Capítulo 7
Software del ControlDSC

En este capítulo:

La organización del software del SensorDSC

En este capítulo se verá cómo está organizado el software que se ejecuta en el ControlDSC. Éste sigue la misma estructura que la del SensorDSC, por lo que en este capítulo sólo se centrará en las diferencias principales. Para el diseño de la arquitectura, se siguió la aproximación *Outside-In* como se explicó en el capítulo 5. Una vez descompuesto la aplicación se decidió clasificar los distintos módulos en capas como se muestra en la Figura 1.

En la parte más baja se encuentra el kernel del RTOS. Este es el sistema operativo en tiempo real que ofrece servicios básicos de gestión de memoria, tareas, semáforos, *mutex*, *mailbox*, etc.

La capa CSP (*Chip Support Package*) se encarga de manejar los periféricos del microcontrolador. Aquí se definen las funciones necesarias para contralar los periféricos añadiendo un grado de abstracción del hardware. Esta capa está íntimamente ligada al hardware del microcontrolador y utiliza los servicios del sistema operativo. También se implementan las rutinas de servicios de interrupción que controlan los periféricos. Esta capa da servicios a la capa superior, ocultando los detalles del hardware.

La capa BSP (*Board Support Package*) se encarga de manejar los distintos componentes de la placa que interaccionan con el microcontrolador del ControlDSC. En esta capa ya se empiezan a crear tareas siguiendo la aproximación *Outside-In*. Estas tareas utilizan los servicios del CSP y del kernel para implementar una arquitectura software que permite dar servicios a la capa superior de aplicación. Esta capa principalmente se ocupa de mantener actualizado las señales de los servos y payload, comunicase continuamente con el SensorDSC para la lectura y escritura de los servos y de las comunicaciones.

La arquitectura implementada, formada por las capas del BSP, el CSP y el kernel se le ha llamado SupersonicOS.

La capa de aplicación del ControlDSC básicamente se encarga de inicializar el sistema y las comunicaciones. Una de las funciones del ControlDSC es de seguridad. El software debe ser sencillo para evitar fallos ya que éste es quién se encarga de realizar la conmutación de manual

a automático. A esta función se le ha dado la máxima prioridad, ya que en caso de fallo, siempre se puede tomar el control manual.

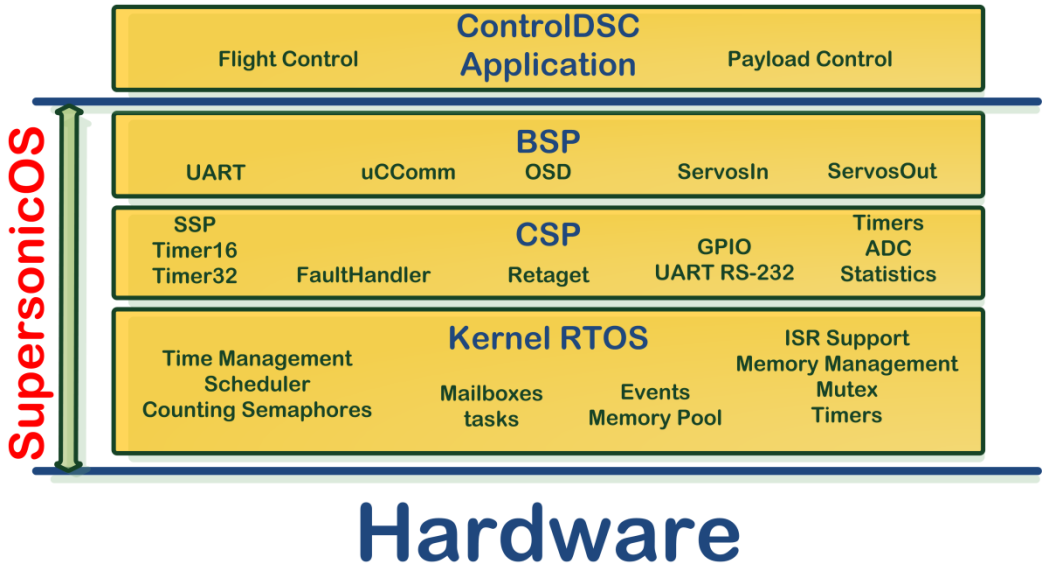

Figura 1 – Software visto en capas.

StartUp

La programación del ControlDSC es muy parecida a la del SensorDSC. Incorpora un núcleo Cortex-M3 y una serie de periféricos como se muestra en la Figura 2. Al ser el mismo núcleo, la programación del NVIC e interrupciones es exactamente igual. La única diferencia está en la arquitectura de los periféricos que cambia al tratarse de otra familia. La configuración y utilización de los periféricos cambia un poco, pero se sigue manteniendo la arquitectura software.

El proceso de inicialización del sistema comienza con la configuración del reloj que alimenta al núcleo y los distintos periféricos. Un diagrama de bloques de la unidad de generación de reloj (CGU) del LPC13xx se puede ver en la Figura 3.

El LPC131x incluye tres osciladores independientes. Estos son el oscilador sistema, el oscilador RC interno (IRC), y el oscilador de Watchdog. Cada oscilador puede ser utilizado para más de un propósito según se requiera en una aplicación particular.

Después de un RESET el LPC131x operará desde el oscilador RC interno hasta que sea conmutado por software. Esto permite al sistema operar sin ningún cristal externo y que el código del bootloader pueda operar a una frecuencia conocida.

El registro SYSAHBCLKCTRL controla como se lleva el reloj del sistema a los distintos periféricos y memorias. El UART, SSP0/1, el SysTick timer, y el reloj del ARM Trace tienen divisores de reloj individuales para derivar los relojes de los periféricos del reloj principal.

Antes de iniciar el sistema operativo, hay que configurar el microcontrolador y los periféricos para que funcione correctamente.

Primero se comienza configurando la pila y el *heap*. Para el MSP se definió un tamaño de 0x200 bytes y el *heap* se definió con un tamaño de 0x0 bytes. Para trabajar con memoria dinámica se utilizan los servicios que ofrece el RTOS.

