Resolución Final Informática 2

```
/*****************************************************************************//**
 * main(): Aplicacion que resuelve el Final de Informatica 2 de Fecha 18-12-2014
*******************************************************************************/
int main(void)
{
    /* Inicializaciones de HW … */

    /* Lazo principal */
    while(1) {

        if(Timer_Paso1MS() == SI) { /* Verifica si paso 1 milisegundo */
            /* Verifica si el Boton fue presionado, y en caso de serlo activa
             * el sistema sonoro para que comienze a funcionar en proximo ROJO
             */
            if(Boton_FuePulsado() == SI)  {
                Semaforo_ActivarSistemaSonoro();
            }

            /* Maquina de Estados que controla el funcionamiento del Semaforo */
            Semaforo_ControlDeLuces();

            /* Funciones que necesitan ser invocadas periodicamente para
             * el correcto funcionamiento de cada modulo            */
            Beeps_InvocarCada1MS();
            Display_InvocarCada1MS();
            Boton_InvocarCada1MS();
        }
    }
    return 0 ;
}

/*****************************************************************************************************
*** MODULO SEMAFORO.C
*****************************************************************************************************/

*** PRIVATE DATA TYPES
/* Enumeracion que indica los distintos estados del Semaforo */
typedef enum {LUZ_ROJA_ENCENDIDA, LUZ_AMARILLA_ENCENDIDA, LUZ_VERDE_ENCENDIDA} EstadoDelSemaforo;

/* Enumeracion que indica los distintos estados de la Luz Roja */
typedef enum {TRANSCURRIENDO_TR1, TRANSCURRIENDO_TR2, TRANSCURRIENDO_TR3} SubEstadosDeLaLuzRoja;

/* Enumeracion que indica las distintas luces del Semaforo */
typedef enum {VERDE, AMARILLA, ROJA, PEATON_CAMINANDO, PEATON_ESPERANDO} LuzDelSemaforo;

/* Enumeracion que indica los distintos estados de la Recepcion de la Trama de Configuracion */
typedef enum { START_TAG,
               TR1_TAG, TR1_VALOR, TR2_TAG, TR2_VALOR, TR3_TAG, TR3_VALOR,
               TA_TAG, TA_VALOR,
               TV_TAG, TV_VALOR,
               END_TAG
             } EstadoDeLaTrama;

/* Enumeracion que indica los distintos estados del Sistema Sonoro */
typedef enum {APAGADO, REQUERIDO, ENCENDIDO} EstadoDelSistemaSonoro;

/* Estrucutura de Datos para guardar la Configuracion de los Tiempos del Semaforo */
typedef struct
{
    uint8_t tiempoTr1;
    uint8_t tiempoTr2;
    uint8_t tiempoTr3;
    uint8_t tiempoTa;
    uint8_t tiempoTv;
    uint8_t fueConfigurado;

}SEM_CFG_TypeDef;

*** PRIVATE TABLES
static const GPIO_Cfg Luz[] = { {2, 0}, {2, 1}, {2, 2}, {2, 3}, {2, 4} }; /* Puerto y Pin de cada Luz */

*** PRIVATE GLOBAL VARIABLES
static SEM_CFG_TypeDef SemaforoConfig = {0, 0, 0, 0, 0, NO}; /* Configuracion del Semaforo */
static uint32_t tickSemaforo = 0;
static EstadoDelSistemaSonoro sistemaSonoro = APAGADO;

*** PUBLIC FUNCTIONS
/*****************************************************************************//**
 * Semaforo_ActivarSistemaSonoro(): Activa el Sistema Sonoro para personas no videntes, reflejando el estado del semaforo por
 * todo un ciclo completo. Los beeps comienzan a emitirse desde la siguiente luz roja, y al finalizar el ciclo
 * rojo-amarillo-verde-amarillo se desactiva automaticamente.
