Proyecto Final de *Software en Tiempo Real*

Sistema con sensor controlador por una línea serial

Luis A. Guanuco

Febrero 2014

Resumen

El presente trabajo se enfocada en la implementación de un sistema de comunicación sobre una red serial donde se encuentran conectados un sensor, un sistema electrónico con un microcontrolador y por último se conecta una computadora personal (PC). Los sistemas electrónicos basados en un procesador poseen un Sistema Operativo en Tiempo Real, RTOS por sus siglas en inglés (*Real-Time Operating Systems*). El sensor presenta una particularidad, y es que tanto la alimentación del transductor como el canal de datos es el mismo. Los parámetros principales en el diseño son definidos por el circuito del sensor.

1. Introducción

La posibilidad de reducir el costo en cualquier sistema electrónico es uno de los grandes retos con el que se presenta el desarrollador electrónico. Si bien hay un gran avance en los procesos de fabricación de circuitos integrados, existen dos inconvenientes para hacerse de dicha tecnología. El primero es el *acceso* a los nuevos dispositivos electrónicos que se encuentran en el mercado. El segundo inconveniente que se podría enumerar, muy relacionado al primero, es el *costo* que tienen estos nuevos dispositivos. Obviamente que estos inconvenientes planteados son desde un perfil estudiantil. Seguramente en desarrollos industriales los factores de disponibilidad y costo son evaluados en función de las prestaciones que ofrece el dispositivo en cuestión. Pero como nuestro trabajo se encuentra orientado a un ámbito académico se priorizará el diseño que requiera mayor tiempo de investigación y desarrollo para obtener un sistema que cumpla con nuestros requerimientos a un costo reducido y que pueda ser reproducible fácilmente.

El procesamiento de datos se realizará sobre un dispositivo microcontrolador dsPIC® fabricado por Microchip Inc.. En este μC se encuentra embebido un RTOS donde se implementan varias tareas para el sistema operativo (OS). Debido a la complejidad del desarrollo, a nivel de *software*, no se presentan problemas en los tiempos de procesamiento del OS. Como se dijo en el Resumen del informe, las principales especificaciones son propias del sistema transductor (sensor).

La información relevada por el sistema embebido es transmitida a una PC, quién presentará la información obtenida del sensor. Siguiendo la línea del uso sistemas operativos en tiempo real, se implementa nuevamente un RTOS. Al igual que en el caso del dsPIC®, aquí se adaptan la capa más baja del código máquina para compilar el sistema operativo con la arquitectura a utilizar. La mayoría de la información necesaria para esta *adaptación* se encuentra disponible por el desarrollador del RTOS (Micriμm Inc.).

2. Especificaciones del diseño

Las especificaciones del desarrollo se las puede dividir en dos grupos. Por un lado especificaciones de *Hardware*, aquí se encuentra tanto el circuito del sensor como así también la plataforma de procesamiento y los recursos disponibles. Por otro lado se especifica el *Software* que se debe implementar, el sistema operativo en tiempo real μC/OS-II™. Ya familiarizado por el desarrollador del proyecto por ser el OS utilizado en la *Cátedra de Software en Tiempo Real* dictado como materia electiva en la Carrera Ingeniería Electrónica de la Universidad Tecnológica Nacional – Facultad Regional Córdoba.

2.1. *Hardware*

El esquema general del *hardware* se puede ver en la Fig. 1. Aquí se especifica el flujo de dato entre los diferentes bloques. Sobre las líneas de comunicación se tiene comentado que información dispone físicamente cada conexión. En la conexión entre el sensor y el dsPIC® se no solo se comparte información (datos) sino también se transmite energía para el circuito transductor. Entre el dsPIC® y la PC se implementa una comunicación RS-232.

2.2. *Software*

Como se mencionó anteriormente, tanto en el dsPIC® como en el PC se implementan el sistema operativo μC/OS-II™. Si bien las especificaciones a nivel *hardware* no demanden el uso de un RTOS, ya que los requerimientos en la respuesta del procesamiento no son críticas, se hará uso de estos sistemas operativos como una aplicación de las características mas destacadas con la que cuentan los RTOS (μC/OS-II™en nuestro caso).

En las secciones siguientes se describirá con mayor detalle tanto el *hardware* como el *software*.

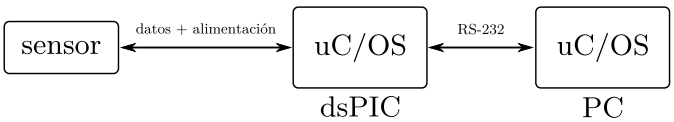


Figura 1: Esquema general de la implementación.

3. Implementación del circuito electrónico

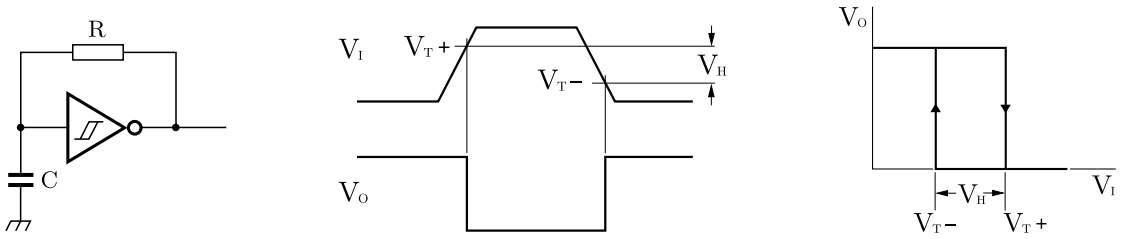
A nivel electrónico, el desarrollo se divide en dos diseños. El primero es el circuito de sensor y el segundo es el sistema embebido con el dsPIC® como dispositivo principal.

3.1. Circuito del sensor

La base del circuito transductor es un oscilador forma por un inversor digital (IC) y una red RC que determina la frecuencia de oscilación del oscilador. Para mantener más estable la frecuencia del oscilador se utiliza un inversor *Schmitt-Trigger*. Esto último se debe a que el sistema sensor presentará algunas perturbaciones en el nivel de la tensión de alimentación, causa que se abordará más adelante en esta sección. La transducción del fenómeno físico que se quiere adquirir se producirá afectado la red RC, es decir la frecuencia del oscilador. Por la disponibilidad de transductores discretos, se opta por usar un sensor fotoeléctrico ó LDR (por sus siglas en inglés *Light Dependant Resistor*).

3.1.1. Oscilador *Schmitt-Trigger*

El oscilador basado con un inversor es un circuito muy sencillo que demanda el uso de una red formada por un resistor y un capacitor, ilustrado en la Figura 2a. El inversor utilizado en este caso es el **MM74HCT14**[1], el cual dispone de ocho inversores con *Schmitt-Trigger*. El concepto sobre de *Schmitt-Trigger* fue inventado por el científico estadounidense *Otto H. Schmitt* en el año 1934. Se lo llama *Trigger* pues la salida del circuito retiene su estado hasta que la entrada presenta un valor suficiente para disparar (en inglés *trigger*) en cambio de estado. En la Figura 2b se observa los niveles umbrales de tensión (*threshold*) que presenta este tipo de inversor. El comportamiento dinámico de ambos niveles umbrales se lo puede graficar como un ciclo de histéresis (Figura 2c), este comportamiento permite decir el *Schmitt-Trigger* posee *memoria* y puede actuar como un circuito biestable (latch o flip-flop). Aunque su principal uso es el acondicionamiento de señales digitales, permitiendo remover el ruido en la señal.



(a) Oscilador (b) Diagrama de tiempo (c) Histéresis

Figura 2: Circuito con *Schmitt-Trigger*.

La frecuencia de oscilación del circuito implementado en la Figura 2a es determinado por el valor de los componentes pasivos (*RC*), el nivel de la tensión de alimentación V_{CC}, y los niveles umbrales V_{T+} y V_{T-}. La ecuación (1) se obtuvo desde una nota de aplicación del integrado a utilizar.

[1]Se puede utilizar cualquier otro integrado con inversores pero debe contar con la tecnología *Schmitt-Trigger*.

$$\begin{aligned} f &\approx \frac{1}{RC\ln\left[\frac{V_{T+}(V_{CC}-V_{T-})}{V_{T-}(V_{CC}-V_{T+})}\right]} \\ f &\approx \frac{1}{RCk} \end{aligned} \tag{1}$$

La ecuación (1) demuestra que puede considerarse como una constante k la operación logarítmica para facilitar la estimación de la frecuencia a la que oscila el circuito con el inversor. Se podría representar el comportamiento del valor constante k como una curva donde se considerará variaciones en el valor de la tensión de alimentación V_{CC}. La Figura 3 demuestra que es posible mantener estable la frecuencia aun con variaciones significativas en la tensión de alimentación. Más adelante se especificará como influye esto en nuestro caso, sobre todo por la particularidad de que nuestro sistema será sometido a perturbaciones intencionales en el nivel de alimentación.

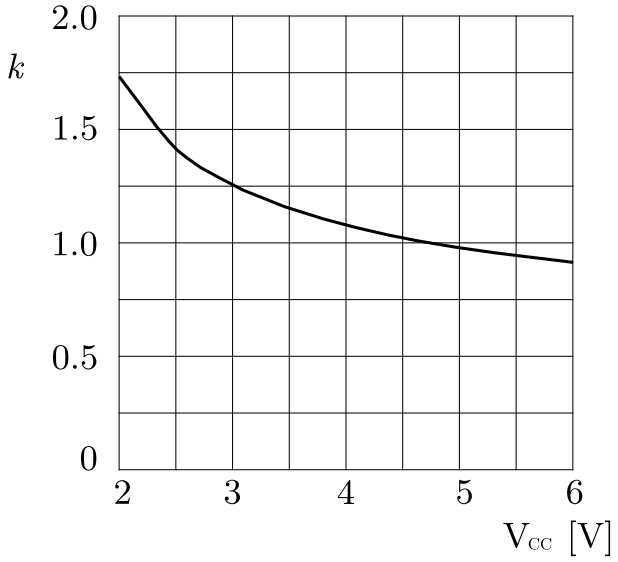


Figura 3: Curva V_{CC} vs. k.