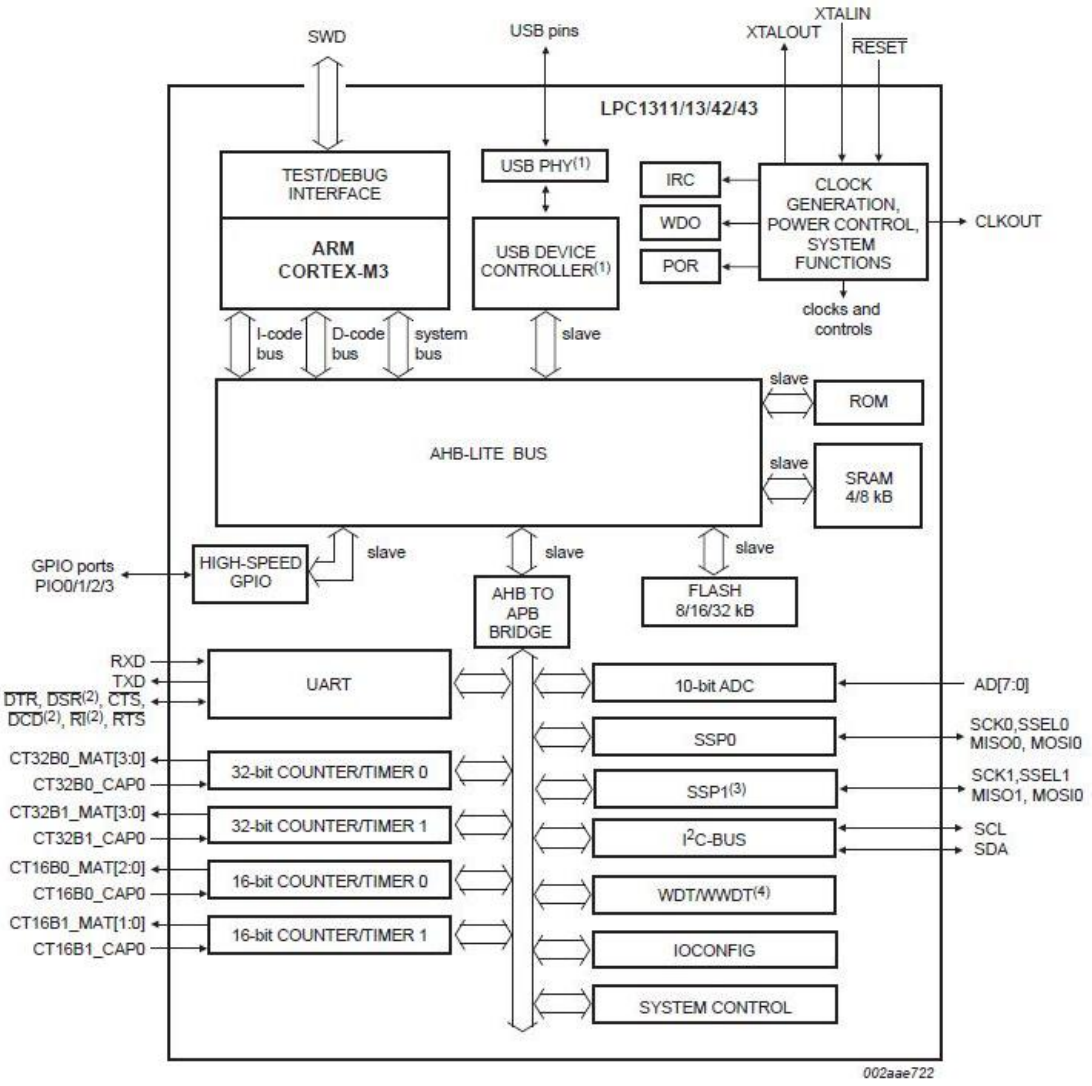

Figura 2 – Diagrama de bloques del LPC1313/01.

Una vez configurado los espacios de pila y *heap* se pasa a configurar el reloj de los periféricos del microcontrolador, el PLL y el módulo de *Flash Accelerator*. A continuación se indican la configuración de cada registro.

- System Oscillator Control Register (SYSOSCCTRL)
  - BYPASS. System Oscillator Bypass: Disable.
  - FREQRANGE. System Oscillator Frequency Range: 1 - 20 MHz.
- Watchdog Oscillator Control Register (WDTOSCCTRL)
  - DIVSEL. Select Divider for Fclkana: 0.
  - FREQSEL. Select Watchdog Oscillator Analog Output Frequency (Fclkana): Undefined.
- System PLL Control Register (SYSPLLCTRL)
  - MSEL. Feedback Divider Selection: 5.
  - PSEL. Post Divider Selection: P = 2.
- System PLL Clock Source Select Register (SYSPLLCLKSEL)
  - SEL. System PLL Clock Source: System Oscillator.
- Main Clock Source Select Register (MAINCLKSEL)
  - SEL. Clock Source for Main Clock: System PLL Clock Out.

