:
*  This function first disconnects the last connection made with FastSelect or
*   Select, then connects the given channel. The FastSelect function is similar
*  to the Select function, except it is faster since it only disconnects the
*  last channel selected rather than all channels.
*
* Parameters:
*  channel:  The channel to connect to the common terminal.
*
* Return:
*  void
*
*******************************************************************************/
void ADC_AMux_FastSelect(uint8 channel)
{
    /* Disconnect the last valid channel */
    if( ADC_AMux_lastChannel != ADC_AMux_NULL_CHANNEL)
    {
        ADC_AMux_Disconnect(ADC_AMux_lastChannel);
    }

    /* Make the new channel connection */
#if (ADC_AMux_MUXTYPE == ADC_AMux_MUX_SINGLE)
    ADC_AMux_Set(channel);
#else
    ADC_AMux_CYAMUXSIDE_A_Set(channel);
    ADC_AMux_CYAMUXSIDE_B_Set(channel);
#endif


    ADC_AMux_lastChannel = channel;   /* Update last channel */
}


#if (ADC_AMux_MUXTYPE == ADC_AMux_MUX_DIFF)
#if (!ADC_AMux_ATMOSTONE)
/*******************************************************************************
* Function Name: ADC_AMux_Connect
********************************************************************************
* Summary:
*   This function connects the given channel without affecting other connections.
*
* Parameters:
*  channel:  The channel to connect to the common terminal.
```

```
*
* Return:
*  void
*
*************************************************************************
****/
void ADC_AMux_Connect(uint8 channel)
{
    ADC_AMux_CYAMUXSIDE_A_Set(channel);
    ADC_AMux_CYAMUXSIDE_B_Set(channel);
}
#endif

/************************************************************************
*****
* Function Name: ADC_AMux_Disconnect
*************************************************************************
*****
* Summary:
*  This function disconnects the given channel from the common or output
*  terminal without affecting other connections.
*
* Parameters:
*  channel:  The channel to disconnect from the common terminal.
*
* Return:
*  void
*
*************************************************************************
****/
void ADC_AMux_Disconnect(uint8 channel)
{
    ADC_AMux_CYAMUXSIDE_A_Unset(channel);
    ADC_AMux_CYAMUXSIDE_B_Unset(channel);
}
#endif

#if (ADC_AMux_ATMOSTONE)
/************************************************************************
*****
* Function Name: ADC_AMux_DisconnectAll
*************************************************************************
*****
* Summary:
*  This function disconnects all channels.
*
* Parameters:
*  void
*
* Return:
*  void
*
*************************************************************************
****/
void ADC_AMux_DisconnectAll(void)
{
    if(ADC_AMux_lastChannel != ADC_AMux_NULL_CHANNEL)
    {
        ADC_AMux_Disconnect(ADC_AMux_lastChannel);
        ADC_AMux_lastChannel = ADC_AMux_NULL_CHANNEL;
    }
}
#endif

/* [] END OF FILE */
/*******************************************************************************
*****
* File Name: Audio.c
* Version 1.20
*
* Description:
*  This file contains the setup, control and status commands for the S/PDIF
TX
*  component.
*
* Note:
*
*******************************************************************************
****
* Copyright 2011-2012, Cypress Semiconductor Corporation.  All rights
reserved.
* You may use this file only in accordance with the license, terms,
conditions,
* disclaimers, and limitations in the end user license agreement
accompanying
* the software package with which this file was provided.
*******************************************************************************
****/

#include "Audio_PVT.h"

uint8 Audio_initVar = 0u;

#if(0u != Audio_MANAGED_DMA)

    /* Channel status streams used for DMA transfer */
    volatile uint8 Audio_cstStream0[Audio_CST_LENGTH];
    volatile uint8 Audio_cstStream1[Audio_CST_LENGTH];

    /* Channel status streams to change from API at run time */
    volatile uint8 Audio_wrkCstStream0[Audio_CST_LENGTH];
    volatile uint8 Audio_wrkCstStream1[Audio_CST_LENGTH];

