

Electrónica Digital II

Santiago Rúa Pérez, PhD.

18 de septiembre de 2022

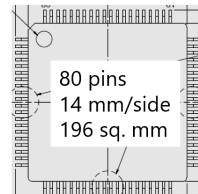
COMUNICACIONES SERIALES

Panorama general

- Comunicaciones seriales
 - Conceptos
 - Herramientas
 - Software: polling, interrupciones y buffering
- Comunicaciones SPI
 - Conceptos
 - Periféricos del K64
- Comunicaciones I²C
 - Conceptos
 - Periféricos del K64
- Comunicaciones UART
 - Conceptos
 - Periféricos del K64

Porque comunicaciones seriales?

- Aunque el tamaño de la palabra nativa para la CPU es de 32 bits, enviar todos los bits de una palabra simultáneamente tiene desventajas:
 - **Costo y peso:** paquete CI más grande, más cables, conectores más grandes
 - **Fiabilidad mecánica:** más cables implican más contactos de conexión para fallar
 - **Complejidad de sincronización:** algunos bits pueden llegar más tarde que otros debido a variaciones en la capacitancia y la resistencia a través de los conductores.
 - **Complejidad y potencia del circuito:** es posible que no desee tener 16 transmisores + receptores diferentes en el sistema
- La comunicación serial reduce el numero de señales requeridas

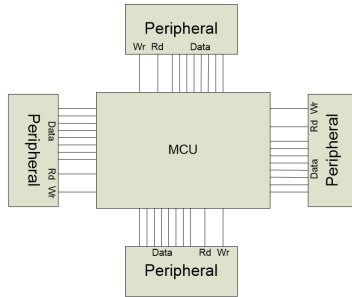


32 pins
5 mm/side
25 sq. mm



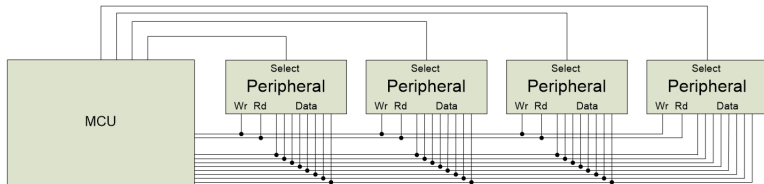
20 pins
1.94 mm/side
3.76 sq. mm

Ejemplo de sistema



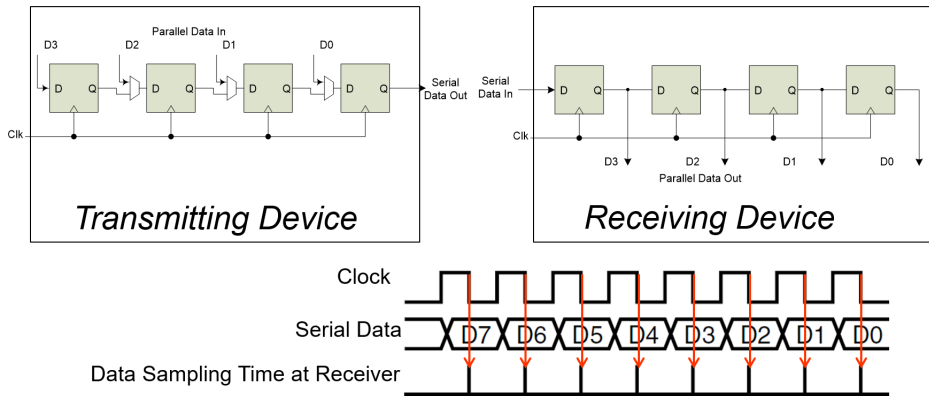
- Conexiones dedicadas punto a punto
 - Línea de datos paralelas, líneas de lectura y escritura entre el MCU y cada periféricos.
- Rápido, permite transferencias simultáneas
- Requiere muchas conexiones, area de PCB, y escalamiento malo.
 - Se necesitarían $4 * (8 + 2) = 40$ pines para el MCU solo comunicarse.