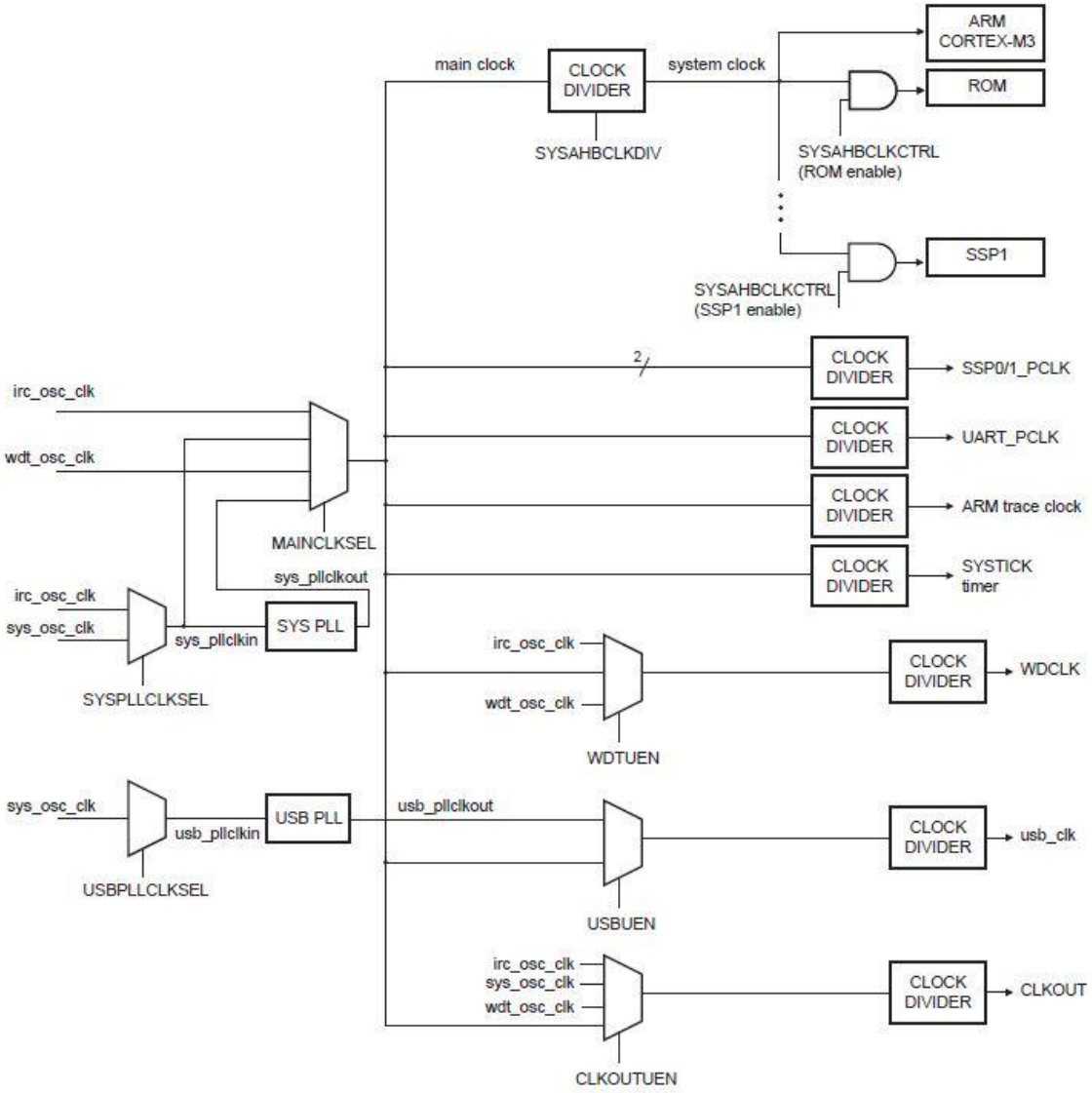

Figura 3 – Diagrama de bloques del LPC13xx CGU.

- System AHB Clock Divider Register (SYSAHBCLKDIV)
  - DIV. System AHB Clock Divider: 1.
- USB PLL Control Register (USBPLLCTRL)
  - MSEL. Feedback Divider Selection: 3
  - PSEL. Post Divider Selection: P = 2.
- USB Clock Source Select Register (USBCLKSEL)
  - SEL. System PLL Clock Source: System Oscillator.
- USB Clock Source Select Register (USBCLKSEL)
  - SEL. System PLL Clock Source: USB PLL out.
- USB Clock Divider Register (USBCLKDIV)
  - DIV. USB Clock Divider: 0.

En la Figura 4 se muestran la configuración del PLL y un esquema de la distribución del reloj dentro del microcontrolador.

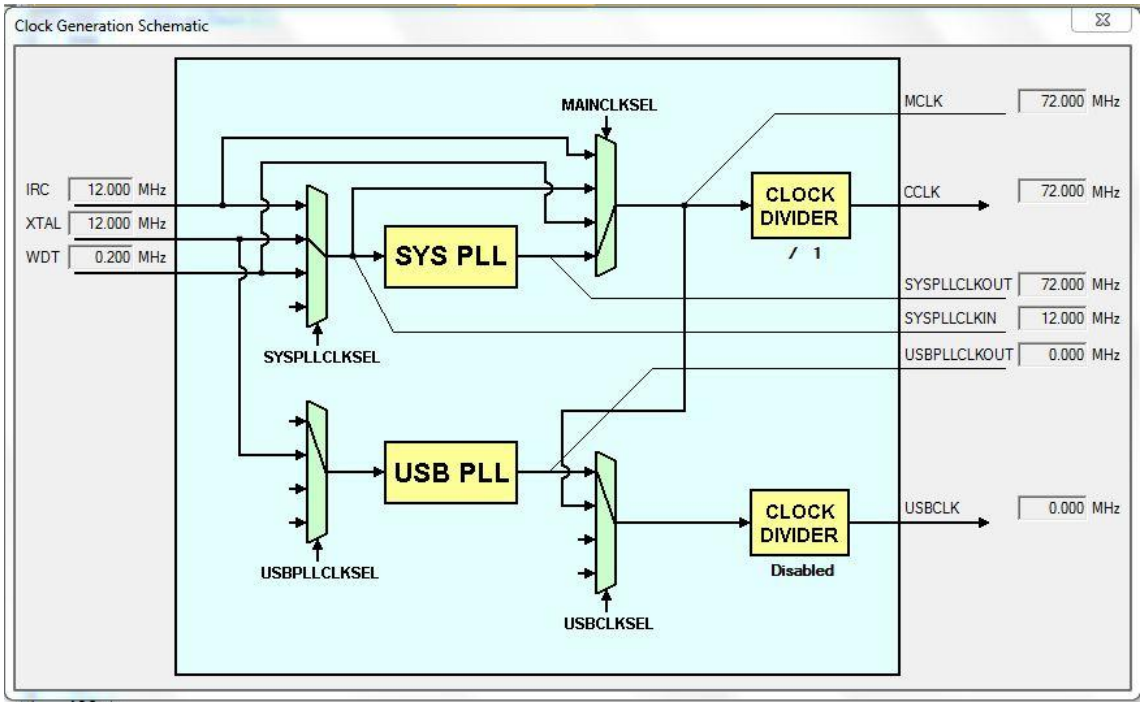

Figura 4 – Esquema de la configuración de la señal de reloj en el ControlDSC.