    /* Buffer offset variables */
    volatile uint8 Audio_cst0BufOffset = 0u;
    volatile uint8 Audio_cst1BufOffset = 0u;

    /* Cst DMA channels and transfer descriptors */
    static uint8 Audio_cst0Chan;
    static uint8 Audio_cst1Chan;

    static uint8 Audio_cst0Td[2u] = {CY_DMA_INVALID_TD, CY_DMA_INVALID_TD};
    static uint8 Audio_cst1Td[2u] = {CY_DMA_INVALID_TD, CY_DMA_INVALID_TD};

    /* Function prototype to set/release DMA */
    static void Audio_CstDmaInit(void)       ;
    static void Audio_CstDmaRelease(void)    ;

#endif /* 0u != Audio_MANAGED_DMA */
```

```
/*******************************************************************************
*****
* Function Name: Audio_Enable
*******************************************************************************
*****
*
* Summary:
*   Enables S/PDIF interface. Starts the generation of the S/PDIF output
with
*   channel status, but the audio data is set to all 0's. This allows the
S/PDIF
*  receiver to lock on to the component's clock.
*
* Parameters:
*  None.
*
* Return:
*  None.
*
* Reentrant:
*  No.
*
*******************************************************************************
****/
void Audio_Enable(void)
{
    uint8 enableInterrupts;

    enableInterrupts = CyEnterCriticalSection();
    Audio_BCNT_AUX_CTL_REG    |= Audio_BCNT_EN;     /* Bit counter enabling
*/
    Audio_STATUS_AUX_CTL_REG |= Audio_INT_EN;       /* Interrupt generation
enabling */
    CyExitCriticalSection(enableInterrupts);

    #if(0u != Audio_MANAGED_DMA)
        /* Enable channel status ISRs */
        CyIntEnable(Audio_CST_0_ISR_NUMBER);
        CyIntEnable(Audio_CST_1_ISR_NUMBER);

        /* Prepare and enable channel status DMA transfer */
        Audio_CstDmaInit();

        while(0u != (Audio_STATUS_REG & Audio_CHST_FIFOS_NOT_FULL))
        {
            ; /* Wait for DMA fills status FIFOs to proceed */
        }
    #endif /* 0u != Audio_MANAGED_DMA */

    Audio_CONTROL_REG |= Audio_ENBL;
}


#if (0u != Audio_MANAGED_DMA)

/*******************************************************************************
*****
    * Function Name: Audio_CstDmaInit

*******************************************************************************
*****
```

```
    *
    * Summary:
    *  Inits channel status DMA transfer.
    *
    * Parameters:
    *  None.
    *
    * Return:
    *  None.
    *
    * Global Variables:
    *  Audio_cst0Chan - DMA Channel to be used for Channel 0 Status
    *     DMA transfer.
    *  Audio_cst1Chan - DMA Channel to be used for Channel 1 Status
    *     DMA transfer.
    *  Audio_cst0Td[] - TD set to be used for Channel 0 Status DMA
    *     transfer.
    *  Audio_cst1Td[] - TD set to be used for Channel 1 Status DMA
    *     transfer.
    *  Audio_cstStream0[] - Channel 0 Status stream. Used as the source
    *     buffer for Channel 0 Status DMA. Modified when the data is copied
for the
    *     first cycle.
    *  Audio_wrkCstStream0[] - Channel 0 Status intermediate buffer
    *       between API and DMA. This is required to allow changing of
Channel Status
    *     at run time. Used when the data is copied for the first cycle.
    *  Audio_cstStream1[] - Channel 1 Status stream. Used as the source
    *     buffer for Channel 1 Status DMA. Modified when the data is copied
for the
    *     first cycle.
    *  Audio_wrkCstStream1[] - Channel 1 Status intermediate buffer
    *       between API and DMA. This is required to allow changing of
Channel Status
    *     at run time. Used when the data is copied for the first cycle.
    *
    * Reentrant:
    *  No.
    *