Buses paralelos



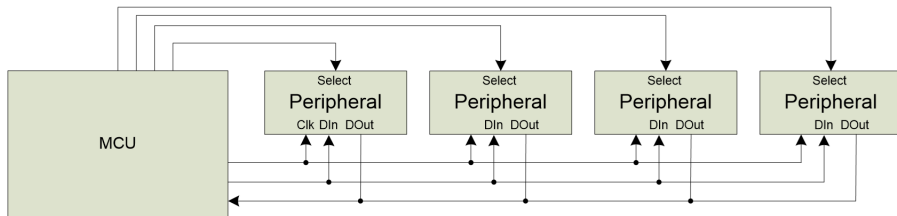
- Todos los dispositivos utilizan un bus para compartir datos, escribir y leer señales.
- El MCU utiliza líneas individuales para seleccionar el periférico con el que se está comunicando.
- El MCU requiere menos pines de datos, pero de todas formas uno por bit.
 - Se requieren $4 + (8 + 2) = 14$ pines para comunicarse.
- El MCU puede comunicarse con un periférico al tiempo.

Transmisión de datos serial sincrónica



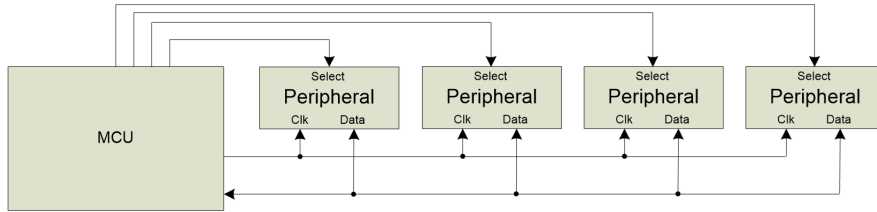
- Utiliza registros de desplazamiento y señales de clock para convertir formatos seriales y paralelos
- Sincrónica: una señal de reloj está de forma explícita acompañando a los datos.

Transmisión de datos serial sincrónica full duplex



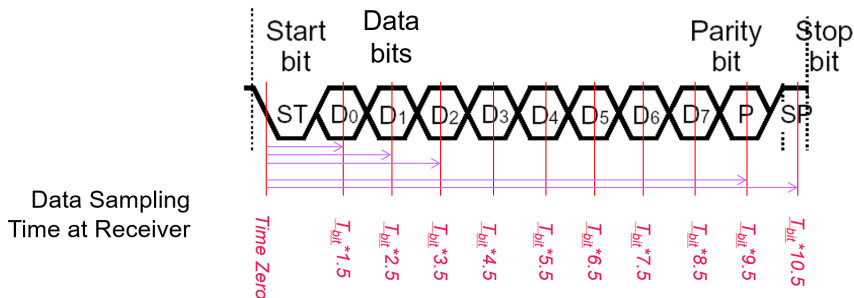
- Ahora se utilizan dos líneas seriales - una para lectura y otra para escritura
 - Permite simultáneamente tener envío y recepción de datos
 - Se requiere $4 + 3 = 7$ pines del MCU para comunicarse.

Transmisión de datos serial sincrónica half duplex



- Se comparten la línea de datos serial
 - Se requiere $4 + 2 = 6$ pines del MCU para comunicarse.
- No se permite envío y recepción de datos simultáneamente.

Transmisión de datos serial asincrónica



- Se elimina la línea de clock
- El transmisor y el receptor deben generar su clock local
- El transmisor debe adicionar un bit de start para indicar el inicio de una comunicación.
- El receptor detectar ese cambio de flanco, luego utiliza su referencia de tiempo para muestrear cada línea y extraer cada bit N en el tiempo $T_{bit} * (N + 1.5)$
- El bit de stop es usado para detectar errores de tiempo.