A continuación activa el reloj para el GPIO y el IOCON. Éste último es el encargado de configurar la función de los pines del microcontrolador. Finalmente se llama al kernel del RTOS.

Chip Support Package

Al igual que en el SensorDSC, el CSP se encarga del control de los periféricos incluidos en el microcontrolador. Para ello se crean los drivers que controlan estos periféricos, utilizando los servicios del sistema operativo para conseguir el mayor rendimiento.

UART

El ControlDSC dispone de un UART para las comunicaciones. Las principales características son las siguientes:

- FIFOs de transmisión y recepción de 16 bytes.
- Se puede fijar el umbral en la FIFO de recepción para generar una interrupción en 1, 4, 8 y 14 bytes.
- Generador interno de tasa de bit, incluyendo un divisor fraccional para mayor versatilidad.
- Soporte para control de flujo por software.
- Soporte para RS-485.
- Control completo de modem disponible con *handshaking*.

La configuración básica del UART se realiza usando los siguientes registros.

1. Pines. Se debe configurar con el registro IOCONFIG antes de activar el reloj para el UART en el caso de utilizar LPC1311/13/42/43. Para el LPC1311/01 y LPC1313/01 no se requiere ninguna secuencia especial. Aunque en el proyecto se utiliza el LPC1313/01, se utiliza esta secuencia de activación para evitar problemas de compatibilidad futura.
2. Power. En el registro SYSAHBCLKCTRL se activa el bit 12.

3. Reloj. Para habilitar el reloj del UART se utiliza el registro UARTCLKDIV.

La arquitectura software implementada es la misma en que el SensorDSC en modo interrupción explicado en el capítulo anterior, por lo que no se va a entrar en detalles. Se recuerda el esquema de la arquitectura en la Figura 5 y Figura 6.

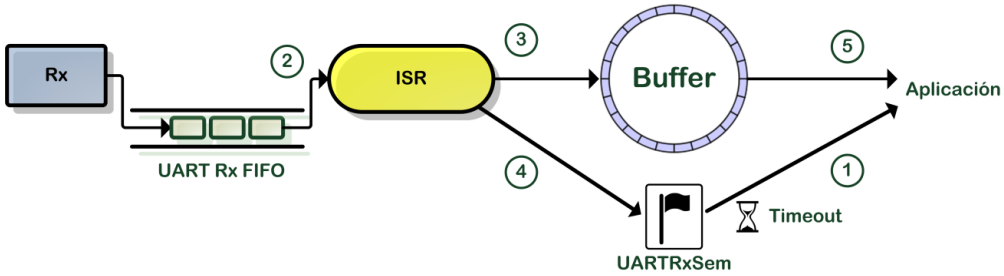

Figura 5 – Serial I/O con buffer y semáforo, recibiendo datos.

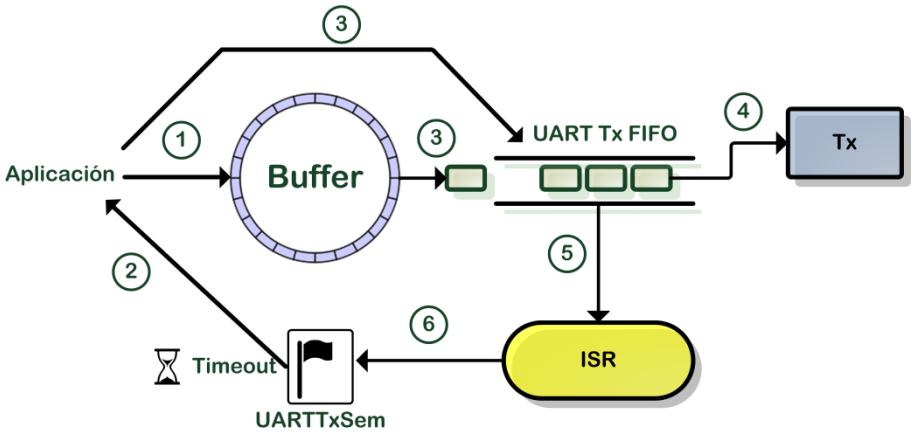

Figura 6 – Serial I/O con buffer y semáforo, transmitiendo datos.

Servos

Un servomotor de modelismo, conocido generalmente como servo o servo de modelismo, es un dispositivo actuador que tiene la capacidad de ubicarse en cualquier posición dentro de su rango de operación, y de mantenerse estable en dicha posición. Está formado por un motor de corriente continua, una caja reductora y un circuito de control, y su margen de funcionamiento generalmente es de menos de una vuelta completa. Su aspecto es como se muestra en la Figura 7.

Figura 7 – Servo futaba.

El componente principal de un servo es un motor de corriente continua, que realiza la función de actuador en el dispositivo: al aplicarse un voltaje entre sus dos terminales, el motor gira en un sentido a alta velocidad, pero produciendo un bajo par. Para aumentar el par del dispositivo, se utiliza una caja reductora, que transforma gran parte de la velocidad de giro en torsión.

Dependiendo del modelo del servo, la tensión de alimentación puede estar comprendida entre los 4 y 8 voltios. El control de un servo se reduce a indicar su posición mediante una señal PWM: el ángulo de ubicación del motor depende de la duración del nivel alto de la señal.

Cada servo, dependiendo de la marca y modelo utilizado, tiene sus propios márgenes de operación. Por ejemplo, para algunos servos los valores de tiempo de la señal a nivel alto están entre 1 y 2 ms, que posicionan al motor en ambos extremos de giro (-90° y +90°, respectivamente). Los valores de tiempo de nivel alto para ubicar el motor en otras posiciones se hallan mediante una relación completamente lineal: el valor 1,5 ms indica la posición central, y otros valores de duración del pulso dejarían al motor en la posición proporcional a dicha duración.