3.1.2. Funcionamiento básico

En la sección anterior se describe el funcionamiento del oscilador basado en un circuito con un inversor. En este circuito la frecuencia de oscilación, ecuación (1), es inversamente proporcional al producto RCk. Aquí vamos a considerar que tanto C como k son constantes, el parámetro que determinará la frecuencia es el valor de R. El resistor R es en nuestro caso el transductor LDR. Este transductor varía su parámetro de resistividad en función de la cantidad de luz que incida sobre él. Un esquema del proceso de conversión de variables se puede ver en la Figura 4.

En función a lo anteriormente dicho, se debe adquirir y procesar una onda oscilante donde la información se encuentra en el frecuencia que dicha señal posee. Además se especifica que *no se considerarán variaciones bruscas* en la intensidad de la luz recibida por el LDR. Para determinar los rangos máximos y mínimos en la variación de la resistividad del LDR se realizaron algunas mediciones que se pueden ver en la Tabla 1.

Con la información del rengo resistivo del LDR se puede definir el rango de frecuencias de la señal que se obtendrá del sensor oscilador. Otra especificación a tener en

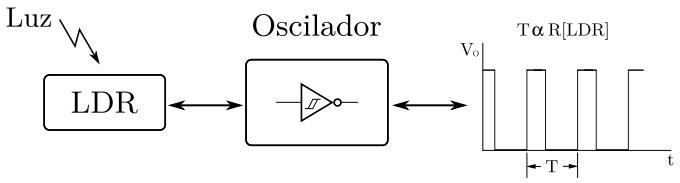


Figura 4: Proceso de conversión de variables en el sistema de censado.

Condición de Luz	Valor de resistividad
Oscuridad	180 KΩ
Claridad	1 KΩ

Tabla 1: Rango de variación del LDR.

cuenta es la frecuencia de muestreo que dispone el bloque siguiente al sensor, el microcontrolador dsPIC®. Aquí se tiene que el tiempo de muestreo es de 100 μS.. Según el teorema del muestreo de Nyquist-Shannon, se podría definir un límite en la frecuencia máxima posible a ser adquirida. El enunciado de este teorema dice,

$$F_S > 2F_{MÁX} \text{ , donde } F_{MÁX} \text{ es la frecuencia máxima de la señal.} \quad (2)$$

en función a esta definición, se puede encontrar cual será la frecuencia máxima que se detectará correctamente si la frecuencia de muestreo es de 100 μS. Es decir,

$$\begin{array}{lrcl} \text{sí} & F_S & > & 2F_{MÁX} \\ \text{entonces} & F_{MÁX} & < & \frac{F_S}{2} \\ & F_{MÁX} & < & \frac{1}{2T_S} \\ & F_{MÁX} & < & \frac{1}{2\,100\mu S} = 5KHz \end{array} \quad (3)$$

Una vez obtenida la máxima frecuencia que se puede reconocer por parte del sensor, conjuntamente con los valores de resistividad (Tabla 1), se realizan ensayos para determinar el valor que debe tener el capacitor de la red *RC* del oscilador[2]. En la Tabla 2 se muestran dos configuraciones diferentes para distintos valores de capacitancia (10 nF y 100 nF).

Capacitor	Rango del LDR [Ω]	Frecuencia del osc.
10 nF	1 KΩ	100 KHz
	180 KΩ	555 Hz
100 nF	1 KΩ	10 KHz
	180 KΩ	55 Hz

Tabla 2: Rango de frecuencias de salida en función de la red *RC*.

En función a la Tabla 2 se puede decir que el valor de capacidad más apropiado es *100 nF*. Pues con este valor se tiene un rango de frecuencia aceptable para la frecuencia de muestreo de 100 μs.

[2]Se considera $k = 1$

Una *características importante* del circuito planteado en este trabajo es la posibilidad de *reutilizar* la línea de comunicación como canal de alimentación. El circuito a utilizar se presenta en la Fig. 5. En el esquema de conexión se destacan dos componentes que son responsables de mantener el nivel de tensión de alimentación, el capacitor electrolítico $C1$ y el diodo $D1$. El capacitor electrolítico almacenará carga recibida de la línea serial, direccionada por $D1$. En el momento que la línea baje su nivel de tensión, el capacitor proporcionará la corriente al circuito inversor, obviamente que lo hará por un tiempo reducido ya que comenzará a descargarse. El resistor $R2$ permite aumentar la impedancia de la salida del invesor (TTL).

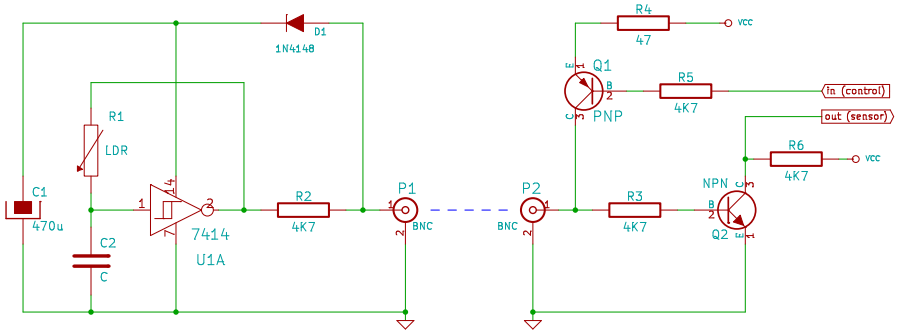


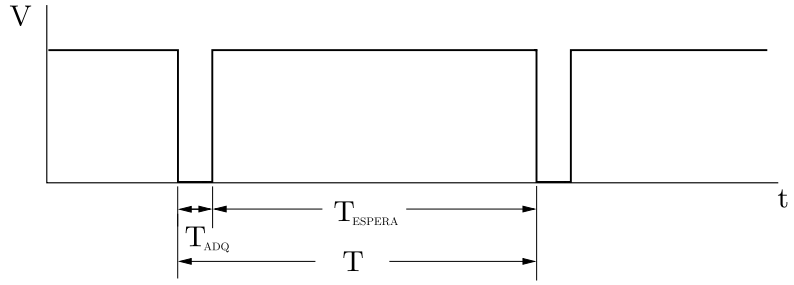
Figura 5: Implementación del circuito electrónico.

La línea de comunicación/alimentación conecta el circuito del sensor (del lado izquierdo de la Figura 5) con el circuito de control del sistema (a la derecha del la figura). El sistema de control es muy sencillo y cuenta de solo dos secuencias.

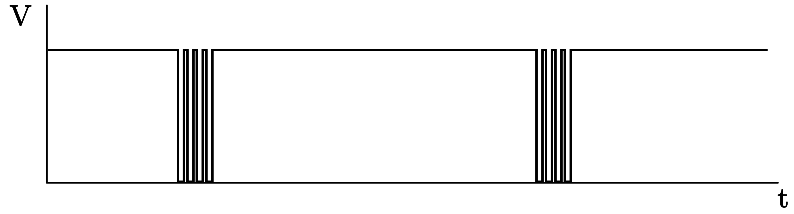
Mode de espera, este modo consiste en mantener la línea a un nivel de tensión necesario para alimentar el circuito del sensor (inversor). Para esta situación se debe enviar una señal de control a la base del transistor $Q1$. La señal *in*(*control*) proviene del dsPIC®, controlador por el RTOS, que veremos en las siguientes secciones. Si la señal presenta un 0 lógico el transistor PNP se polarizará y la línea quedará conectada a la tensión V_{CC}. La señal de salida *out*(*sensor*) tendrá el mismo nivel que la línea (V_{CC}) ya que el transistor $Q2$ está polarizado con la señal de la línea.

Modo adquisición, la adquisición de la señal oscilante que genera el circuito inversor es transmitida cuando la línea es liberada por el transistor $Q1$. En el momento que se presenta un 1 en la señal de control de $Q1$, éste queda abierto. Por una lado, el *Schmitt-trigger* deja de ser alimentado por la línea pero recibe carga del capacitor $C1$. Por otro lado la señal generada por le oscilador es transmitida a la línea que será amplificada por el transistor $Q2$. Si bien la señal del oscilador podría encontrarse debilita por pérdidas en la línea, el circuito de $Q2$ permite polarizar con niveles bajos de corriente el transistor NPN.

Esta secuencia debe mantener una relación entre el *ciclo de adquisición* y *ciclo de espera*. Por ejemplo, un ciclo de adquisición muy largo provocará una elevada reducción en la tensión de alimentación del inversor que, en este ciclo, es proporcionado por el capacitor $C1$. En caso contrario, un ciclo de adquisición muy corto podría no registrar el ciclo completo de oscilación del sensor. Una relación que se consideró apropiada es $^1/_{10}$, para un ciclo completo de control T el tiempo de adquisición se toma $^1/_T$ y el tiempo de espera $^9/_T$. La representación temporal de este proceso se puede ver en la Figura 6.

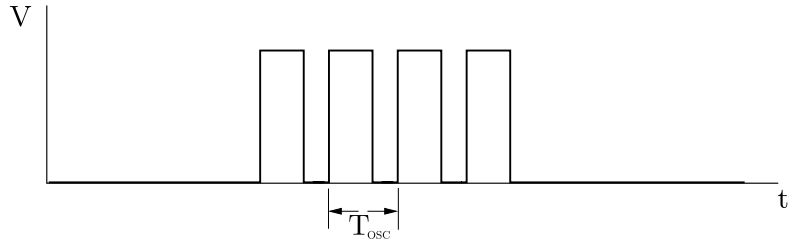


(a) Señal de control (Base del transistor $Q1$)



(b) Señal en la línea (Colector $Q1$ / Base $Q2$)

(c) Señal de salida (Colector del transistor $Q2$)



(d) Señal de salida amplificada de la figura (c)

Figura 6: Diagramas temporales de los ciclos de Espera y Adquisición.