Especificaciones de la comunicaciones seriales

- Campos del data frame
 - Bit de start (un bit)
 - Datos (LSB a MSB): tamaños de 7, 8 y 9 bits.
 - Bit opcional de paridad para detección de errores.
 - Bit de stop, uno o dos bits.
- Todos los dispositivos deben usar la misma configuración
 - Eg: velocidades de comunicación de 300, 600, 1200, 2400, 9600, 14400, 19200 baud
- Protocolos sofisticados tienen mayor información el los data frame
 - Control de acceso al medio: cuando múltiples nodos están en el mismo bus, deben arbitrar para solicitar permisos de transmisión.
 - Información de direccionamiento: para cual nodo se pretenden mandar la información?
 - Carga útil más grande
 - Detección de errores superior, o información para corrección.
 - Solicitud para respuesta inmediata

Detección de errores

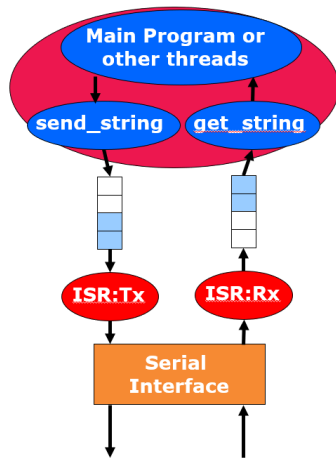
- Puede ser enviada información adicional para verificar que los datos fueron recibidos apropiadamente
- Se necesita especificar que tipo de paridad se espera: par, impar o ninguna
- El bit de paridad se pone 1 de tal forma que la cantidad total sea par (para paridad par) o impar (para paridad impar)
 - 01110111 tiene en total 6 bits en 1, entonces el bit de paridad se pone en 1 si se quiere paridad impar y 0 si se quiere par.
 - 01100111 tiene en total 5 bits en 1, entonces el bit de paridad se pone en 0 si se quiere paridad impar y 1 si se quiere par.
- El bit de paridad puede detectar errores si 1, 3, 5, 7 o 9 bits están corruptos, pero no detecta bits corruptos pares.
- Códigos de detección de errores mas fuertes existen (Cyclic redundancy Check CRC) y utilizan múltiples bits, lo que pueden ayudar a detectar más errores.
 - Usado en buses CAN, USB, Ethernet, Bluetooth, etc

Arquitectura de software para manejo de comunicaciones asincrónicas

- La comunicación es asincrónica para el programa
 - No se sabe que código esta ejecutando el programa cuando se reciben datos.
 - No se sabe cuando llega el próximo dato
 - No se sabe cuando el dato de salida se completa
 - Cuando ocurre un error
 - Se necesita algún tipo de sincronización entre el programa y la interfaz serial.
- Opciones
 - Polling
 - Esperar hasta que el dato este disponible
 - Sencillo pero ineficiente
 - Interrupción
 - La CPU interrumpe el programa cuando el dato está disponible
 - Eficiente, pero no complejo

Comunicaciones seriales e interrupciones

- Se desea tener múltiples hilos de control en el programa
 - Programa principal
 - ISR
 - Transmitir la actividad del ISR - se ejecuta cuando la interfaz serial está lista para mandar otro carácter.
 - Recibir la actividad ISR - ejecuta el serial cuando recibe un carácter.
 - Error en el ISR - ejecuta si hay un error.
- Se necesita una forma de bufferear la información entre hilos.
 - Solución: cola circular con punteros de cabeza y cola.
 - Uno para Tx, y otro para Rx.



Conexiones al ISR

- El ARM Cortex-M4 tiene dos IRQ para cada interfaz serial de comunicación.
- Dentro del ISR se necesita determinar que disparo la atención al servicio de la interrupción.

```
void UART2_RX_TX_IRQHandler() {  
    if (transmitter ready) {  
        if (more data to send) {  
            get next byte  
            send it out transmitter  
        }  
    }  
    if (received data) {  
        get byte from receiver  
        save it  
    }  
}  
  
void UART2_ERR_IRQHandler() {  
  
    if (error occurred) {  
        handle error  
    }  
}
```