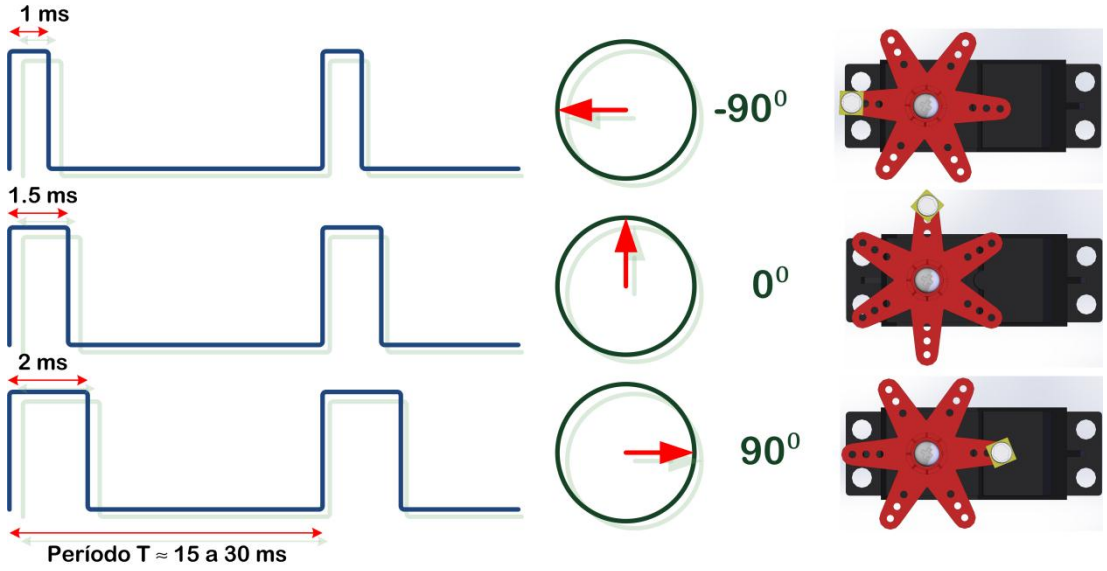

Figura 8 – Señal PWM para controlar un servo analógico.

El ControlDSC es capaz de leer hasta 12 señales PWM independientes provenientes de un receptor de RC y de generar hasta 20 señales PWM gracias a la ayuda de un hardware externo.

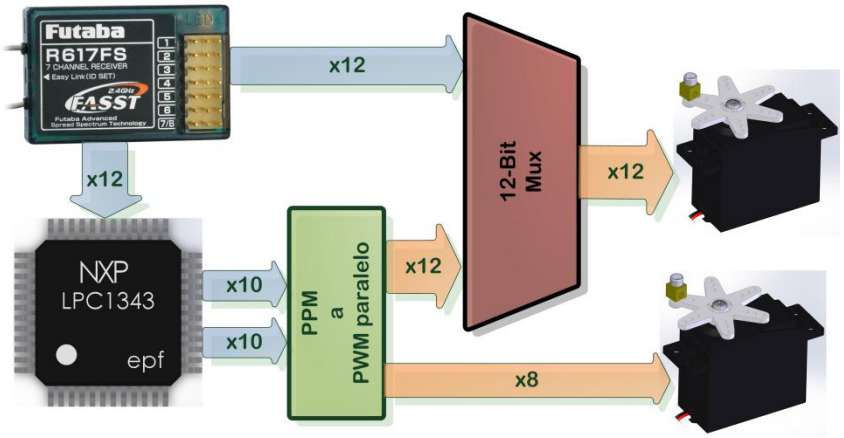

Figura 9 – Diagrama de bloques de la arquitectura hardware del control de servos.

Los 12 primeros canales generados son conmutados por hardware con los canales que provienen del receptor a través de un multiplexor. Los ocho canales restantes pasan directamente a los servos de salida. En la Figura 9 se puede ver de forma esquemática como se conecta el microcontrolador con los servos de entrada y de salida.

Para generar los 20 canales de salida, el ControlDSC modula dos señales PPM[1] con la información de los 20 canales. Cada señal PPM lleva información de 10 canales de servos de salida. Con una lógica externa contenida en la placa se extrae la información contenida en las dos señales PPM y se crean canales independientes PWM. La señal PPM se crea colocando de forma secuencial los canales PWM como se ilustra en la Figura 10.

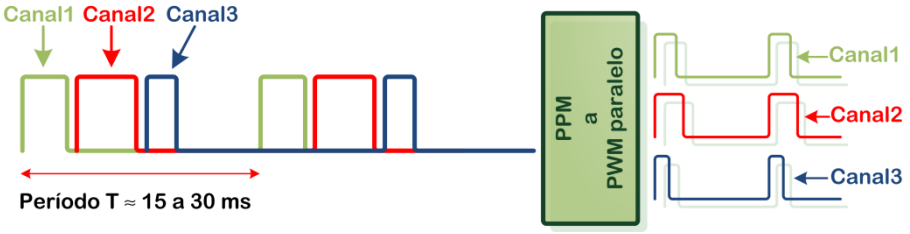

Figura 10 – Extracción de canales de una señal PPM.

Timer32

El ControlDSC dispone de dos *timers* de 32 bits. Las principales características son.

- Dos contadores/temporizadores de 32 bits con un pre-escalador de 32 bits programable.
- Operación como contador o temporizador.
- Un canal de captura de 32 bits que puede tomar una instantánea del valor del temporización cuando ocurre una transición en la señal de entrada. Un evento de captura también puede generar opcionalmente una interrupción.
- Cuatro registros de coincidencia (match) de 32 bits que permiten:
  - Operación continua con la generación de interrupción opcional en coincidencia.
  - Detener temporizador en coincidencia con la generación de interrupción opcional.
  - Reiniciar el temporizador en coincidencia con la generación de interrupción opcional.
- Hasta cuatro salidas externas correspondientes a los registros de coincidencia (*match registers*), con las siguientes capacidades.
  - Fijar a nivel bajo cuando hay coincidencia.
  - Fijar a nivel alto cuando hay coincidencia.
  - Conmutar cuando hay coincidencia.
  - No hacer nada cuando hay coincidencia.
- Para cada *timer* hasta cuatro registros de coincidencia.