3.2. Circuito del dsPIC®

Para realizar el proceso de control del sistema de censado se utiliza un microcontrolador del fabricante Microchip, el DSPIC33FJ128GP804-E/PT. Este dispositivo presenta enormes prestaciones en recursos de *hardware,* aquí se presentan algunos de ellos:

Clock Management

- 2 % internal oscillator
- Programmable PLL and oscillator clock sources
- Fail-Safe Clock Monitor (FSCM)
- Independent Watchdog Timer
- Low-power management modes

- Fast wake-up and start-up

Core Performace

- Up to 40 MIPS 16-bit dsPIC33F CPU
- Single-cycle MUL plus hardware divide

Communication Interfaces

- Parallel Master Port (PMP)
- Two UART modules (10 Mbps)
  - Supports LIN 2.0 protocols
  - RS-232, RS-485, and IrDA® support
- Two 4-wire SPI modules (15 Mbps)
- Enhanced CAN (ECAN) module (1 Mbaud) with 2.0B support
- I2C module (100K, 400K and 1Mbaud) with SMbus support
- Data Converter Interface (DCI) module with I2S codec support

System Peripherals

- 16-bit dual channel 100 ksps Audio DAC
- Cyclic Redundancy Check (CRC) module
- Up to five 16-bit and up to two 32-bit Timers/Counters
- Up to four Input Capture (IC) modules
- Up to four Output Compare (OC) modules
- Real-Time Clock and Calendar (RTCC) module

Advanced Analog Features

- 10/12-bit ADC with 1.1Msps/500 ksps rate:
  - Up to 13 ADC input channels and four S&H
  - Flexible/Independent trigger sources
- 150 ns Comparators:
  - Up to two Analog Comparator modules
  - 4-bit DAC with two ranges for Analog Comparators

Input/Output

- Software remappable pin functions
- 5V-tolerant pins
- Selectable open drain and internal pull-ups
- Up to 5 mA overvoltage clamp current/pin
- Multiple external interrupts

Direct Memory Access (DMA)

- 8-channel DMA with no CPU stalls or overhead
- UART, SPI, ADC, ECAN, IC, OC, INT0

Debugger Development Support

- In-circuit and in-application programming
- Two program breakpoints
- Trace and run-time watch

La placa de desarrollo de este microcontrolador es un diseño realizado por estudiantes de nuestra casa de estudios *Andrés Hoc* y *Gonzalo Vassia*[3], Figura 7. Para la programación de este microcontrolador se utilizó el entorno de desarrollo *MPLAB® IDE 8.8v* y el correspondiente compilador *MPLAB C30 C Compiler 3.32v*.

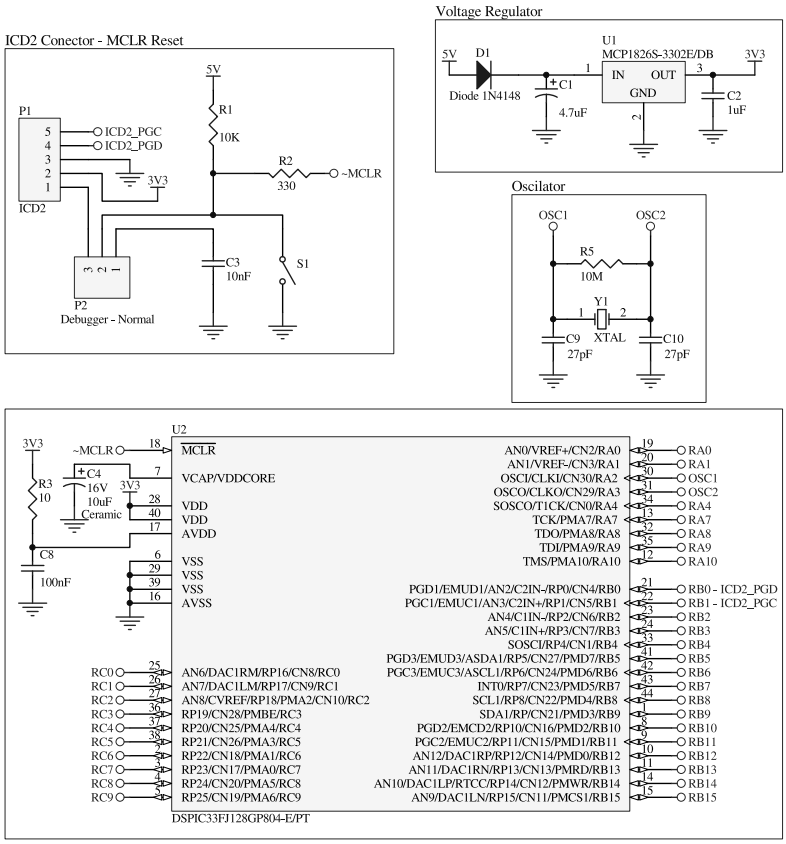


Figura 7: Esquemático del circuito del microcontrolador, solo los bloques fundamentales.

Como se mencionó anteriormente, quizá la complejidad del planteo del proyecto no requiere tanta capacidad de procesamiento, como así también justifique la utilización de un sistema operativo en tiempo real a fin de lograr adquirir la información censada. Pero debido a que el presente trabajo es una implementación de los RTOS, se prioriza la posibilidad de experimentar el manejo de sistemas operativos en desarrollos electrónicos. Los recursos de hardware a utilizar son:

[3]Los estudiantes realizaron la placa con la finalidad de disponer un diseño sencillo y flexible para múltiples propósitos.

Puerto de Entradas y Salidas, son necesarios para el control del circuito de censado que se describió en la Sección 3.1.2.

UART, se utiliza el interfaz serial UART sobre RS-232 para comunicarse con la PC quién procesará y mostrará la información censada al usuario.

El modo en que serán utilizados y su configuración es tema del código fuente del *software* del proyecto. La estructura de directorio para la compilación del μC/OS-II™en el dsPIC® se puede ver en el siguiente árbol,

```
firmware_dsPIC
├── stdint.h
├── p33FJ128GP804.inc
├── p33FJ128GP804.h
├── p33FJ128GP804.gld
├── os_cfg.h
├── MeDEf.mcw
├── MeDEf.mcp
├── isr.s
├── interfaz_cfg.h
├── interfaz_cfg.c
├── iniinterfaz.c
├── includes.h
├── globals.h
├── globals.c
├── GenericTypeDefs.h
├── dsPIC_delay.h
├── dsPIC_delay.c
├── dsPIC_cfg.h
├── dsPIC_cfg.c
├── dsPIC_a.s
├── drivers
│   ├── UART.h
│   ├── UART.c
│   └── DIO
│       └── SOURCE
│           ├── DIO.C
│           └── DIO.H
├── configobj.h
├── configinterfaz.h
├── configinterfaz.c
├── Compiler.h
├── app_hooks.c
├── app_cfg.h
├── app.c
└── RTOS
    ├── ucos_ii.h
    ├── os_tmr.c
    └── os_time.c
```

```
os_task.c
os_sem.c
os_q.c
os_mutex.c
os_mem.c
os_mbox.c
os_flag.c
os_dbg.c
os_cpu_util_a.s
os_cpu.h
os_cpu_c.c
os_cpu_a.s
os_core.c
lib_str.h
lib_str.c
lib_mem.h
lib_mem.c
lib_def.h
cpu.h
cpu_def.h
```

4. μC/OS-II™en el dsPIC®

μC/OS-II™es el acrónimo de *Micro-Controller Operating Systems Version 2*. Este es un sistema operativo multitareas en tiempo real *pre-emptive* basado en prioridades en su funcionamiento. El código se encuentra escrito principalmente en lenguaje de programación C. Se encuentra destinado para el uso en sistemas embebidos. Sus características son:

- Es un *kernel* de tiempo real muy pequeño.
- Con un espacio en memoria de aproximadamente unos 20KB para un completo funcionamiento del *kernel*.
- El código fuente se encuentra escrito en su mayoría en ANSI C.
- Gran portabilidad, muy escalable, *preemptive* en tiempo real, determinístico, *kernel* multitareas.
- Puede manejar hasta 64 tareas (con 56 tareas disponibles al usuario).
- Se encuentra disponible para más de 100 microcontroladores y microcontroladores.
- Es sencillo de usar y simple de implementar a la vez que es muy eficiente a la hora de comparar con otros RTOS en relación precio/*performance*.
- Soporta todos los tipos de procesadores desde 8-bit a 64-bit.

En el presente documento no se describirá el funcionamiento del μC/OS-II™. No solo porque nuestra intensión es la implementación del mismo, sino también se dispone de una gran cantidad de documentación sobre estos RTOS. De todas formas se señalará algunas características básicas de como trabaja el μC/OS-II™.