*******************************************************************************
****/
    static void Audio_CstDmaInit(void)
    {

        /* Copy channels' status values for the first cycle */
        (void) memcpy((void *) Audio_cstStream0,
                      (void *) Audio_wrkCstStream0, Audio_CST_LENGTH);

        (void) memcpy((void *) Audio_cstStream1,
                      (void *) Audio_wrkCstStream1, Audio_CST_LENGTH);

        Audio_cst0Td[0u] = CyDmaTdAllocate();
        Audio_cst0Td[1u] = CyDmaTdAllocate();

        Audio_cst1Td[0u] = CyDmaTdAllocate();
        Audio_cst1Td[1u] = CyDmaTdAllocate();

        (void) CyDmaTdSetConfiguration(
                Audio_cst0Td[0u],
                Audio_CST_HALF_LENGTH,
                Audio_cst0Td[1u],
                (CY_DMA_TD_INC_SRC_ADR | Audio_Cst0_DMA__TD_TERMOUT_EN));

        (void) CyDmaTdSetConfiguration(
                Audio_cst0Td[1u],
                Audio_CST_HALF_LENGTH,
                Audio_cst0Td[0u],
                (CY_DMA_TD_INC_SRC_ADR | Audio_Cst0_DMA__TD_TERMOUT_EN));

        (void) CyDmaTdSetConfiguration(
                Audio_cst1Td[0u],
                Audio_CST_HALF_LENGTH,
                Audio_cst1Td[1u],
                (CY_DMA_TD_INC_SRC_ADR | Audio_Cst1_DMA__TD_TERMOUT_EN));

        (void) CyDmaTdSetConfiguration(
                Audio_cst1Td[1u],
                Audio_CST_HALF_LENGTH,
                Audio_cst1Td[0u],
                (CY_DMA_TD_INC_SRC_ADR | Audio_Cst1_DMA__TD_TERMOUT_EN));

        (void) CyDmaTdSetAddress(
                Audio_cst0Td[0u],
                LO16((uint32)Audio_cstStream0),
                LO16((uint32)Audio_CST_FIFO_0_PTR));

        (void) CyDmaTdSetAddress(
                Audio_cst0Td[1u],
                LO16((uint32) (&Audio_cstStream0[Audio_CST_HALF_LENGTH])),
                LO16((uint32) Audio_CST_FIFO_0_PTR));

        (void) CyDmaTdSetAddress(
                Audio_cst1Td[0u],
                LO16((uint32) Audio_cstStream1),
                LO16((uint32) Audio_CST_FIFO_1_PTR));

        (void) CyDmaTdSetAddress(
                Audio_cst1Td[1u],
                LO16((uint32) (&Audio_cstStream1[Audio_CST_HALF_LENGTH])),
                LO16((uint32) Audio_CST_FIFO_1_PTR));

        (void) CyDmaChSetInitialTd(Audio_cst0Chan, Audio_cst0Td[0u]);
        (void) CyDmaChSetInitialTd(Audio_cst1Chan, Audio_cst1Td[0u]);

        (void) CyDmaChEnable(Audio_cst0Chan, 1u);
        (void) CyDmaChEnable(Audio_cst1Chan, 1u);
    }


/***************************************************************************
*****
    * Function Name: Audio_CstDmaRelease
***************************************************************************
*****
    *
    * Summary:
    *  Release allocated DMA channels and transfer descriptors.
    *
    * Parameters:
```

```
    *  None.
    *
    * Return:
    *  None.
    *
    * Global Variables:
    *  Audio_cst0Chan - DMA Channel to be used for Channel 0 Status
    *     DMA transfer.
    *  Audio_cst1Chan - DMA Channel to be used for Channel 1 Status
    *     DMA transfer.
    *  Audio_cst0Td[] - TD set to be used for Channel 0 Status DMA
    *     transfer.
    *  Audio_cst1Td[] - TD set to be used for Channel 1 Status DMA
    *     transfer.
    *
    * Reentrant:
    *  No.
    *