La configuración básica del *timer* se realiza utilizando los siguientes registros:

1. PINS. Los pines de CT32B0/1 se deben configurar con el registro IOCONFIG.
2. Power y reloj. En el registro SYSAHBCLKCTRL se activan los bits 9 y 10.

La clase que define el driver del *Timer* para el CSP se define a continuación.

[1] Modulación utilizada para transmitir la información de los canales por RF en los sistemas RC. Para más información se puede consultar el apartado “Radio Control (R/C)” en la página 9.22 del libro “The ARRL HANDBOOK FOR RADIO COMMUNICATIONS 2008” [9].

```
Class TIMER32 {
   uint8_t timer_num;
   public:
      TIMER32(uint8_t timer_num);
      void delay32Ms(uint32_t delayInMs);
      void enable();
      void disable();
      void reset();
      void init(uint32_t timerInterval);
};
```

Este periférico se utilizó principalmente para hacer cuentas de tiempo para leer los servos de entrada como se verá en el apartado GPIO.

Timer16

El ControlDSC dispone de dos *timers* de 16 bits. Las principales características son.

- Dos contadores/temporizadores de 16 bits con un pre-escalador de 16 bits programable.
- Operación como contador o temporizador.
- Un canal de captura de 16 bits que puede tomar una instantánea del valor del temporización cuando ocurre una transición en la señal de entrada. Un evento de captura también puede generar opcionalmente una interrupción.
- Cuatro registros de coincidencia (match) de 16 bits que permiten:
    - Operación continua con la generación de interrupción opcional en coincidencia.
    - Detener temporizador en coincidencia con la generación de interrupción opcional.
    - Reiniciar el temporizador en coincidencia con la generación de interrupción opcional.
- Hasta tres (CT16B0) o dos (CT16B1) salidas externas correspondientes a los registros de coincidencia (*match registers*), con las siguientes capacidades.
    - Fijar a nivel bajo cuando hay coincidencia.
    - Fijar a nivel alto cuando hay coincidencia.
    - Conmutar cuando hay coincidencia.
    - No hacer nada cuando hay coincidencia.
- Para cada *timer* hasta cuatro registros de coincidencia.

La configuración básica del *timer* se realiza utilizando los siguientes registros:

3. PINS. Los pines de CT16B0/1 se deben configurar con el registro IOCONFIG.
4. Power y reloj. En el registro SYSAHBCLKCTRL se activan los bits 7 y 8.

La clase que define el driver del *Timer* para el CSP se define a continuación.

```
Class TIMER16 {
   uint8_t timer_num;
   public:
      TIMER16(uint8_t timer_num);
      void delayMs(uint32_t delayInMs);
      void enable();
      void disable();
      void reset();
      void init(uint16_t timerInterval);
};
```

En este proyecto, los dos *timers* de 16 bits se utilizan para generar las señales PPM. Cada *timer* genera una señal PPM con diez canales de información. El *timer* CT16B0 se encarga de

los canales que van del 0 al 10 y el CT16B1 de los canales que van del 11 al 20. Ambos *timers* se ajustan para tener una resolución de 1µseg y con los registros de *match* van generando la señal PPM. En la Figura 11 se puede ver de forma simplificada cómo se generan cada canal. La ISR ajusta el ancho de cada pulso de acuerdo con el valor almacenado en el array `ServoOut[]`. Si se desea cambiar la posición de un servo en particular, la aplicación sólo tiene que modificar el valor correspondiente en el array `ServosOut[]` y la ISR se encarga de forma automática de actualizar este valor.

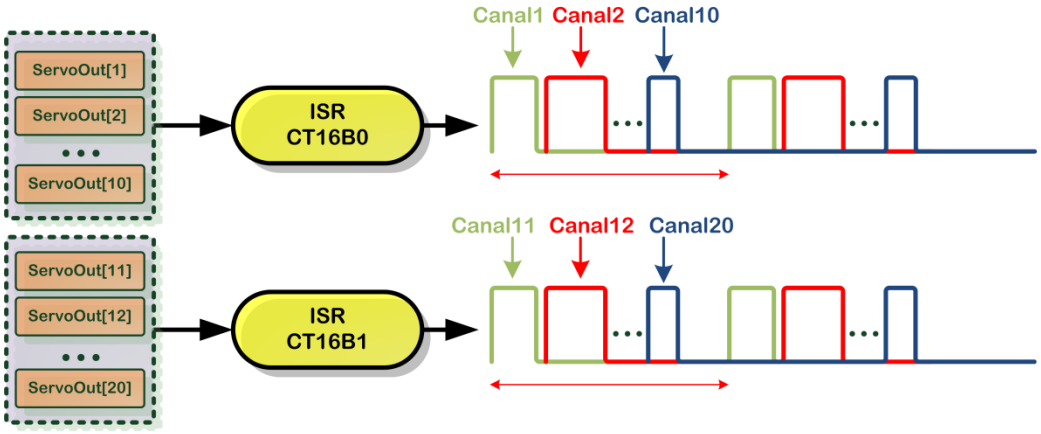

Figura 11 – Generación de las señales PPM para los servos de salida.

GPIO

El CSP del puerto paralelo se encarga principalmente de controlar las seis entradas salidas de propósito general y de leer los doce servos de entrada. Cada canal de los servos de entrada está conectado a un pin del microcontrolador. Estos pines se configuran para que generen una interrupción cada vez que cambie su valor. En la ISR de cada pin, se mide el ancho del pulso de cada pin utilizando uno de los *timers* de 32. Este *timer* se configura para incrementarse cada 1µseg, por lo que el valor calculado por la ISR está en unidades de µsegundos y con una resolución de 1µseg. Este valor se almacena en el array `ServoIn[]`.