El *kernel* del μC/OS-II™dispone de los recursos de hardware, sobre todo los recursos de procesamiento y el indexado de memoria. Para cada implementación del RTOS se debe adaptar tanto el compilador como los *drivers* o *ports* necesarios para facilitar el acceso a los diferentes periféricos del microcontrolador. Una vez logrado establecer estos requerimientos de *software* simplemente se agrega al μC/OS-II™*tareas* para que el RTOS las procese. Una *tarea* es un código de programa el cual, en principio, se puede escribir con la idea de suponer que se dispone de todo el CPU para ella misma. El proceso de diseño para una aplicación en tiempo real implica la división del trabajo a realizar en tareas que son responsables de una parte del problema. A cada tarea se le asigna una prioridad, registros del CPU, y su propia área de stack para almacenar las variables locales que contiene cuando se está conmutando de tareas en el proceso *multitasking*. Cada tarea se encuentra en un bucle infinito (*loop*) el cual, en el tiempo de ejecución del código en el hardware, puede encontrarse en cualquier de cinco estados posibles:

DORMANT

READY

RUNNING

WAITING

INTERRUPTED

El estado *DORMANT* corresponde a una tarea que reside en memoria pero no ha sido habilitada a ingresar al *multitasking*. Una tarea está *READY* cuando puede ejecutarse pero su prioridad es menor que la tarea que se encuentra actualmente corriendo. Una tarea esta en *RUNNING* cuando tiene el control del CPU. Una tarea está en *WAITING* cuando requiere que ocurra un evento (por ejemplo; esperando que a una operación de un periférico se complete). Finalmente, una tarea está en el estado *INTERRUPTED* cuando una interrupción ocurrió y el CPU se encuentra atendiendo este llamado. La Figura 8 se muestran los estados anteriormente nombrados y además se puede ver las funciones provistas por el μC/OS-II™para realizar la conmutación entre un estado a otro.

4.1. Tareas del RTOS

Antes de comenzar a escribir el código de las *tareas* se define los procesos requeridos. A continuación se lista las tareas necesarias para el proyecto

- Generación de la señal de control para el sistema del sensor (Sección 3.1.2).
- Procesamiento de los datos recibidos por el sensor.
- Comunicación serial con la PC (envío de datos adquiridos).

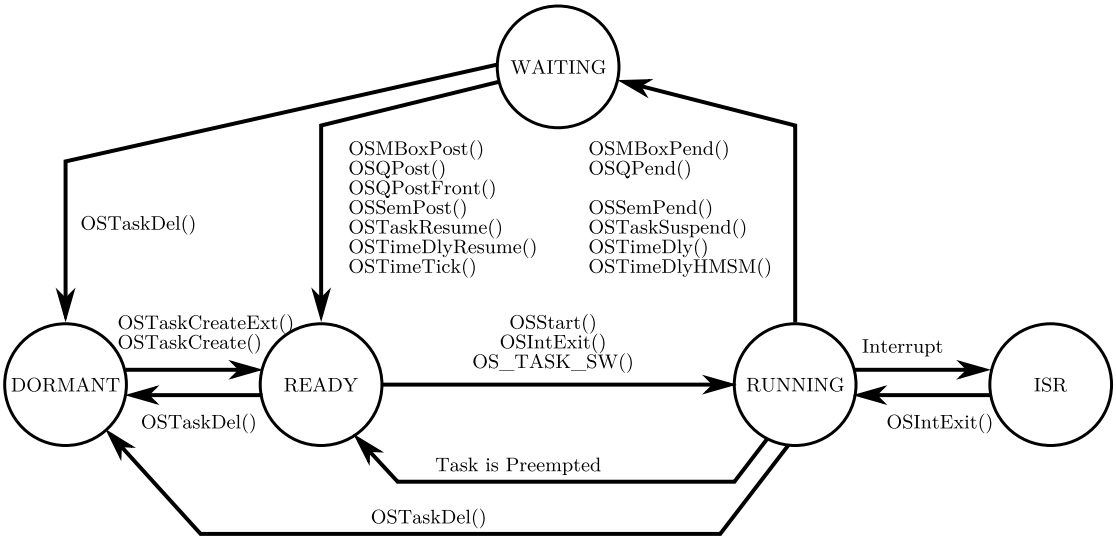


Figura 8: Estados de las tareas en el µC/OS-II™.

Obviamente que se requerirá de otras tareas pero son requerimientos del µC/OS-II™y son comunes en cualquier proyecto de este tipo. La función principal del proyecto *main()* contiene el llamado a todas las funciones de inicialización y configuración del microcontrolador, Código 1. Luego del llamado a la función *OSInit()* se debe crear todas las tareas del RTOS previo a que el sistema operativo entre en un estado de funcionamiento a través del llamado a *OSStart()*. Como convención se creará una tarea *TareaInicio()* que será donde se lancen la creación de todas las tareas. También se puede ver que se hace el llamado a las funciones *DIOInit()* que inicializa el módulo DIO que será descrito en la Sección 4.2.

Código 1: Función principal del proyecto, *main()*

```
CPU_INT16S  main (void)
{
  CPU_INT08U  err;

  BSP_IntDisAll(); /* Disable all interrupts until we are ready to accept
      them */

  RCON = 0;

  OSInit(); /* Initialize "uC/OS-II, The Real-Time Kernel" */
  DIOInit();

  OSTaskCreateExt(TareaInicio,
  (void *)0,
  (OS_STK *)&tareaInicioStk[0],
  TAREA_INICIO_PRIO,
  TAREA_INICIO_PRIO,
  (OS_STK *)&tareaInicioStk[TAREA_INICIO_STK_SIZE-1],
  TAREA_INICIO_STK_SIZE,
  (void *)0,
  OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

  #if OS_TASK_NAME_SIZE > 11
```

```
    OSTaskNameSet(TAREA_INICIO_PRIO, (CPU_INT08U *)"Start Task", &err);
    #endif

    OSStart(); /* Start multitasking (i.e. give control to uC/OS-II) */

    return (-1); /* Return an error - This line of code is unreachable */
}
```

La *TareaInicio()* realiza el llamado a la función *AppTaskCreate()* que será la encargada de crear todas las tareas del proyecto. Además en esta primera tarea se crea e inicializa el *MailBox* que será utilizado para la comunicación interna entre tareas ofrecido por el *kernel,* pero esto se verá más adelante en la Sección 4.4.

Código 2: Función de la primer tarea creada*, TareaInicio()*

```
static  void  TareaInicio (void *p_arg)
{
  CPU_INT08U  i;
  CPU_INT08U  j;

  (void)p_arg;

  BSP_Init(); /* Initialize BSP functions */

  #if OS_TASK_STAT_EN > 0
  OSStatInit(); /* Determine CPU capacity */
  #endif

  AppTaskCreate(); /* Create additional user tasks */

  /*Se crea un MailBox para indicar cuando hay un nuevo periodo de
      velocidad */
  mBoxPromData= OSMboxCreate((void *) NULL);
  mBoxPromData->OSEventPtr = 0;

  while (DEF_TRUE)
  { /* Task body, always written as an infinite loop. */
    OSTimeDly(100);
  }
}
```

La creación de las tareas se pueden ver en el código 3. Esta función solo realiza el llamado a la función *OSCreateExt()* que permite crear tareas y que sea procesada por el µC/OS-II™. Los argumentos que se debe proporcionar para cada llamado son

```
INT8U  OSTaskCreateExt (void   (*task)(void *p_arg),
                        void    *p_arg,
                        OS_STK  *ptos,
                        INT8U    prio,
                        INT16U   id,
                        OS_STK  *pbos,
                        INT32U   stk_size,
                        void    *pext,
                        INT16U   opt)
```

task: es un puntero a la función de la tarea.

p_arg: es un puntero opcional a una área de datos el cual pueda ser usado para pasar parámetros a la tarea cuando es ejecutado por primera vez.

ptos: es un puntero al extremo superior del *stack*. (Debe revisarse la constate de configuración *OS_STK_GROWTH*.

prio: es la prioridad de la tarea. Un único número debe ser asignado para cada tarea.

id: es el identificador de la tarea ($0 \cdots 65535$).

pbos: es un puntero al extremo inferior del *stack*. (Debe revisarse la constate de configuración *OS_STK_GROWTH*.

stk_size: es el tamaño del *stack* en número de elementos.

pext: es el puntero a una posición de memoria suministrada por el usuario para ser usada como una extensión *TCB (Task Control Block)*.

opt: contiene información adicional (u opcional) sobre el comportamiento de la tarea.

- OS_TASK_OPT_STK_CHK, chequeo del *stack* está permitido por la tarea.
- OS_TASK_OPT_STK_CLR, limpiar el *stack* cuando la tarea es creada.
- OS_TASK_OPT_SAVE_FP, si el CPU tiene registros de punto-flotante, guardarlos durante un intercambio de estado.

En función de los argumentos recibidos, la función *OSCreateExt()* devuelve,

OS_ERR_NONE: si la función ha sido creada correctamente.

OS_PRIO_EXIT: si la prioridad de la tarea ya existe.

OS_ERR_PRIO_INVALID: si la prioridad especificada es mayor que la prioridad más alta permitida.