***************************************************************************
****/
    static void Audio_CstDmaRelease(void)
    {
        /* Disable the managed channel status DMA */
        (void) CyDmaChDisable(Audio_cst0Chan);
        (void) CyDmaChDisable(Audio_cst1Chan);

        /* Clear any potential DMA requests and re-reset TD pointers */
        while(0u != (CY_DMA_CH_STRUCT_PTR[Audio_cst0Chan].basic_status[0] &
CY_DMA_STATUS_TD_ACTIVE))
        {
            ; /* Wait for to be cleared */
        }

        (void) CyDmaChSetRequest(Audio_cst0Chan, CY_DMA_CPU_TERM_CHAIN);
        (void) CyDmaChEnable    (Audio_cst0Chan, 1u);

        while(0u  !=  (CY_DMA_CH_STRUCT_PTR[Audio_cst0Chan].basic_cfg[0]  &
CY_DMA_STATUS_CHAIN_ACTIVE))
        {
            ; /* Wait for to be cleared */
        }


        while(0u != (CY_DMA_CH_STRUCT_PTR[Audio_cst1Chan].basic_status[0] &
CY_DMA_STATUS_TD_ACTIVE))
        {
            ; /* Wait for to be cleared */
        }

        (void) CyDmaChSetRequest(Audio_cst1Chan, CY_DMA_CPU_TERM_CHAIN);
        (void) CyDmaChEnable    (Audio_cst1Chan, 1u);

        while(0u  !=  (CY_DMA_CH_STRUCT_PTR[Audio_cst1Chan].basic_cfg[0]  &
CY_DMA_STATUS_CHAIN_ACTIVE))
        {
            ; /* Wait for to be cleared */
        }

        /* Release all allocated TDs and mark them as invalid */
        CyDmaTdFree(Audio_cst0Td[0u]);
        CyDmaTdFree(Audio_cst0Td[1u]);
        CyDmaTdFree(Audio_cst1Td[0u]);
        CyDmaTdFree(Audio_cst1Td[1u]);
        Audio_cst0Td[0u] = CY_DMA_INVALID_TD;
        Audio_cst0Td[1u] = CY_DMA_INVALID_TD;
        Audio_cst1Td[0u] = CY_DMA_INVALID_TD;
        Audio_cst1Td[1u] = CY_DMA_INVALID_TD;
    }
#endif /* 0u != Audio_MANAGED_DMA */


/***************************************************************************
*****
* Function Name: Audio_Init
****************************************************************************
*****
*
* Summary:
*  Initializes the customizer settings for the component including channel
*  status.
*
* Parameters:
*  None.
*
* Return:
*  None.
*
* Global Variables:
*  Audio_wrkCstStream0[] - Channel 0 Status internal buffer. Modified
*     when  default  S/PDIF  configuration  provided  with  customizer  is
initialized or
*  restored.
*  Audio_wrkCstStream1[] - Channel 1 Status internal buffer. Modified
*     when  default  S/PDIF  configuration  provided  with  customizer  is
initialized or
*  restored.
*
* Reentrant:
*  No.
*
****************************************************************************
****/
void Audio_Init(void)
{
    #if(0u != Audio_MANAGED_DMA)
        /* Channel status set by user in the customizer. Used to initialize
the
        *  settings in Audio_Init() API.
        */
        static const uint8 CYCODE Audio_initCstStream0[Audio_CST_LENGTH]  =
{

0x00,0x00,0x00,0x01,0x0B,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
        };
        static const uint8 CYCODE Audio_initCstStream1[Audio_CST_LENGTH]  =
{
            0x00u, 0x00u, 0x11u, 0x01u, 0x0Bu, 0x00u, 0x00u, 0x00u, 0x00u,
0x00u,  0x00u,  0x00u,  0x00u,  0x00u,  0x00u,  0x00u,  0x00u,  0x00u,  0x00u,
0x00u, 0x00u, 0x00u, 0x00u, 0x00u
        };
    #endif /* (0u != Audio_MANAGED_DMA) */

    uint8 enableInterrupts;

    enableInterrupts = CyEnterCriticalSection();
    /* Set FIFOs in the Single Buffer Mode */
    Audio_FCNT_AUX_CTL_REG     |= Audio_FX_CLEAR;
    Audio_PREGEN_AUX_CTL_REG   |= Audio_FX_CLEAR;
    CyExitCriticalSection(enableInterrupts);