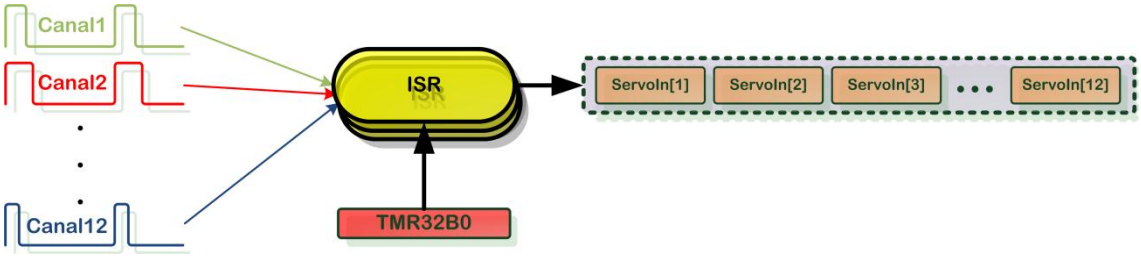

Figura 12 – Lectura de los servos de entrada.

SSP

El periférico SSP (*Synchronous Serial Port*) es un controlador de puerto serie síncrono capaz de operar en SPI, 4-wire SSI o *Microwire bus*. Puede interactuar con múltiples maestros y esclavos en el bus. Durante una transferencia, sólo se pueden comunicar un maestro con un esclavo. Las transferencias de datos son principalmente full dúplex, con tramas de 4 a 16 bits de datos.

El ControlDSC tiene dos periféricos SSP. En este proyecto, los periféricos SSP se configuran como SPI. Las principales características del periférico son las siguientes:

- Comunicación Full Duplex, serial y síncrona.
- SPI maestro o esclavo.

- La máxima tasa de bit de datos es un octavo del reloj del periférico (*peripheral clock rate*).
- El periférico SSP puede hacer transferencias de 4 a 16 bits.
- El SSP dispone de una FIFO de tamaño 8 frames tanto para transmisión como para recepción.

El SPI se configura usando los siguientes registros:

1. PINs. Los pines del SSP se deben configurar en el registro IOCONFIG. La función de SCK0 también debe configurarse en el registro IOCON_SCK0_LOC.
2. Power. En el registro SYSAHBCLKCTRL se activa el bit 11.
3. Reloj. Escribiendo en el registro SSP0CLKDIV se habilita el reloj para el periférico SSP.
4. Reset. Antes de acceder al bloque SSP, asegurarse que el campo SSP_RST_N bits (bits 0 y 2) del registro PRESETCTRL se han fijado a 1. Esto desactiva la señal de reset para el bloque SSP.

A continuación se muestra la clase del driver SSP del CSP. Esta clase implementa las funciones necesarias para la configuración y realizar transferencia de datos con el SSP en modo interrupción.

```
class SSP {
   int interfaceNum;
   public:
      SSP(int portNum);
      void init(void);
      void format(int bits, int mode = 0);
      void frequency(int hz = 1000000);
      void transfer(uint16_t *Wbuf, uint16_t *Rbuf, uint32_t TransferSize);
      void LoopBackModeOn();
      void LoopBackModeOff();
      void irq_SetPriority(uint32_t priority);
      OS_RESULT wait_transfer(U16 timeout);
};
```

La arquitectura software implementada para el driver del SSP es la misma que se implementó para el SensorDSC con manejo de interrupciones ya que el ControlDSC no dispone de DMA. En la Figura 13 se recuerda la arquitectura implementada.

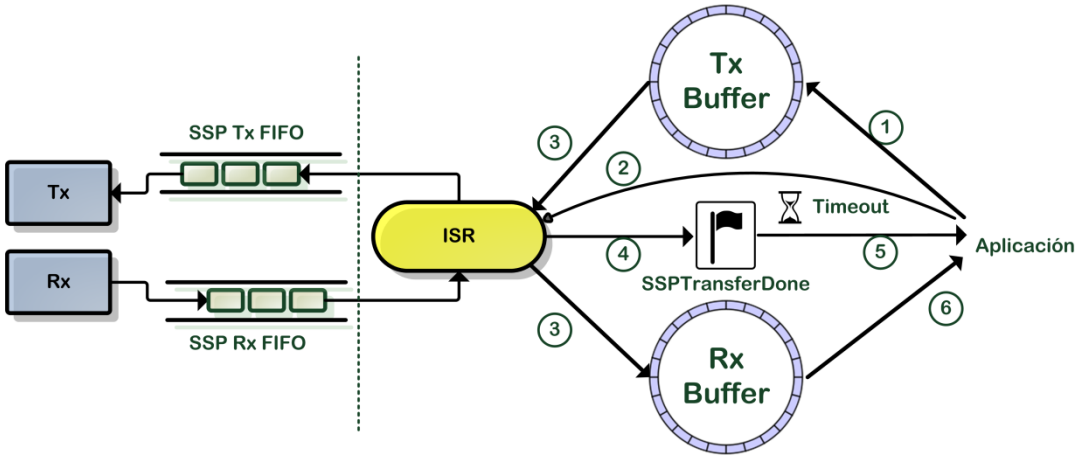

Figura 13 – Arquitectura de la transferencia de datos por interrupción en el SSP.

Estadísticas y FaultHandlers

El manejo de las estadísticas y de fallos es exactamente igual que en el SensorDSC, por lo que no se repite en este capítulo.

Board Support Package

El *Board Support Package* (BSP) es el nombre común para todo el código específico de control del hardware de la placa. La arquitectura implementada para el UART y el SSP es exactamente la misma que en el SensorDSC por lo que no se va a repetir en este capítulo. Principalmente este apartado se va a centrar en la lectura y control de los servos.

BSPServoIn

El BSP para los servos de entrada se encarga de configurar los pines del puerto paralelo, activar las interrupciones y configurar el *timer* CT32B0 que utiliza la ISR para hacer las cuentas de tiempo. También proporciona los métodos para leer los servos de entrada, tanto el ancho de pulso en µsegundos, como normalizado entre -1 y +1. La interfaz de aplicación se define a través de la clase SERVOIN como se muestra a continuación.

```
class SERVOIN {
   public:
      void init();
      uint16_t read(char servoN);
      float read(char servoN, uint16_t limiteSuperior, uint16_t centro,
                 uint16_t limiteInferior);
      void read_complete(uint16_t *servoptr);
};
```

El método `init()` se encarga de inicializar y configurar los pines, las interrupciones y el timer. El método `uint16_t read(char servoN)` devuelve el valor del ancho de pulso del servo `servoN`. El otro método `read(…)` devuelve un valor normalizado entre -1 a +1 del `servoN`. Por último el método `read_complete(…)` almacena la lectura actual de los 12 servos en el buffer apuntado por `servoptr`.