OS_ERR_TASK_CREATE_ISR: si intenta crear una tarea desde un *ISR*

Código 3: Función de la tarea *AppTaskCreate()*

```
static  void  AppTaskCreate (void)
{
  CPU_INT08U  err;

  OSTaskCreateExt(TareaControlSensor,
  (void *)0,
  (OS_STK *)&tareaControlSensor[0],
  TAREA_CONTROLSENSOR_PRIO,
  TAREA_CONTROLSENSOR_PRIO,
  (OS_STK *)&tareaControlSensorStk[TAREA_CONTROLSENSOR_STK_SIZE-1],
  TAREA_CONTROLSENSOR_STK_SIZE,
  (void *)0,
  OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR | OS_TASK_OPT_SAVE_FP);

  OSTaskCreateExt(TareaPromDatos,
  (void *)0,
  (OS_STK *)&tareaPromDatos[0],
  TAREA_PROMDATOS_PRIO,
  TAREA_PROMDATOS_PRIO,
  (OS_STK *)&tareaPromDatosStk[TAREA_PROMDATOS_STK_SIZE-1],
  TAREA_PROMDATOS_STK_SIZE,
```

```
  (void *)0,
  OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR | OS_TASK_OPT_SAVE_FP);

  OSTaskCreateExt(TareaComSerial,
  (void *)0,
  (OS_STK *)&tareaComSerial[0],
  TAREA_COMSERIAL_PRIO,
  TAREA_COMSERIAL_PRIO,
  (OS_STK *)&tareaComSerialStk[TAREA_COMSERIAL_STK_SIZE-1],
  TAREA_COMSERIAL_STK_SIZE,
  (void *)0,
  OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR | OS_TASK_OPT_SAVE_FP);

}
```

4.2. Módulo DIO

El módulo *DIO* es un bloque de código que fue desarrollado por el mismo grupo que escribió el µC/OS-II™. Este se encuentra adaptado a la estructura de funcionamiento del RTOS. El bloque permite controlar canales de entradas y salidas digitales en forma independiente. En la Figura 9 se puede ve run diagrama del módulo completo. El módulo DIO consiste en una simple tarea (*DIOTask()*) que se ejecuta a intervalos regulares *DIO_TASK_DLY_TICKS*. *DIOTask()* puede manejar una gran cantidad de canales discretos de Entradas y Salidas (hasta 250 cada una). El módulo DIO se inicializa con la llamada a la función *DIOInit()*. Cada *DIO_TASK_DLY_TICKS*, *DIOTask()* llama a las funciones *DIRd()*, *DIUpdate()*, *DOWr()* y *DOUpdate()*. *DITbl[ ]* es una tabla que contiene configuración e información en tiempo real de cada canal de entrada. Las entradas son leídas y mapeadas a *DITbl[i].DIIn* por el controlador del *hardware* implementado en la función *DIRd()*. *DIRd()* es la función que interacciona con el dispositivo físico (puerto del dsPIC®). Además en la Figura 9 se puede ver que se cuentan con varias funciones que permiten abstraerse del *hardware* y disponer de ellos, siendo la tarea del módulo DIO la que lidie con los problemas que puedan existir en la lectura y escritura de los puertos físicos de nuestra placa.

El diagrama en bloque de la Figura 10 muestra el funcionamiento del módulo DIO para el tratado de las señales de entrada. *.DIIn*, *.DIModeSel*, *.DIBypassEn* y *.DIVal* son miembros de la estructura de datos *DIO_DI* que se encuentra definida en el código fuente del módulo dio (archivo `DIO.H`). *DIUpdate()* es el responsable de actualizar todas los canales de entradas discretas del módulo.

Al igual que los canales de entrada, Figura 9, *DOTbl[ ]* es una tabla que contiene configuración e información en tiempo real por cada canal de salidas. Las salidas discretas son mapeadas desde *DOTbl[i].DOOut* al puerto físico a través de la función *DOWr()*. La función *DOWr()* es el controlador o *driver* que interacciona con la capa física de nuestro sistema (al igual que *DIRr()*). En la Figura 11 se muestra un diagrama de como funciona el módulo DIO para las señales de salida de nuestro bloque. *.DOCtrl*, *.DOBypassEn*, *.DOBypass*, *.DOBlinkEnSel*, *.DOModeSel*, *.DOInv* y *.DOOut* son miembros de la estructura de datos *DIO_DO* definido en el archivo `DIO.H`. *DOUpdate()* es el responsable de actualizar todos los canales de salidas discretas.

Como toda tarea que debe atender el RTOS, se debe asignar una prioridad y una latencia en ejecución. Se asigna una de las más altas prioridades al módulo pues será quién controle el sensor. Las además tareas planteadas en la Sección 4.1 no son tan deman-

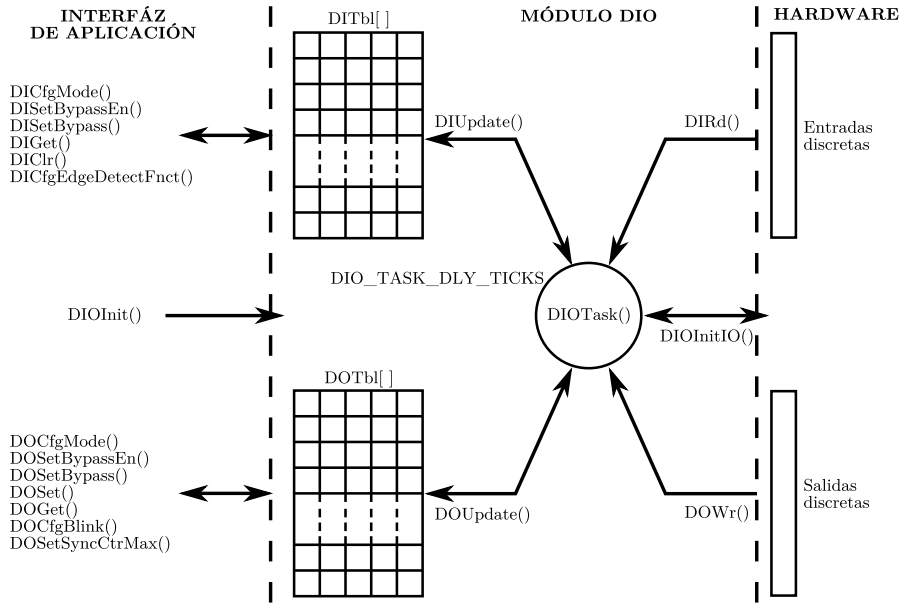


Figura 9: Diagrama de flujo del Módulo DIO.

dantes. En el caso de especificar a que frecuencia debe ejecutarse el módulo DIO, se considera el valor más alto, lo que implica que se refrescará las Entradas/Salidas lo más rápido posible. El archivo `DIO.H` contiene varios #define que parametrizan el comportamiento del módulo.

Código 4: Definiciones de parámetros de prioridad y latencia del Módulo DIO.

```
#define  DIO_TASK_PRIO              1
#define  DIO_TASK_DLY_TICKS         1
```

Como se dijo anteriormente, las funciones *DOWr()* y *DIRd()* serán quienes interactúen con el *hardware*. Originalmente el código del módulo DIO estaba escrito para acceder al puerto paralelo de una computadora de escritorio. La configuración de los puertos del microcontrolador se realiza en la función *DIOInitIO()*,

Código 5: Función *DIOInitIO()*

```
void  DIOInitIO (void)
{
  /* Configuración de los puertos I/O */
  /* SALIDAS */
  TRISBbits.TRISB9 = 0;
  TRISBbits.TRISB8 = 0;
  TRISBbits.TRISB7 = 0;
  TRISBbits.TRISB6 = 0;

  /* ENTRADAS */
  TRISCbits.TRISC0 = 1;
  TRISCbits.TRISC1 = 1;
  TRISCbits.TRISC2 = 1;
  TRISCbits.TRISC3 = 1;

  AD1PCFGLbits.PCFG6 = 1;     //RC0
  AD1PCFGLbits.PCFG7 = 1;     //RC1
```

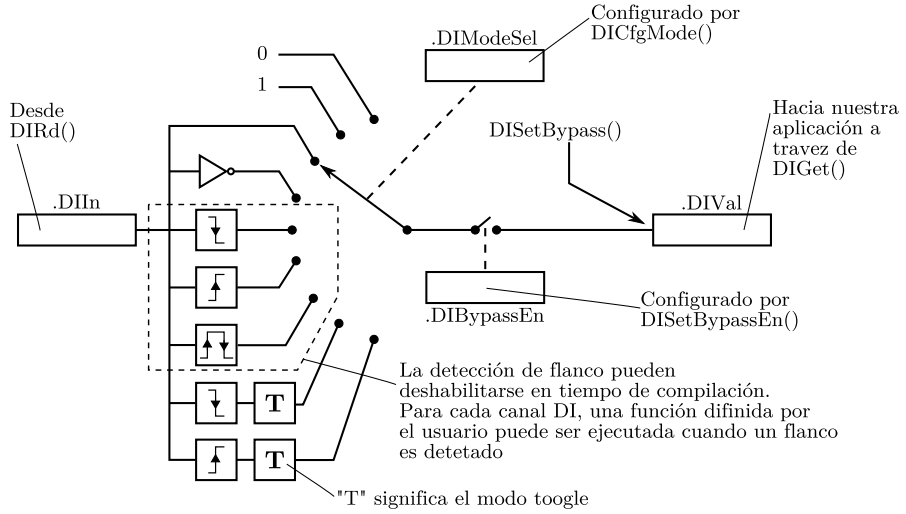


Figura 10: Canales de entradas discretas del Módulo DIO.

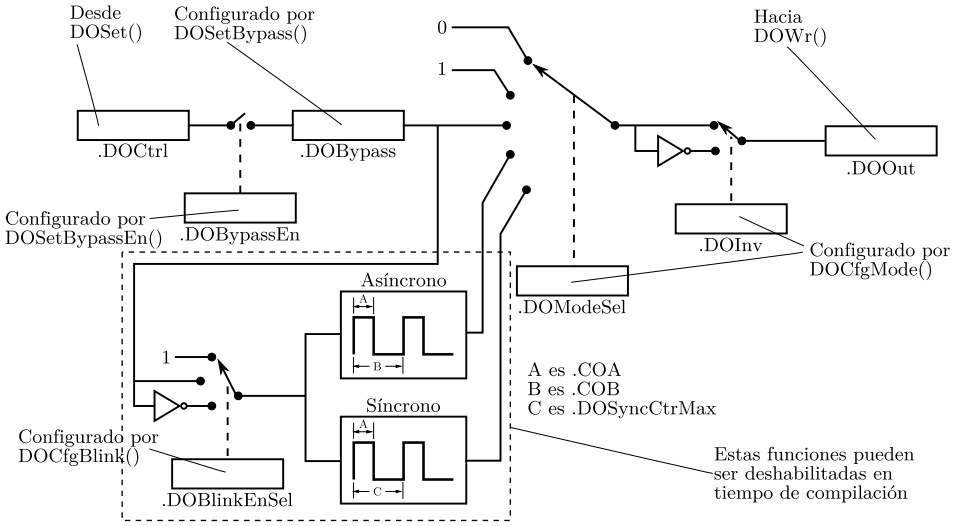


Figura 11: Canales de salidas discretas del Módulo DIO.

```
  AD1PCFGLbits.PCFG8 = 1;      //RC2
}
```

Una vez configurados los puertos, la función *DIOInitIO()* es llamada desde la función principal. La función *DIRd()* es modificada para asignar los puertos que serán leídos y cargados en la tabla *DITbl[ ]*.