    /* Channel status ISR initialization  */
    #if(0u != Audio_MANAGED_DMA)
        CyIntDisable(Audio_CST_0_ISR_NUMBER);
        CyIntDisable(Audio_CST_1_ISR_NUMBER);

        /* Set the ISR to point to the Interrupt processing routines */
        (void) CyIntSetVector(Audio_CST_0_ISR_NUMBER, &Audio_Cst0Copy);
        (void) CyIntSetVector(Audio_CST_1_ISR_NUMBER, &Audio_Cst1Copy);

        /* Set the priority */
        CyIntSetPriority(Audio_CST_0_ISR_NUMBER, Audio_CST_0_ISR_PRIORITY);
        CyIntSetPriority(Audio_CST_1_ISR_NUMBER, Audio_CST_1_ISR_PRIORITY);
    #endif /* (0u != Audio_MANAGED_DMA) */

    /* Setup Frame and Block Intervals */
    /* Frame Period */
    Audio_FRAME_PERIOD_REG = Audio_FRAME_PERIOD;
    /* Preamble and Post Data Period */
    Audio_FCNT_PRE_POST_REG = Audio_PRE_POST_PERIOD;
    /* Audio Sample Word Length */
    Audio_FCNT_AUDIO_LENGTH_REG = Audio_AUDIO_DATA_PERIOD;
    /* Number of frames in block */
    Audio_FCNT_BLOCK_PERIOD_REG = Audio_BLOCK_PERIOD;

    /* Set Preamble Patterns */
    Audio_PREGEN_PREX_PTRN_REG = Audio_PREAMBLE_X_PATTERN;
    Audio_PREGEN_PREY_PTRN_REG = Audio_PREAMBLE_Y_PATTERN;
    Audio_PREGEN_PREZ_PTRN_REG = Audio_PREAMBLE_Z_PATTERN;

    /* Set Interrupt Mask. By default interrupt generation is allowed only
for
    *  error conditions, including audio or channel status FIFOs underflow.
    */
    Audio_STATUS_MASK_REG = Audio_DEFAULT_INT_SRC;

    /* Channel Status DMA Config */
    #if(0u != Audio_MANAGED_DMA)
        /* Init channel status streams */
        (void) memcpy((void *) Audio_wrkCstStream0,
                      (void *) Audio_initCstStream0, Audio_CST_LENGTH);

        (void) memcpy((void *) Audio_wrkCstStream1,
                      (void *) Audio_initCstStream1, Audio_CST_LENGTH);

        /* Init DMA, 1 byte bursts, each burst requires a request */
        Audio_cst0Chan = Audio_Cst0_DMA_DmaInitialize(
            Audio_CST_DMA_BYTES_PER_BURST, Audio_CST_DMA_REQUEST_PER_BURST,
            HI16(Audio_CST_DMA_SRC_BASE), HI16(Audio_CST_DMA_DST_BASE));

        Audio_cst1Chan = Audio_Cst1_DMA_DmaInitialize(
            Audio_CST_DMA_BYTES_PER_BURST, Audio_CST_DMA_REQUEST_PER_BURST,
            HI16(Audio_CST_DMA_SRC_BASE), HI16(Audio_CST_DMA_DST_BASE));
    #endif /* (0u != Audio_MANAGED_DMA) */
}


/***************************************************************************
*****
* Function Name: Audio_Start
****************************************************************************
*****
*
* Summary:
*  Starts the S/PDIF interface.
*
* Parameters:
*  None.
*
* Return:
*  None.
*
* Global Variables:
*  Audio_initVar - used to check initial configuration, modified on
*  first function call.
*
* Reentrant:
*  No.
*
****************************************************************************
****/
void Audio_Start(void)
{
    if(0u == Audio_initVar)
    {
        Audio_Init();
        Audio_initVar = 1u;
    }