Código para implementar colas

- **Enqueue:** el puntero `tail_ptr` apunta a la siguiente entrada libre (cola). Aquí debe ingresar el nuevo item.
- **Dequeue:** el puntero `head_ptr` apunta al item a retirar (cabeza).
- `#define` para el tamaño de la cola
- Una cola por dirección
 - ISR saca datos `tx_q` para transmitir
 - ISR carga datos `rx_q` para recibir.
- Otro hilos cargar y descargan datos de las colas.
- Se necesita ajustar el puntero al final del buffer para volverlo circular
 - Usar el operado módulo (%) si el tamaño de la cola no es potencia de dos.
 - Usar & si la cola es potencia de dos.

```
void UART2_RX_TX_IRQHandler() {  
    if (transmitter ready) {  
        if (more data to send) {  
            get next byte  
            send it out transmitter  
        }  
    }  
    if (received data) {  
        get byte from receiver  
        save it  
    }  
}  
  
void UART2_ERR_IRQHandler() {  
    if (error occurred) {  
        handle error  
    }  
}
```


Definir las colas

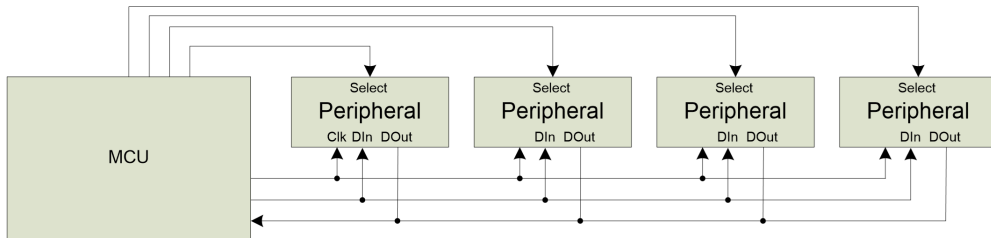
```
1 #define Q_SIZE (32)
2 typedef struct {
3     unsigned char Data[Q_SIZE];
4     unsigned int Head; // points to oldest data element
5     unsigned int Tail; // points to next free space
6     unsigned int Size; // quantity of elements in queue
7 } Q-T;
8
9 Q-T tx_q, rx_q;
10
11 void Q_Init(Q-T *q) {
12     unsigned int i;
13     for (i=0; i<Q_SIZE; i++)
14         q->Data[i] = 0; // to simplify our lives when debugging
15     q->Head = 0;
16     q->Tail = 0;
17     q->Size = 0;
18 }
19
20 int Q_Empty(Q-T *q) {
21     return q->Size == 0;
22 }
23
24 int Q_Full(Q-T *q) {
25     return q->Size == Q_SIZE;
26 }
```

Enqueue and Dequeue

```
1  int Q_Enqueue(Q-T *q, unsigned char d) {
2      // What if queue is full?
3      if (!Q_Full(q)) {
4          q->Data[q->Tail++] = d;
5          q->Tail %= Q_SIZE;
6          q->Size++;
7          return 1; // success
8      } else
9          return 0; // failure
10 }
11 unsigned char Q_Dequeue(Q-T *q) {
12     // Must check to see if queue is empty before dequeuing
13     unsigned char t=0;
14     if (!Q_Empty(q)) {
15         t = q->Data[q->Head];
16         q->Data[q->Head++] = 0; // to simplify debugging
17         q->Head %= Q_SIZE;
18         q->Size--;
19     }
20     return t;
21 }
22
23
```