Código 6: Función *DIRd()*

```
void  DIRd (void)
{
  DIO_DI  *pdi;
  INT8U    i;
  INT8U    in;
  INT8U    msk;

  pdi = &DITbl[0]; /* Point at beginning of discrete inputs */
```

```
  msk = 0x01; /* Set mask to extract bit 0 */
  in = 0;
  in  |= PORTCbits.RC0; /* Read the physical port (8 bits) */
  in  |= (PORTCbits.RC1<<1);
  in  |= (PORTCbits.RC2<<2);
  in  |= (PORTCbits.RC3<<3);
  for (i = 0; i < 8; i++) /* Map all 8 bits to first 8 DI channels */
  {
    pdi->DIIn   = (BOOLEAN)(in & msk) ? 1 : 0;
    msk       <<= 1;
    pdi++;
  }
}
```

Recordemos que el módulo DIO puede controlar hasta 256 canales de entradas y salidas. En nuestro caso no se requería de más de dos canales, uno de entrada y otro de salida. De todas formas se configuran cuatro canales I/O. Para la función que escribe el puerto de salida del microcontrolador, *DOWr()*, se debe asignar que puertos físicos tomarán los valores de la tabla *DOTbl[ ]*. Al igual que en el caso de la lectura, aquí se utilizaron solo cuatro canales de los que dispone el módulo DIO.

Código 7: Función *DIWr()*

```
void  DOWr (void)
{
  DIO_DO  *pdo;
  INT8U    i;
  INT8U    out;
  INT8U    msk;


  pdo = &DOTbl[0]; /* Point at first discrete output channel */
  msk = 0x01; /* First DO will be mapped to bit 0 */
  out = 0x00; /* Local 8 bit port image */
  for (i = 0; i < 8; i++) /* Map first 8 DO to 8 bit port image */
  {
    if (pdo->DOOut == TRUE) {
      out |= msk;
    }
    msk <<= 1;
    pdo++;
  }
  PORTBbits.RB9 = out & 0x01;
  PORTBbits.RB8 = (out>>1) & 0x01;
  PORTBbits.RB7 = (out>>2) & 0x01;
  PORTBbits.RB6 = (out>>3) & 0x01;
}
```

4.3. Tarea para el control del sensor

En la Sección 3.1.2 se describe el funcionamiento del sistema de censado que se implementa en el correspondiente trabajo. Las señales que controlan el proceso de adquisición de datos a través de la línea de comunicación son generados por una tarea del µC/OS-II™. La tarea *TareaControlSensor()* utiliza el módulo DIO para generar la señal de *control*, véase el circuito de la Figura 5. El tiempo de *adquisición* y el tiempo de *espera* es controlado por las funciones de tiempo del RTOS,

Código 8: Función de la tarea *TareaControlSensor()*

```
static void TareaControlSensor (void)
{

  // Configuración de canales del módulo DIO

  DOCfgMode(0,DO_MODE_DIRECT,0);
  DICfgMode(0,DI_MODE_EDGE_HIGH_GOING);
  ValChZero = 0;
  CntData = 0;
  DIClr(0);

  while(1)
  {

    DOSet(0,TRUE);
    OSTimeDlyHMSM(0,0,0,100);
    DOSet(0,FALSE);
    OSTimeDlyHMSM(0,0,1,0);

    ValChZero += DIGet(0);
    CntData++;
    DIClr(0);

  }
}
```

La salida del microcontrolador que maneja la señal de control del sensor será el primer canal del módulo DIO (canal 0). En esta función se puede apreciar que el tiempo de *adquisición* es 100 microsegundos y el tiempo de *espera* es de 1 segundo, relación especificada en la Sección 3.1.2.

La entrada será el canal 0 y es conectada a la señal de salida del sensor (colector del transistor *Q*2, Fig. 5). Este canal se configura en el modo de detección de flanco ascendente, *DICfgMode(0,DI_MODE_EDGE_HIGH_GOING)*. Se acumula en la variable *ValChZero* la cantidad de flancos recibidos desde la línea de comunicación y a la vez se acumula el número de muestras recibidas, pues se enviará un promedio del periodo calculado. Se aclara que la mayoría de las variables que se utilizan en este proyecto son variables globales. La declaración de estas variables se pueden ver al final del documento donde se proporciona todo el código fuente del proyecto.

4.4. Tarea procesamiento de datos del sensor

La tarea *TareaPromDatos()* realiza un promedio de la cantidad de pulsos recibidos. La tarea procesa la cantidad de flancos recibidos, que son acumulados en la variable *ValChZero*, y los promedia con la cantidad de veces que se realiza la captura, *CntData*. Este proceso se realiza a una frecuencia menor que la tarea *TareaControlSensor()*, cada 4 segundos.

En esta función se implementan los *MailBoxes* para comunicarse con la tarea que envíe los datos por el puerto serie. Los *MailBoxes* son recursos que ofrece el µC/OS-II™para intercambiar mensajes entre tareas a través del *kernel*. Una tarea que desee un mensaje desde un *MailBox* vacío es suspendida y puesta en una lista de espera hasta que el mensaje sea recibido. El *kernel* permite que la tarea espere por el mensaje hasta que haya transcurrido un tiempo determinado (*timeout*). Si el mensaje no es recibido,

transcurrido el tiempo de espera, la tarea que requiere del mensaje pasará del estado *Ready* a *Run* retornará un código de error en el retorno del pedido. El mecanismo con el que funciona este servicio de mensajería del sistema operativo se puede ver en al Figura 12. La función *OSMboxPost()* realiza la acción de **poner** el mensaje que en este caso es el valor promedio de los datos adquiridos. El primer argumento de la función *OSMboxPost()*, *mBoxPromData*, es una variable del tipo *OS_EVENT* que es una estructura de datos definida por el μC/OS-II™para el manejo de información interna del sistema operativo.

Código 9: Función de la tarea *TareaPromDatos()*

```
static  void  TareaPromDatos (void)
{
  TRISBbits.TRISB5 = 0;

  while(1)
  {
    if(CntData!=0)
    {
      Prom = (float) ValChZero / (float) CntData;
      ValChZero = 0;
      CntData = 0;
    }
    else
    {
      Prom = 0;
    }

    OSMboxPost(mBoxPromData, &Prom); /* Envío msj. a tarea que transmite
        los datos (promedio de los pulsos recibidos) */

    OSTimeDlyHMSM(0,0,4,0);
  }
}
```

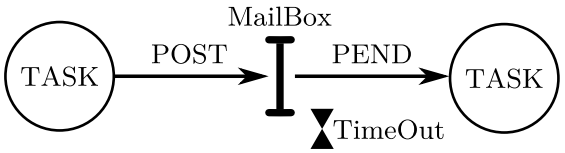


Figura 12: Funcionamiento de los *MailBoxes*.

4.5. Tarea de comunicación serial

La últimas de las tareas implementadas en el microcontrolador es la comunicación serial a través de la UART del dsPIC®. La configuración de la UART se realiza en la función *InitUART()*. La tarea *TareaComSerial()* se encuentra condicionada a la recepción del mensaje desde la tarea *TareaPromDatos()* a través del *MailBox mBoxPromData*. El tiempo de espera por el mensaje se define en *TIMEOUT_PEND_MBOX* y se encuentra en unidades ticks. Una vez que se reciba el mensaje, *OSMboxPend()* proporciona un puntero a la posición de memoria donde se encuentra el mensaje. Y sobre esta información se procesa el dato a ser enviado por el canal serial. La frecuencia con que trabaja esta tarea

es de un segundo, pero como se dijo al comienzo, se encuentra condicionada por la espera del *MailBox*.

Código 10: Función de la tarea *TareaComSerial()*

```
static  void  TareaComSerial (void)
{
  uint8_t errorMboxPend;

  asm volatile (``mov #OSCCONL, w1  \n''
  ``mov #0x46, w2     \n''
  ``mov #0x57, w3     \n''
  ``mov.b w2, [w1]    \n''
  ``mov.b w3, [w1]    \n''
  ``bclr OSCCON, #6'');

  RPINR18bits.U1RXR = 22;
  RPOR11bits.RP23R = 0b00011;

  asm volatile (``mov #OSCCONL, w1  \n''
  ``mov #0x46, w2     \n''
  ``mov #0x57, w3     \n''
  ``mov.b w2, [w1]    \n''
  ``mov.b w3, [w1]    \n''
  ``bset OSCCON, #6'');

  InitUART1();

  while(1)
  {
    PtrPromFromMBox = OSMboxPend(mBoxPromData, TIMEOUT_PEND_MBOX, &
        errorMboxPend); /* Esperamos a que llegue un nuevo promedio */
    PromFromMbox = *PtrPromFromMBox;
    whole=(int) PromFromMbox ; /* obtains whole part */
    decimal=(int) ((PromFromMbox - (float) whole)*100.0); /* obtains
        decimal part (1 digit) */
    sprintf(array,":%04d.%02d\n",whole,decimal); /* converts to string */
    for(iiTaskUart = 0; iiTaskUart<9 ; iiTaskUart++)
    {
      while(U1STAbits.UTXBF != 0);
      U1TXREG = array[iiTaskUart];
    }

    OSTimeDlyHMSM(0,0,1,0);

  }
}
```

5. µC/OS-II™en la PC

Siguiendo la línea del uso de los sistemas operativos en tiempo real, se implementa el sistema µC/OS-II™en una arquitectura de PC estándar. Se utilizará un proyecto base publicado por Micriµm Inc., creador del µC/OS-II™, compilado con el *software TurboC*. Las características del proyecto serán,

- Procesar los datos recibidos a través de una comunicación serial con el RTOS que

corre en el dsPIC®.