    Audio_Enable();
}


/***************************************************************************
*****
* Function Name: Audio_Stop
****************************************************************************
*****
*
* Summary:
*  Disables the S/PDIF interface. The audio data and channel data FIFOs are
*   cleared. If the component is configured to manage channel status DMA,
then
*  the DMA channels and TDs are released to the system.
*
* Parameters:
*  None.
*
* Return:
*  None.
*
* Reentrant:
*  No.
```

```
*
****************************************************************************
****/
void Audio_Stop(void)
{
    uint8 enableInterrupts;

    /* Disable audio data transmission */
    Audio_DisableTx();

    Audio_CONTROL_REG &= ((uint8) ~Audio_ENBL);

    enableInterrupts = CyEnterCriticalSection();
    Audio_STATUS_AUX_CTL_REG  &=  ((uint8)  ~Audio_INT_EN);    /*  Disable
Interrupt generation */
    Audio_BCNT_AUX_CTL_REG    &=  ((uint8)  ~Audio_BCNT_EN);  /*  Disable  Bit
counter */
    CyExitCriticalSection(enableInterrupts);

    #if (0u != Audio_MANAGED_DMA)
        /* Disable channel status ISRs */
        CyIntDisable(Audio_CST_0_ISR_NUMBER);
        CyIntDisable(Audio_CST_1_ISR_NUMBER);

        CyIntClearPending(Audio_CST_0_ISR_NUMBER);
        CyIntClearPending(Audio_CST_1_ISR_NUMBER);

        /* Clear the buffer offset variables */
        Audio_cst0BufOffset = 0u;
        Audio_cst1BufOffset = 0u;
        Audio_CstDmaRelease();
    #endif /* 0u != Audio_MANAGED_DMA */

    Audio_ClearTxFIFO();
    Audio_ClearCstFIFO();
}


/***************************************************************************
*****
* Function Name: Audio_EnableTx
****************************************************************************
*****
*
* Summary:
*   Enables  the  audio  data  output  in  the  S/PDIF  bit  stream.  Transmission
will
*  begin at the next X or Z frame.
*
* Parameters:
*  None.
*
* Return:
*  None.
*
****************************************************************************
****/
void Audio_EnableTx(void)
{
    Audio_CONTROL_REG |= Audio_TX_EN;
}
```

```
/**************************************************************************
*****
* Function Name: Audio_DisableTx
**************************************************************************
*****
*
* Summary:
*  Disables the Tx direction of the the audio output S/PDIF bit stream.
*  Transmission of data will stop at the next rising edge of the clock and
a
*  constant 0 value will be transmitted.
*
* Parameters:
*  None.
*
* Return:
*  None.
*
**************************************************************************
****/
void Audio_DisableTx(void)
{
    Audio_CONTROL_REG &= ((uint8) ~Audio_TX_EN);
}


/**************************************************************************
*****
* Function Name: Audio_SetInterruptMode
**************************************************************************
*****
*
* Summary:
*  Sets the interrupt source for the S/PDIF. Multiple sources may be ORed
*  together.
*
* Parameters:
*  Byte containing the constant for the selected interrupt sources.
*   Audio_AUDIO_FIFO_UNDERFLOW
*   Audio_AUDIO_0_FIFO_NOT_FULL
*   Audio_AUDIO_1_FIFO_NOT_FULL
*   Audio_CHST_FIFO_UNDERFLOW
*   Audio_CHST_0_FIFO_NOT_FULL
*   Audio_CHST_1_FIFO_NOT_FULL
*
* Return:
*  None.
*
**************************************************************************
****/
void Audio_SetInterruptMode(uint8 interruptSource)
{
    Audio_STATUS_MASK_REG = interruptSource;
}


/**************************************************************************
*****
* Function Name: Audio_ReadStatus
```