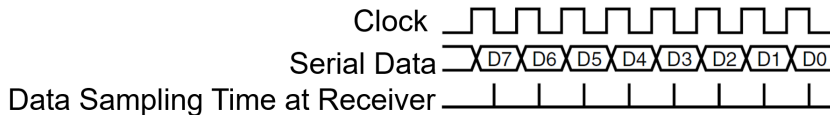
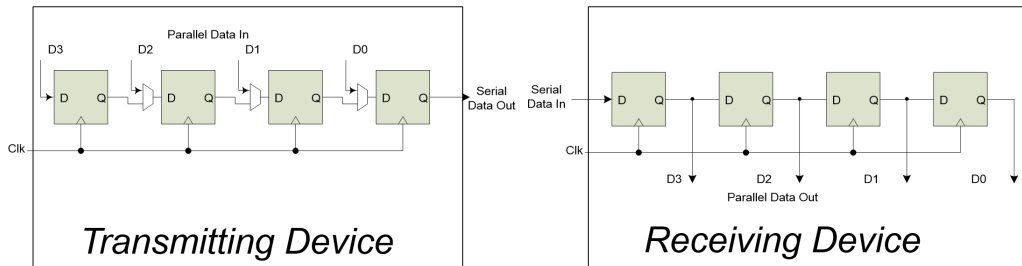
COMUNICACIONES SPI (Serial Peripheral Interface)

SPI - Arquitectura de hardware



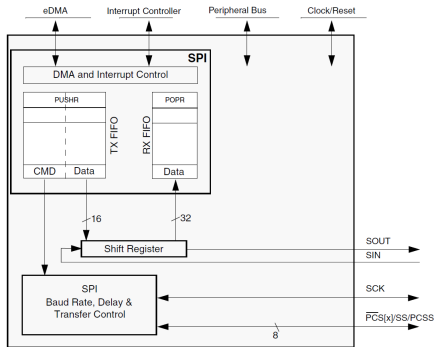
- Todos los chips comparten el bus de señales.
 - Clock SCK
 - Líneas de datos: MOSI (master out, slave in) y MISO (master in, slave out)
- Cada periférico tiene su propio selector (chip select CS)
 - El maestro activa la línea CS solo con el periférico que se está comunicando.

SPI - Transmisión serial



- Utiliza shift register y señales de clock para convertir entre serial y paralelo.
- **Sincrónico**: una señal de clock explícita va en conjunto con las líneas de datos.

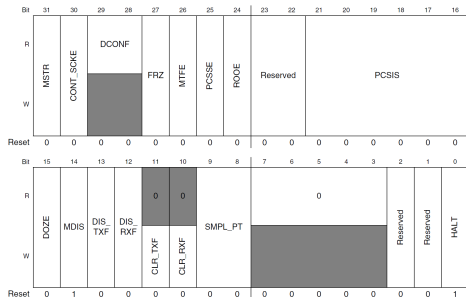
SPI - Diagrama y registros



- La comunicación SPI consiste en intercambiar datos entre el maestro y el esclavo. Al mismo tiempo se desplaza el de envío como el de recepción.
- El MCU K64F cuenta con tres SPI, los cuales a su vez cuentan con pilas para la transmisión y recepción de datos.
 - SPI0: 4 FIFO para Tx y Rx, SPI1 y SPI2: 1 FIFO para Tx y Rx

SPI - Registros

Module Configuration Register - SPI0_MCR



- MSTR: 1 modo maestro, 0 modo esclavo.
- FRZ: detiene la transferencia serial en modo config.
- PCSIS: el estado inactivo del chip select
- MDIS: habilitar o deshabilitar el módulo
- DIS_TXF: deshabilitar el Tx FIFO
- DIS_RXF: deshabilitar el Rx FIFO
- HALT: Para las transferencias

SPI - Registros

Clock y atributos de transferencia - SPI0_CTAR

Address: Base address + Ch offset + (4d × i), where i=0d to 1d

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|-------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| R | | | | | | | | | | | | | | | | |
| W | | | | | | | | | | | | | | | | |
| Reset | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| R | | | | | | | | | | | | | | | | |
| W | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

- DBR: dobla el baud rate.
- FMSZ: el número de bits a enviar más uno. El mínimo es 4.
- CPOL: polaridad del clock
- CPHA: fase del clock
- LSBFE: primero se transmite el menos significativo
- PBR: Baud rate prescaler.
- BR: Baud rate escaler.