- Manejo de funciones gráficas para la presentación de los datos en la pantalla de la PC.

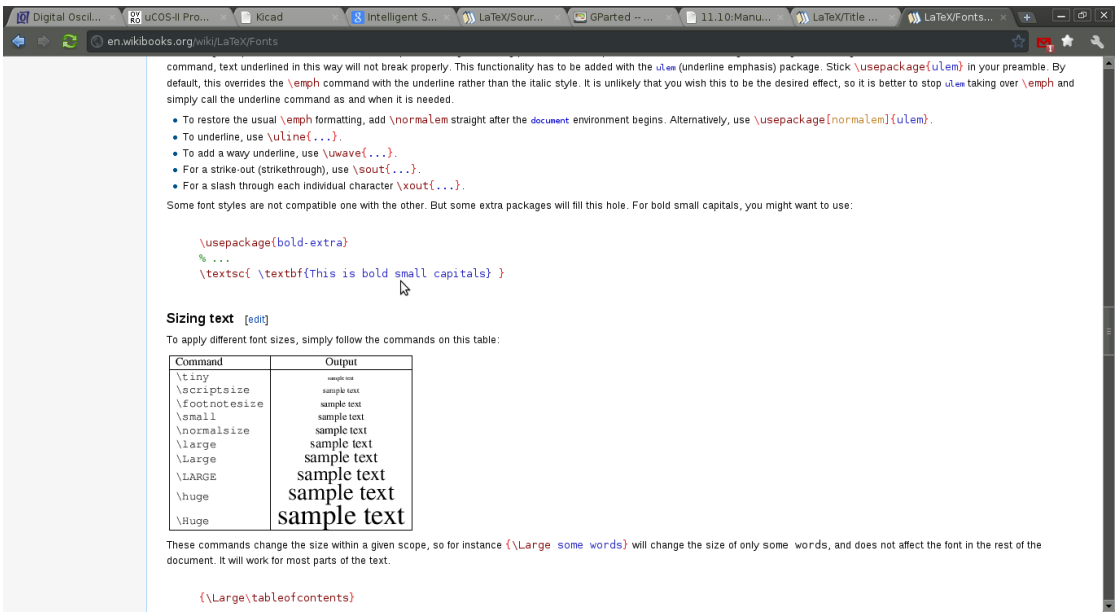


Figura 13: Entorno gráfico del *software* TurboC.

La función principal *main()* se puede ver en el Código 11. Como en el caso del µC/OS-II™en el dsPIC®, se debe realiza la configuración y la creación de las tareas luego del llamado a la función *OSInit()* y antes de que el RTOS comience a correr *OSStart()*.

Código 11: Función principal del µC/OS-II™sobre la PC (*main()*)

```
void main (void)
{
  PC_DispClrScr(DISP_FGND_WHITE); /* Clear the screen */

  OSInit(); /* Initialize uC/OS-II */

  PC_DOSSaveReturn(); /* Save environment to return to DOS */
  PC_VectSet(uCOS, OSCtxSw); /* Install uC/OS-II's context switch vector */
  PC_ElapsedInit(); /* Initialized elapsed time measurement  */

  CommInit();

  /* Crear los Semaphores (Buzones) */
  TransmisionesListasSem = OSSemCreate(1);

  OSTaskCreate(TaskStart, (void *)0, &TaskStartStk[TASK_STK_SIZE-1], 0);

  OSStart(); /* Start multitasking   */
}
```

Junto a los archivos del µC/OS-II™se dispone de varios códigos fuentes para hacer uso de varios periféricos. Estos se encuentran en el directorio *BLOCKS*. Se puede listar estos archivos como,

```
SOFTWARE
├── BLOCKS
│   ├── AIO
│   ├── CLK
│   ├── COMM
│   │   └── SOURCE
│   │       ├── COMMRTOS.H
│   │       ├── COMMRTOS.C
│   │       ├── COMM_PC.H
│   │       ├── COMM_PC.C
│   │       ├── COMM_PCA.ASM
│   │       ├── COMMBGND.H
│   │       └── COMMBGND.C
│   ├── DIO
│   ├── KEY_MN
│   ├── LCD
│   ├── LED
│   ├── PC
│   │   └── SOURCE
│   │       ├── PC.C
│   │       └── PC.H
│   ├── SIMPLE
│   └── TMR
├── HPLISTC
├── UCOS-II
└── doc
```

Los bloques que se utilizarán se encuentran resaltados en texto negrita. Pero también se puede ver el módulo DIO utilizado en el dsPIC® también está listado en este directorio.

Para el manejo de la pantalla se utilizan funciones del módulo *PC*. Este módulo provee servicios para mostrar caracteres ASCII en una pantalla VGA básica de una computadora. En el modo normal, la pantalla de una PC puede manejar hasta 2000 caracteres organizados en un arreglo de 25 filas por 80 columnas. La memoria de vídeo en la PC y comienza en una dirección absoluta de memoria 0x000B8000 (o usando notación segmentada, B800:0000). Cada caracter a mostrar requiere dos *bytes* para ser visualizado en la pantalla. El primer *byte* (menor posición en memoria) es el caracter que se quiere mostrar, mientras que el segundo *byte* es un atributo que determina la combinación de color *foreground*/*background* del caracter. El color *foreground* es especificado en los menores 4 *bits* del atributo, mientras que el color *background* en los superiores, del *bits* 4 al 6. Finalmente el más significativo *bit* determina si el caracter titilará (con 1) o no (con 0).

El el Código 12 muestra la primer tarea donde se configura y ejecuta varias llamadas a las funciones de pantalla para nuestro proyecto.

Código 12: Primera tarea del µC/OS-II™sobre la PC (*TaskStart()*)

```
void  TaskStart (void *data)
{
  char   s[80];
```

```
    WORD key;

    data = data; /* Prevent compiler warning */

    PC_DispStr(25, 27, "<-PRESS␣'ESC'␣TO␣QUIT->", DISP_FGND_WHITE +
        DISP_BLINK);
    PC_DispStr(16,  1, "Trabajo␣Final␣-␣uC/OS-II,␣The␣Real-Time␣Kernel",
        DISP_FGND_WHITE + DISP_BGND_RED + DISP_BLINK);

    OS_ENTER_CRITICAL(); /* Install uC/OS-II's clock tick ISR */
    PC_VectSet(0x08, OSTickISR);
    PC_SetTickRate(OS_TICKS_PER_SEC); /* Reprogram tick rate  */
    OS_EXIT_CRITICAL();
    OSStatInit();

    OSTaskCreate(Task_Times, (void *)0, &TaskTimesStk[TASK_STK_SIZE-1], 7);
    OSTaskCreate(Task_Rx, (void *)0, &TaskRxStk[TASK_STK_SIZE-1], 8);


    for (;;)
    {
      PC_DispStr(16,  1, "Trabajo␣Final␣-␣uC/OS-II,␣The␣Real-Time␣Kernel",
          DISP_FGND_WHITE + DISP_BGND_RED + DISP_BLINK);
      PC_DispStr(25, 27, "<-PRESS␣'ESC'␣TO␣QUIT->", DISP_FGND_WHITE +
          DISP_BLINK);
      if (PC_GetKey(&key)== TRUE)       /* See if key has been pressed */
      {
        if (key == 0x1B) /* Yes, see if it's the ESCAPE key */
        {
          PC_DOSReturn(); /* Yes, return to DOS */
        }
      }
      OSTimeDlyHMSM(0, 0, 0, 200);
    }
  }
```

En la primera tarea se crean las demás tareas del µC/OS-II™. En este caso son, la tarea que realiza la recepción de datos a través del puerto serie y última es una tarea que realiza la limpieza de la pantalla. Una captura de la pantalla se observa en la Figura 14.

5.1. Tarea de comunicación serial

Para manejar el interfaz serial de la PC se hace uso de un módulo proporcionado por el autor del µC/OS-II™, el *COMM_PC*. Este módulo hace mucho más fácil de usar el puerto serie. El código y sus funcionalidades del *driver* son fácilmente portable a diferentes arquitecturas y entornos de desarrollo. La estructura del módulo cuenta con dos bloques como se muestra en la Figura 15. El bloque *Low Level PC Driver* es responsable de interactuar con el *hardware*, es decir, la UART de la PC. Se provee de funciones a nuestras aplicaciones que nos permitan configurar los dos puertos (*COM1* ó *COM2*), habilitar/deshabilitar interrupciones de comunicación, y adquirir/cargar el vector de interrupción del puerto *COM*. Nuestras aplicaciones también interaccionan a dos posibles tipos de *buffers* seriales, los módulos de I/O: *COMMBGND* ó *COMMRTOS*. Se puede usar el código del *COMMBGND* en aplicaciones *foreground*/*background* y el *COMMRTOS* si se está utilizando un *kernel*en tiempo real como el µC/OS-II™.