```
****************************************************************************
*****
*
* Summary:
*  Returns state in the SPDIF status register.
*
* Parameters:
*  None.
*
* Return:
*  State of the SPDIF status register
*   Audio_AUDIO_FIFO_UNDERFLOW (Clear on Read)
*   Audio_AUDIO_0_FIFO_NOT_FULL
*   Audio_AUDIO_1_FIFO_NOT_FULL
*   Audio_CHST_FIFO_UNDERFLOW (Clear on Read)
*   Audio_CHST_0_FIFO_NOT_FULL
*   Audio_CHST_1_FIFO_NOT_FUL
*
* Side Effects:
*  Clears the bits of SPDIF status register that are Clear on Read.
*
****************************************************************************
****/
uint8 Audio_ReadStatus(void)
{
    return(Audio_STATUS_REG & Audio_INT_MASK);
}


/***************************************************************************
*****
* Function Name: Audio_WriteTxByte
****************************************************************************
*****
*
* Summary:
*  Writes a single byte into the specified Audio FIFO.
*
* Parameters:
*  wrData: Byte containing the data to transmit.
*  channelSelect: Byte containing the constant for Channel to write.
*    Audio_CHANNEL_0 indicates to write to the Channel 0 and
*    Audio_CHANNEL_1 indicates to write to the Channel 1.
*  In the interleaved mode this parameter is ignored.
*
* Return:
*  None.
*
* Reentrant:
*  No.
*
****************************************************************************
****/
void Audio_WriteTxByte(uint8 wrData, uint8 channelSelect)
{
    #if(0u != Audio_DATA_INTERLEAVING)

        if(0u != channelSelect)
        {
            /* Suppress compiler warning */
        }
        Audio_TX_FIFO_0_REG = wrData;

    #else

        if(Audio_CHANNEL_0 == channelSelect)
        {
            Audio_TX_FIFO_0_REG = wrData;
        }
        else
        {
            Audio_TX_FIFO_1_REG = wrData;
        }

    #endif /* (0u != Audio_DATA_INTERLEAVING) */
}


/*************************************************************************
*****
* Function Name: Audio_ClearTxFIFO
*************************************************************************
*****
*
* Summary:
*  Clears out the Tx FIFO. Any data present in the FIFO will not be sent.
This
*  call should be made only when transmit is disabled. In the case of
separated
*  audio mode, both audio FIFOs will be cleared.
*
* Parameters:
*  None.
*
* Return:
*  None.
*
*************************************************************************
****/
void Audio_ClearTxFIFO(void)
{
    uint8 enableInterrupts;

    enableInterrupts = CyEnterCriticalSection();
    Audio_TX_AUX_CTL_REG |= ((uint8)  Audio_FX_CLEAR);
    Audio_TX_AUX_CTL_REG &= ((uint8) ~Audio_FX_CLEAR);
    CyExitCriticalSection(enableInterrupts);
}


/*************************************************************************
*****
* Function Name: Audio_WriteCstByte
*************************************************************************
*****
*
* Summary:
*  Writes a single byte into the specified Channel Status FIFO.
*
* Parameters:
*  wrData: Byte containing the status data to transmit.
*  channelSelect: Byte containing the constant for Channel to write.
*    Audio_CHANNEL_0 indicates to write to the Channel 0 and
*    Audio_CHANNEL_1 indicates to write to the Channel 1.
*
* Return:
*  None.
*
* Reentrant:
*  No.
*
***************************************************************************
****/
void Audio_WriteCstByte(uint8 wrData, uint8 channelSelect)
{
    if(Audio_CHANNEL_0 == channelSelect)
    {
        Audio_CST_FIFO_0_REG = wrData;
    }
    else
    {
        Audio_CST_FIFO_1_REG = wrData;
    }
}