SPI - Modos

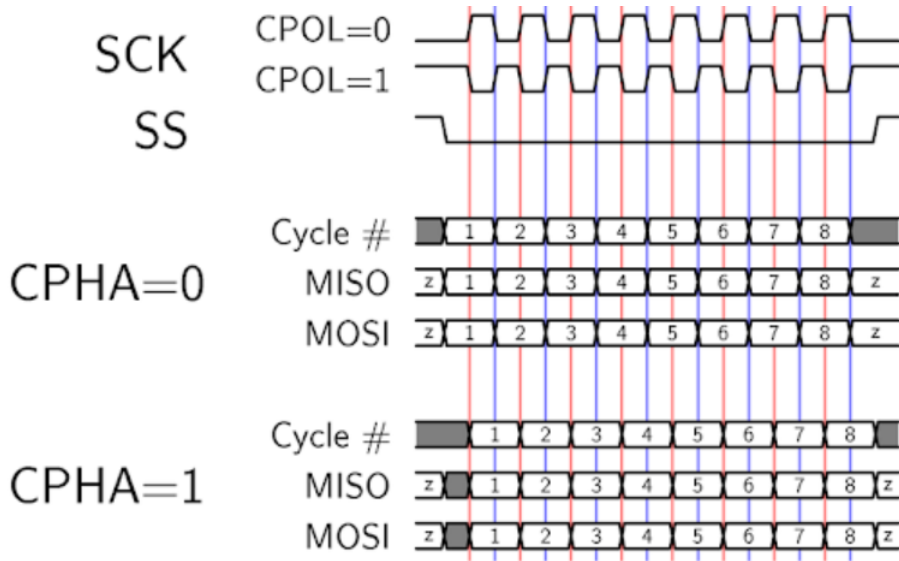
Address: Base address + Ch offset + (4d × i), where i=0d to 1d

| Bit | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|-------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| R | | | | | | | | | | | | | | | | |
| W | | | | | | | | | | | | | | | | |
| Reset | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| R | | | | | | | | | | | | | | | | |
| W | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

- DBR: dobla el baud rate.
- FMSZ: el número de bits a enviar más uno. El mínimo es 4.
- CPOL: polaridad del clock
- CPHA: fase del clock
- LSBFE: primero se transmite el menos significativo
- PBR: Baud rate prescaler.
- BR: Baud rate scaler.

SPI - Registros



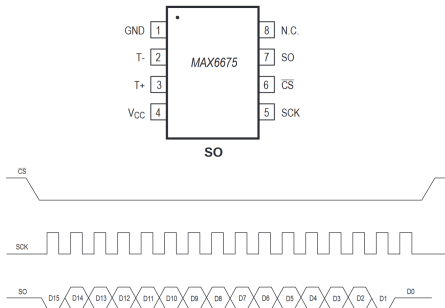
SPI - Ejemplo

Se requiere leer un termopar tipo k el cual esta conectado a través de un transmisor MAX6675 con salida o protocolo SPI

Solución:

- Revisar la hoja de datos del MAX 6675.
- Inicializar el clock del bus
- Inicializar los clocks y puertos del SPI.
- Inicializar los registros PCR para que los pines trabajen en modo SPI
- Configurar el SPI

SPI - Ejemplo



- Solo se requiere conectar la línea de datos MISO, no MOSI.
- El dataframe es de 16 bits. El dato se encuentra entre D14-D3 del más significativo al menos.
- D2 esta en alto cuando no está el termopar
- D1 está en bajo, y D0 tres estados.
- Los datos se capturan en flanco de bajada
- Todo cero corresponde a 0 °C y todos unos a 1023 °C.
- Máxima velocidad del clock 4.3 MHz