La conmutación de manual a gestionado por el ControlDSC lo controla la ISR. Cuando la ISR lee el canal 10 de los servos de entrada, decide conmutar a controlado por el ordenador de abordo si el ancho del pulso es superior a 1700 µsegundos y conmuta a manual cuando el ancho del pulso es inferior a 1300 µsegundos.

BSPServoOut

El BSP para los servos de salida se encarga de controlar los servos de salida y que se mantengan dentro de su límite de operación. Para ello se crea una tarea que se ocupa de leer el valor que ordena la aplicación, y si sale fuera de los límites de operación del servo, ésta tarea lo limita a su rango de operación. La clase SERVOSOUT define el driver del BSP.

```
class SERVOSOUT {
   static char ServosOutTaskExist;
   public:
      void init();
      void setPosicion(unsigned int n, short pos);
      void setPosicion(unsigned int n, float pos);
      short getPosicion(unsigned int n);
      void setDelta(unsigned int n, short delta);
      short getDelta(unsigned int n);
      void setCentro(unsigned int n, short centro);
      short getCentro(unsigned int n);
      void setLimite(unsigned int n, short *limite);
};
```

En la Figura 14 se muestra de forma esquemática la arquitectura interna del driver de los servos de salidas del BSP. Éste está compuesto principalmente por un array de 20 elementos de tipo `ServosOutTypedef` y una tarea. Cada elemento del array define la posición actual del

servo, los límites del servo y otros parámetros internos. Esto se ha hecho así, para permitir conectar servos de distintos fabricantes en cada canal y que sean independientes unos de otros. La tarea se encarga de analizar la posición en la que la aplicación desea que se encuentre el servo, y evalúa los límites del servo. Si se sobrepasa los límites se limita para que el servo no sufra y no haya exceso de consumo de corriente.

La forma en que el BSP interactúa con las ISR del CSP es a través de los eventos proporcionados por el RTOS. Cada vez que la ISR termina con el décimo canal, envía un evento a la tarea que está esperando por dos eventos. De esta forma, se notifica a la tarea que en ese momento puede actualizar los valores de los servos que va a leer la ISR en el siguiente periodo. El evento 0x0001 corresponde con la ISR que controla los servos del 1 al 10, y el evento 0x0002 corresponde con la ISR que controla los servos del 11 al 20.

```
typedef struct {
   volatile short posicion;
   volatile short delta;
   volatile short limite[2];
   volatile short centro;
} ServosOutTypedef;
volatile ServosOutTypedef ServosOut[20];
```

El BSP permite escribir directamente en el array `ServoOut[]` de tipo `ServosOutTypedef` y también proporciona métodos para facilitar esta labor.

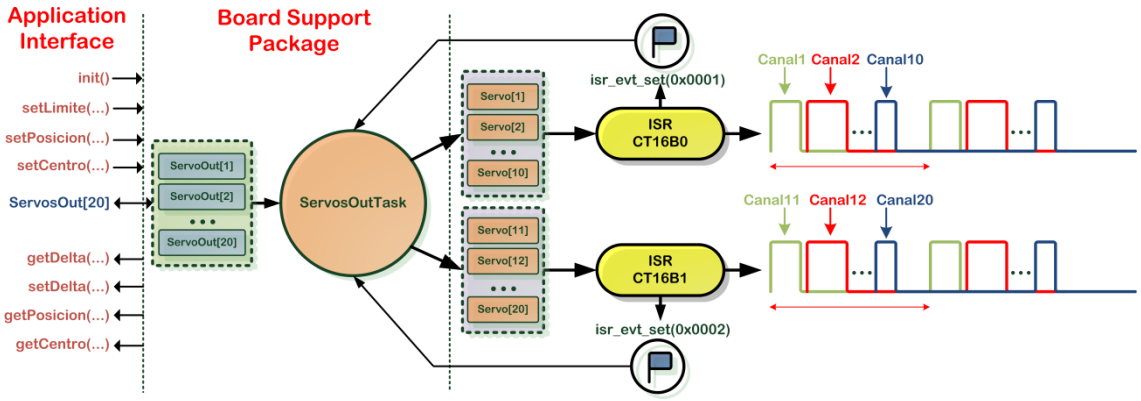

Figura 14 – BSP ServosOut.

Capa de aplicación

La capa de aplicación se encarga de las funciones relacionadas más con la aplicación y se olvida del funcionamiento del hardware. En este proyecto, la orden de control de los servos proviene del SensorDSC y la capa de aplicación básicamente se encarga de inicializar el sistema y los drivers del BSP. Como se ha visto, los drivers se encargan de forma automática de realizar su función sin necesidad de intervención de ningún elemento externo de control. No se ha hecho control de *Payload* debido a que para el proyecto no ha hecho falta, pero con la arquitectura implementada, la utilización sería muy fácil. Suponiendo que una tarea deseara mover el servo número 15 a la posición -0.5 el código sería el siguiente.

```
SERVOSOUT payload;
payload.setPosicion(15,-0.5);
```

Bibliografía

[1] Lizarraga Mariano, Curry Renwick y Elkaim Gabriel. “Reprogrammable UAV Autopilot”. Circuit Cellar. - Abril, Mayo de 2011.

[2] Rogers Robert M. “Applied Mathematics in Integrated Navigation Systems” 3ª ed. - AIAA EDUCATION SERIES.

[3] ARM. “Cortex-M3 Technical Reference Manual”.

[4] ARM. “Cortex-M3 Devices Generic User Guide”.

[5] Joseph Yiu. “The Definitive Guide to the ARM Cortex-M3”. NEWNES.

[6] NXP. “LPC1711/13/42/43 User Manual”.

[7] Andrew N. Sloss. “ARM System Developer’s Guide” Designing and Optimizing System Software. ELSEVIER.

[8] http://es.wikipedia.org/wiki/Servomotor_de_modelismo

[9] The ARRL HANDBOOK FOR RADIO COMMUNICATIONS 2008.