/**************************************************************************
*****
* Function Name: Audio_ClearCstFIFO
***************************************************************************
*****
*
* Summary:
*   Clears out the Channel Status FIFOs. Any data present in either FIFO
will not
*  be sent. This call should be made only when the component is stopped.
*
* Parameters:
*  None.
*
* Return:
*  None.
*
***************************************************************************
****/
void Audio_ClearCstFIFO(void)
{
    uint8 enableInterrupts;

    enableInterrupts = CyEnterCriticalSection();
    Audio_CST_AUX_CTL_REG |= ((uint8)  Audio_FX_CLEAR);
    Audio_CST_AUX_CTL_REG &= ((uint8) ~Audio_FX_CLEAR);
    CyExitCriticalSection(enableInterrupts);
}


#if(0u != Audio_MANAGED_DMA)

/**************************************************************************
*****
    * Function Name: Audio_SetChannelStatus
```

```
****************************************************************************
*****
    *
    * Summary:
    *  Sets the values of the channel status at run time. This API is only
valid
    *  when the component is managing the DMA.
    *
    * Parameters:
    *  channel: Byte containing the constant for Channel to modify.
    *   Audio_CHANNEL_0 and Audio_CHANNEL_1 are used to
    *   specify Channel 0 and Channel 1 respectively.
    *  byte : Byte to modify. This argument should be in range from 0 to
23.
    *  mask : Mask on the byte.
    *  value: Value to set.
    *
    * Return:
    *  None.
    *
    * Reentrant:
    *  No.
    *

****************************************************************************
****/
    void Audio_SetChannelStatus(uint8 channel, uint8 byte, uint8 mask,
uint8 value) \

    {
        if(Audio_CHANNEL_0 == channel)
        {
            /* Update of status stream needs to be atomic */
            CyIntDisable(Audio_CST_0_ISR_NUMBER);
            Audio_wrkCstStream0[byte] &= ((uint8) ~mask);     /* Clear the
applicable bits */
            Audio_wrkCstStream0[byte] |= ((uint8) value);      /* Set the
applicable bits  */
            CyIntEnable(Audio_CST_0_ISR_NUMBER);
        }
        else
        {
            /* Update of status stream needs to be atomic */
            CyIntDisable(Audio_CST_1_ISR_NUMBER);
            Audio_wrkCstStream1[byte] &= ((uint8) ~mask);     /* Clear the
applicable bits */
            Audio_wrkCstStream1[byte] |= ((uint8) value);      /* Set the
applicable bits  */
            CyIntEnable(Audio_CST_1_ISR_NUMBER);
        }
    }



/***************************************************************************
*****
    * Function Name: Audio_SetFrequency

****************************************************************************
*****
```

```
    *
    * Summary:
    *  Sets the values of the channel status for a specified frequency and
returns
    *  1. This function only works if the component is stopped. If this is
called
    *   while the component is started, a zero will be returned and the
values will
    *   not be modified. This API is only valid when the component is
managing the
    *   DMA.
    *
    * Parameters:
    *  Byte containing the constant for the specified frequency.
    *    Audio_SPS_UNKNOWN
    *    Audio_SPS_22KHZ
    *    Audio_SPS_24KHZ
    *    Audio_SPS_32KHZ
    *    Audio_SPS_44KHZ
    *    Audio_SPS_48KHZ
    *    Audio_SPS_64KHZ
    *    Audio_SPS_88KHZ
    *    Audio_SPS_96KHZ
    *    Audio_SPS_192KHZ
    *
    * Return:
    *  1 on success.
    *  0 on failure.
    *

****************************************************************************
****/
    uint8 Audio_SetFrequency(uint8 frequency)
    {
        uint8 result;

        result = ((uint8) Audio_IS_DISABLED);

        /* The values of the channel status should not be modified if the
component is started */
        if(0u != result)
        {
            /* Refer to sample frequency constansts: Audio_SF_freqKHZ  (3u)
(0xCFu) (freq) */
            Audio_SetChannelStatus(Audio_CHANNEL_0, 3u, 0xCFu, frequency);
            Audio_SetChannelStatus(Audio_CHANNEL_1, 3u, 0xCFu, frequency);
        }

        return(result);
    }
#endif /* (0u != Audio_MANAGED_DMA) */


/* [] END OF FILE */
```