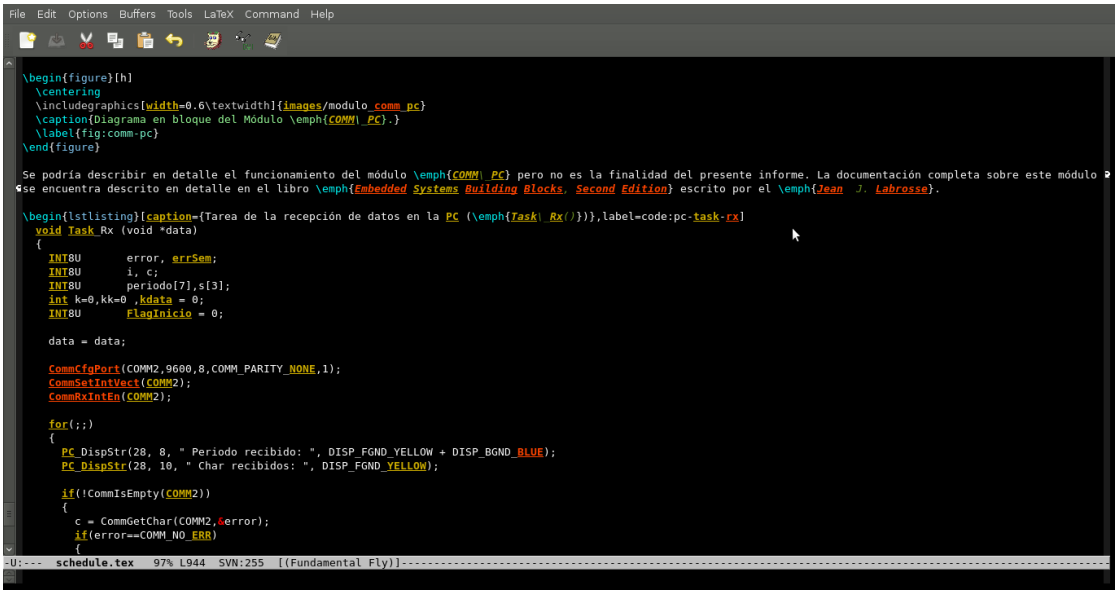


Figura 14: Captura de la pantalla del proyecto en funcionamiento.

Se podría describir en detalle el funcionamiento del módulo *COMM_PC* pero no es la finalidad del presente informe. La documentación completa sobre este módulo se encuentra descrito en detalle en el libro *Embedded Systems Building Blocks, Second Edition* escrito por el *Jean J. Labrosse*.

La comunicación es unidireccional, por las especificaciones del proyecto, pero para realizar una comunicación bidireccional no ha de ser inconveniente. En el Código 13 antes de lanzar el bucle infinito de la tarea, se configura el puerto serie de la PC, *CommCfgPort(COMM2,9600,8,COMM_PARITY_NONE,1)*. Y se lanza las interrupciones del puerto COMM2. Ya en tiempo de ejecución, se espera a que el *buffer* del puerto serie se encuentre con datos a ser procesados y son almacenados temporalmente para ser visualizados con los servicios de pantalla de la función *PC_DispStr()*. Además en esta tarea se utiliza *Semáforos*, recursos de comunicación interna entre tareas que ofrece el µC/OS-II™. Aquí el semáforo *TransmisionesListasSem* da aviso a la tarea de limpieza de pantalla que ya ha logrado visualizar el tren de datos recibidos. Esta trama de datos está identificada por un caracter de comienzo ":" (en decimal 58), y uno de final de trama "
n" (en decimal 10).

Código 13: Tarea de la recepción de datos en la PC (*Task_Rx()*)

```
void Task_Rx (void *data)
{
  INT8U       error, errSem;
  INT8U       i, c;
  INT8U       periodo[7],s[3];
  int k=0,kk=0 ,kdata = 0;
  INT8U       FlagInicio = 0;

  data = data;

  CommCfgPort(COMM2,9600,8,COMM_PARITY_NONE,1);
```

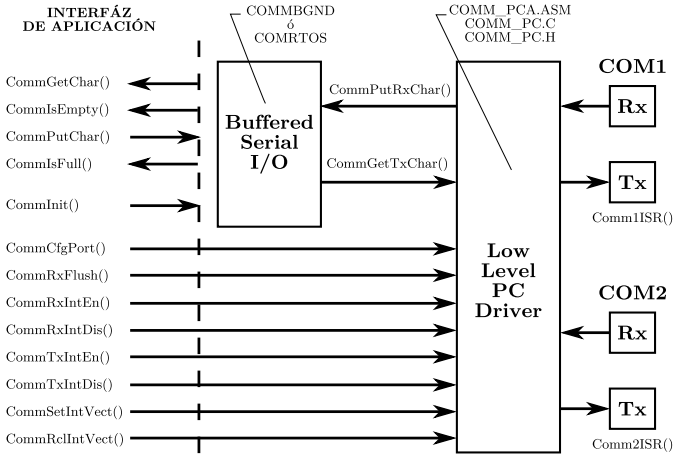


Figura 15: Diagrama en bloque del Módulo *COMM_PC*.

```
CommSetIntVect(COMM2);
CommRxIntEn(COMM2);

for(;;)
{
  PC_DispStr(28, 8, "␣Periodo␣recibido:␣", DISP_FGND_YELLOW +
      DISP_BGND_BLUE);
  PC_DispStr(28, 10, "␣Char␣recibidos:␣", DISP_FGND_YELLOW);

  if(!CommIsEmpty(COMM2))
  {
    c = CommGetChar(COMM2,&error);
    if(error==COMM_NO_ERR)
    {
      switch(c)
      {
        case 58:
        FlagInicio = 1;
        break;
        case 10:
        kdata = 0;
        FlagInicio = 0;
        errSem = OSSemPost (TransmisionesListasSem);
        break;
        default:
        if(FlagInicio == 1)
        {
          k++;
          kdata++;
          sprintf(s, "%d", k);
          PC_DispStr(60, 10, s, DISP_FGND_YELLOW);
          sprintf(s, "%c", (char)c);
          PC_DispStr(60+kdata, 8, s, DISP_FGND_YELLOW);
        }
      }
    }
  }

  OSTimeDlyHMSM(0, 0, 0, 500);
```

```
    }
}
```

5.2. Tarea limpieza pantalla

Esta tarea es muy sencilla de interpretar. Esta tarea esperar a que se haya recibido una trama completa en la tarea *Task_Rx* y luego realiza la limpieza de la pantalla con el uso de la función *PC_ElapsedStart()*. Al especificar un tiempo de espera por el *Semáforo* igual a cero, se esperará infinitamente por esta habilitación.

Código 14: Tarea de limpieza de pantalla (*Task_ClrDisp()*)

```
void  Task_ClrDisp(void *data)
{
  INT8U       err;
  INT8U       errSem;

  data = data;

  for (;;)
  {
    OSSemPend(TransmisionesListasSem,0,&errSem);

    PC_ElapsedStart();

    OSTimeDlyHMSM(0, 0, 0, 200);
  }
}
```

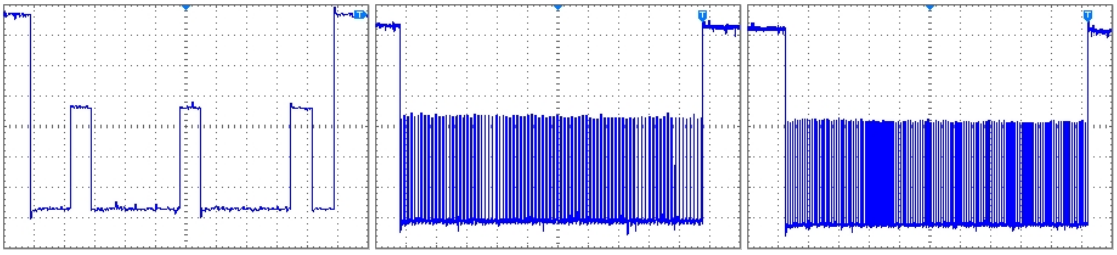
6. Ensayos

Los ensayos realizados sobre todo el sistema funcionando son principalmente sobre el sistema de comunicación entre el *Sensor* y el bloque de *Control*. La forma de las señales sobre el canal serial en el caso ideal se observan en la Figura 6. Para realizar el testeo del sistema se ha utilizado un *Dimmer* que nos permitirá modificar la intensidad de luz que incide sobre el sensor fotoeléctrico. La Tabla 3 presenta varias mediciones realizadas a diferentes niveles de atenuación del *Dimmer*. Se contrasta el periodo adquirido por el dsPIC® (segunda columna) y la medición del mismo con un osciloscopio. (tercera columna).

Muestra	**Periodo**	
	dsPIC®	**Medido**
Muestra 1	3.5	368 μseg.
Muestra 2	29.5	2873 μseg.
Muestra 3	85	8622 μseg.
Muestra 4	120	12.1 μseg.

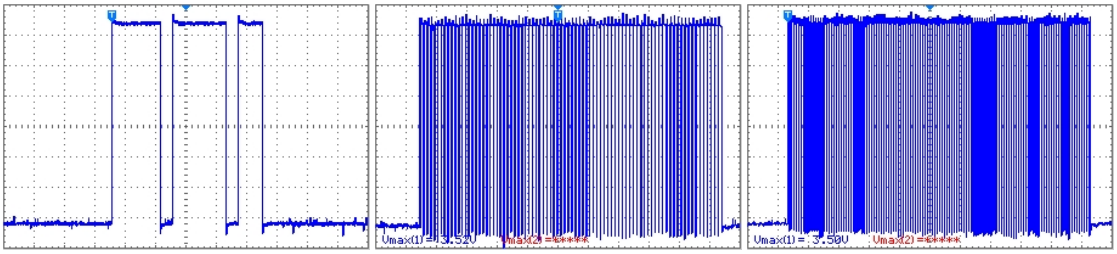
Tabla 3: Captura de datos del sistema de sensor a diferentes niveles de intensidad de luz.

Por último se muestran las capturas desde el osciloscopio para las diferentes muestras presentadas en la Tabla 3.



(a) Muestra 1 (b) Muestra 2 (c) Muestra 3

Figura 16: Señales tomadas sobre la línea de comunicación del sensor/control.



(a) Muestra 1 (b) Muestra 2 (c) Muestra 3

Figura 17: Señales tomadas en la salida del sistema de control (colector $Q2$).