*******************************************************************************/
void Semaforo_ActivarSistemaSonoro(void){
    sistemaSonoro = REQUERIDO; /* Fue requerido y será encendido en el proximo ROJO */
}
```

```
*** Semaforo_ControlDeLuces(): Maquina de Estado que controla las luces del Semaforo ***
void Semaforo_ControlDeLuces(void)
{
    static EstadoDelSemaforo estado = LUZ_ROJA_ENCENDIDA;
    static EstadoDelSemaforo estadoAnterior;
    static SubEstadosDeLaLuzRoja subEstado;

    /* Si el Display está encendido, se actualiza con la cuenta regresiva cada 1 segundo */
    if ((Display_GetEstado() == DISPLAY_ON) && (tickSemaforo % 1000))
        Display_ActualizarValor(tickSemaforo / 1000);

    if(tickSemaforo != 0) { /* Verifica si est tiempo de procesar, es decir, que tickSemaforo == 0, sino retorna */
        tickSemaforo--;
        return;
    }

    switch (estado) { /* Se venció el tiempo cargado (tickSemaforo == 0), se procesa la Maq Est */
        case LUZ_VERDE_ENCENDIDA:
            Luz_Off(VERDE);                             /* LUCES: Cambia de Verde a Amarilla */
            Luz_On(AMARILLA);
            tickSemaforo = SemaforoConfig.tiempoTa;
            estado         = LUZ_AMARILLA_ENCENDIDA;
            estadoAnterior = LUZ_VERDE_ENCENDIDA;
            break;

        case LUZ_AMARILLA_ENCENDIDA:
            Luz_Off(AMARILLA);
            if(estadoAnterior == LUZ_ROJA_ENCENDIDA) {
                Luz_On(VERDE);                          /* LUCES: Cambia de Amarilla a Verde */
                tickSemaforo = SemaforoConfig.tiempoTv;
                estado         = LUZ_VERDE_ENCENDIDA;

                if(sistemaSonoro == ENCENDIDO)
                    Beeps_On(TRES_BEEPS_X_SEG);
            }
            else if(estadoAnterior == LUZ_VERDE_ENCENDIDA)  {
                Luz_On(ROJA);                           /* LUCES: Cambia de Amarilla a Roja */
                tickSemaforo = SemaforoConfig.tiempoTr1;
                estado         = LUZ_ROJA_ENCENDIDA;
                Luz_Off(PEATON_ESPERANDO);                  /* SILUETA: Cambia de Esperando a Caminando */
                Luz_On(PEATON_CAMINANDO);

                if(sistemaSonoro == REQUERIDO) {
                    sistemaSonoro = ENCENDIDO;
                    Beeps_On(UN_BEEP_X_SEG);
                }
                else if(sistemaSonoro == ENCENDIDO)
                    Beeps_Off();
            }

            estadoAnterior = LUZ_AMARILLA_ENCENDIDA;
            break;

        case LUZ_ROJA_ENCENDIDA:
            switch (subEstado) {
                case TRANSCURRIENDO_TR1:
                    subEstado = TRANSCURRIENDO_TR2;
                    tickSemaforo = SemaforoConfig.tiempoTr2;
                    Display_SetEstado(ON); /* CUENTA REGRESIVA: Comienza */
                    Display_ActualizarValor(tickSemaforo / 1000);
                    Luz_Off(PEATON_CAMINANDO);              /* SILUETA: Cambia de Caminando a Esperando */
                    Luz_On(PEATON_ESPERANDO);
                    break;

                case TRANSCURRIENDO_TR2:
                    subEstado = TRANSCURRIENDO_TR3;
                    tickSemaforo = SemaforoConfig.tiempoTr3;
                    Display_SetEstado(OFF); /* CUENTA REGRESIVA: Finaliza */
                    break;

                case TRANSCURRIENDO_TR3:
                    subEstado = TRANSCURRIENDO_TR1;
                    tickSemaforo = SemaforoConfig.tiempoTa;

                    Luz_Off(ROJA);                      /* LUCES: Cambia de Roja a Amarilla */
                    Luz_On(AMARILLA);

                    estado         = LUZ_AMARILLA_ENCENDIDA;
                    estadoAnterior = LUZ_ROJA_ENCENDIDA;

                    if(sistemaSonoro == ENCENDIDO)
                        Beeps_On(DOS_BEEPS_X_SEG);
                    break;
            }
            break;
    }
}
```

```c
*** Semaforo_ProcesarDatoRecibido(): Debe ser invocada cada vez que se recibe un dato por el Puerto Serie. ***
void Semaforo_ProcesarDatoRecibido(uint8_t dato)
{
    static EstadoDeLaTrama estado = START_TAG;

    /* Si el Semaforo ya fue configurado, retorna directamente sin procesar datos */
    if(SemaforoConfig.fueConfigurado == SI)
        return;

    switch(estado) {
        case START_TAG:
            if( dato == START)        /* Verifica que el dato sea el comienzo de trama */
                estado = TR1_TAG;
            break;

        case TR1_TAG:
            /* Verifica que el dato sea "TR1" y pasa al siguiente estado, sino vuelve a START_TAG */
            estado = (dato == TR1) ? TR1_VALOR : START_TAG;
            break;

        case TR1_VALOR:
            /* Actualiza la configuración del Semaforo con el dato recibido y pasa al siguiente estado */
            SemaforoConfig.tiempoTr1 = dato;
            estado = TR2_TAG;
            break;

        case TR2_TAG:
            /* Verifica que el dato sea "TR2" y pasa al siguiente estado, sino vuelve a START_TAG */
            estado = (dato == TR2) ? TR2_VALOR : START_TAG;
            break;

        case TR2_VALOR:
            /* Actualiza la configuración del Semaforo con el dato recibido y pasa al siguiente estado */
            SemaforoConfig.tiempoTr2 = dato;
            estado = TR3_TAG;
            break;

        case TR3_TAG:
            /* Verifica que el dato sea "TR3" y pasa al siguiente estado, sino vuelve a START_TAG */
            estado = (dato == TR3) ? TR3_VALOR : START_TAG;
            break;

        case TR3_VALOR:
            /* Actualiza la configuración del Semaforo con el dato recibido y pasa al siguiente estado */
            SemaforoConfig.tiempoTr3 = dato;
            estado = TA_TAG;
            break;

        case TA_TAG:
            /* Verifica que el dato sea "TA" y pasa al siguiente estado, sino vuelve a START_TAG */
            estado = (dato == TA) ? TA_VALOR : START_TAG;
            break;

        case TA_VALOR:
            /* Actualiza la configuración del Semaforo con el dato recibido y pasa al siguiente estado */
            SemaforoConfig.tiempoTa = dato;
            estado = TV_TAG;
            break;

        case TV_TAG:
            /* Verifica que el dato sea "TV" y pasa al siguiente estado, sino vuelve a START_TAG */
            estado = (dato == TV) ? TV_VALOR : START_TAG;
            break;

        case TV_VALOR:
            /* Actualiza la configuración del Semaforo con el dato recibido y pasa al siguiente estado */
            SemaforoConfig.tiempoTv = dato;
            estado = END_TAG;
            break;

        case END_TAG:
            /* Verifica que el dato sea "END" y finaliza la configuración */
            if(dato == END)
                SemaforoConfig.fueConfigurado = SI;

            estado = START_TAG;
            break;
    }
}

*** PRIVATE FUNCTIONS
static void Luz_Off(LuzDelSemaforo luz) {
    GPIO_ClrPin(Luz[luz].puerto, Luz[luz].pin); /* Apaga la luz correspondiente segun puerto y pin */
}

static void Luz_On(LuzDelSemaforo luz) {
    GPIO_SetPin(Luz[luz].puerto, Luz[luz].pin); /* Enciende la luz correspondiente segun puerto y pin */
}
```

```c
/*****************************************************************************************************
*** MODULO BEEPS.C
*****************************************************************************************************/

/*****************************************************************************************************
*** PRIVATE GLOBAL VARIABLES
*****************************************************************************************************/
static uint8_t beepsAejecutar;
static uint8_t beepsEjecutados;
static uint32_t tick;
static uint8_t estado;

/*****************************************************************************************************
*** PUBLIC FUNCTIONS
*****************************************************************************************************/
* Beeps_Off() Apaga el Buzzer permanentemente
void Beeps_Off(void)
{
    Buzzer_Off();
    estado = OFF;
}

* Beeps_On(): A partir del llamado de esta función, comienza a producir X cant de beeps por segundos, donde X es 1, 2 o 3.
void Beeps_On(TipoDeBeeps cantidadDeBeeps)
{
    beepsAejecutar  = cantidadDeBeeps;
    beepsEjecutados = 0;
    tick = 0;

    estado = ON;
}

* Beeps_InvocarCada1MS(): Se debe invocar cada 1 ms para el correcto funcionamiento del modulo. Maquina de estados que
* produce X cantidad de beeps por segundos, donde X es 1, 2 o 3. Solo se ejecuta si fue invocada Beeps_On()
void Beeps_InvocarCada1MS(void)
{
    uint8_t estadoDelBuzzer = 0;

    if(estado == OFF) /* Si el estado es OFF retorna directamente */
        return;

    if(tick != 0)
        tick--;

    estadoDelBuzzer = Buzzer_GetEstado(); /* Toma el estado actual del Buzzer (OFF o ON) */

    /* Ejecuta la Maquina de estados que produce X cantidad de Beeps x segundo */
    if(estadoDelBuzzer == OFF) {
        if(tick == 0)  {
            tick = 100;
            Buzzer_On();
            beepsEjecutados++;
        }
    }
    else if(estadoDelBuzzer == ON) {
        if(tick == 0) {
            Buzzer_Off();

            if(beepsEjecutados < beepsAejecutar) {
                tick = 100;
            }
            else if(beepsEjecutados == beepsAejecutar)  {
                /* Si: beepsAejecutar = 1 entonces el tiempo del buzzer en off debe ser 900 ms  (1 seg - 100 ms)
                 *     beepsAejecutar = 2 entonces el tiempo del buzzer en off debe ser 700 ms  (1 seg - 300 ms)
                 *     beepsAejecutar = 3 entonces el tiempo del buzzer en off debe ser 500 ms  (1 seg - 500 ms) */
                tick = 1100 - (beepsAejecutar * 200);
            }
        }
    }
}

*** PRIVATE FUNCTIONS
static void Buzzer_Off(void) {
    GPIO_ClrPin(2, 5); /* Apaga el buzzer correspondiente al puerto y pin pasado como parametro*/
}

static void Buzzer_On(void) {
    GPIO_SetPin(2, 5); /* Enciende el buzzer correspondiente al puerto y pin pasado como parametro*/
}

static uint8_t Buzzer_GetEstado(void) {
    return GPIO_GetPin(2, 5); /* Retorna el estado del Buzzer */
}
```

```c
/*****************************************************************************************************
*** MODULO BOTON.C
*****************************************************************************************************/

/*****************************************************************************************************
*** PRIVATE DEFINES
*****************************************************************************************************/
/* Estados para validar si el boton fue pulsado */
#define NO_PULSADO                0
#define VALIDAR_BOTON_PULSADO     1
#define PULSADO                   2

/* Posibles estados de la variable "boton" */
#define NO_FUE_PULSADO    0
#define FUE_PULSADO       1

/*****************************************************************************************************
*** PRIVATE GLOBAL VARIABLES
*****************************************************************************************************/
static uint8_t boton = NO_FUE_PULSADO;

/*****************************************************************************//**
 * @fn         Boton_FuePulsado()
 * @brief      Retorna si el boton fue pulsado (por mas de 200 ms)
*****************************************************************************/
uint8_t Boton_FuePulsado(void)
{
    uint8_t retorno = NO;

    if(boton == FUE_PULSADO)  {
        boton = NO_FUE_PULSADO;       /* Se informa que el boton fue pulsado */
        retorno = SI;
    }

    return retorno;
}

/*****************************************************************************//**
 * @fn         Boton_InvocarCada1MS()
 * @brief      Se debe invocar cada 1 ms para el correcto funcionamiento del modulo.
 *              Maquina de estados que produce valida si el boton fue pulsado, para
 *              ello debe permanecer 200 ms en estado estable.
*****************************************************************************/
void Boton_InvocarCada1MS(void)
{
    static uint8_t estadoBoton = NO_PULSADO;
    static uint8_t tick = 0;
    uint8_t estadoPin;

    estadoPin = GPIO_GetPin(0, 0);

    switch(estadoBoton)
    {
        case NO_PULSADO:
            if(estadoPin == ON) /* Se presiono el boton */
            {
                /* Para que el Boton se considere que fue pulsado, debe permanecer durante 200ms
                 * en un estado estable */
                estadoBoton = VALIDAR_BOTON_PULSADO;
                tick = 200;
            }
            break;

        case VALIDAR_BOTON_PULSADO:
            if(estadoPin == ON)
            {
                if(--tick == 0)
                {
                    estadoBoton = PULSADO;
                    boton = FUE_PULSADO;  /* Se informa que el Boton fue pulsado */
                }
            }
            else
            {
                estadoBoton = NO_PULSADO; /* Si se libera antes de los 200ms vuelve a no pulsado */
            }
            break;

        case PULSADO:
            if(estadoPin == OFF)
            {
                estadoBoton = NO_PULSADO;
            }
            break;
    }
}
```