SPI - Ejemplo

Configuración del SPI

```
1 void Init_SPI0(void){
2     // Enable clock to SPI0 and PORTD
3     SIM->SCGC5 |= SIM_SCGC5_PORTD_MASK;
4     SIM->SCGC6 |= SIM_SCGC6_SPI0_MASK;
5
6     // Enable SPI and stops transfers SPI in debug mode
7     SPI0->MCR &= ~SPI_MCR_MDIS_MASK;
8     SPI0->MCR |= SPI_MCR_HALT_MASK;
9
10    // Set PTD0 as SPI0_PCS0 — ALT2
11    PORTD->PCR[0] &= ~PORT_PCR_MUX_MASK;
12    PORTD->PCR[0] |= PORT_PCR_MUX(2);
13
14    // Set PTD1 as SPI0_SCK — ALT2
15    PORTD->PCR[1] &= ~PORT_PCR_MUX_MASK;
16    PORTD->PCR[1] |= PORT_PCR_MUX(2);
17
18    // Set PTD2 as SPI0_SOUT — ALT2
19    PORTD->PCR[2] &= ~PORT_PCR_MUX_MASK;
20    PORTD->PCR[2] |= PORT_PCR_MUX(2);
21
22    // Set PTD3 as SPI0_SIN — ALT2
23    PORTD->PCR[3] &= ~PORT_PCR_MUX_MASK;
24    PORTD->PCR[3] |= PORT_PCR_MUX(2);
25}
```

SPI - Ejemplo

```
1 // Enable master mode, inactive SS is high
2 SPI0->MCR |= SPI_MCR_MSTR_MASK | SPI_MCR_PCSIS_MASK;
3
4 // SCK baud rate = (Bus Clock /PBR) × [(1+DBR)/BR]
5 // SCK = (60MHz/7) × [1/8192] = 1 kHz approx 1.046 kHz
6 uint32_t temp;
7 temp = SPI0->CTAR[0] & ~(SPI_CTAR_DBR_MASK | SPI_CTAR_PBR_MASK | SPI_CTAR_BR_MASK);
8 SPI0->CTAR[0] = temp | SPI_CTAR_DBR(0) |
9     SPI_CTAR_PBR(3) |
10    SPI_CTAR_BR(12);
11
12 // Temp variable with zeros in changed fields
13 temp = SPI0->CTAR[0] & ~(SPI_CTAR_FMSZ_MASK | SPI_CTAR_CPOL_MASK | SPI_CTAR_CPHA_MASK |
14    SPI_CTAR_LSBFE_MASK);
15
16 // Dataframe de 16 bits, inactive SCK is low, CPHA is one where data is changed on the leading edge,
17 // MSB first
18 SPI0->CTAR[0] = temp | SPI_CTAR_FMSZ(16-1) |
19    SPI_CTAR_CPOL(1) |
20    SPI_CTAR_CPHA(1) |
21    SPI_CTAR_LSBFE(0);
22
23 // Enable SPI transfer
24 SPI0->MCR &= ~SPI_MCR_HALT_MASK;
25 }
```

SPI - Ejemplo

Envío y recepción de datos.

```
1 uint32_t SPI_send(uint16_t data){
2     uint32_t dataRx;
3
4     // Push data to Tx FIFO, and PCS = 0
5     SPI0->PUSHR = (SPI_PUSHR_PCS(1) | data);
6     // Wait for Tx buffer empty
7     while (!(SPI0->SR & SPI_SR_TCF_MASK));
8     SPI0->SR |= SPI_SR_TCF_MASK;
9
10
11    // Wait for data in Rx buffer
12    while (!(SPI0->SR & SPI_SR_RFDF_MASK));
13    //Read data
14    dataRx = SPI0->POPR;
15    SPI0->SR |= SPI_SR_RFDF_MASK;
16    return dataRx;
17 }
18
19
20
```

SPI - Ejemplo

Main.

```
1 int main(void) {  
2     float temperature = 0;  
3  
4     /* Init board hardware. */  
5     BOARD_InitBootClocks();  
6     Init_SPI0();  
7  
8     while(1) {  
9         temperature = ((float)(SPI_send(0x00)>>3))/4.0;  
10    }  
11    return 0 ;  
12 }  
13  
14
```

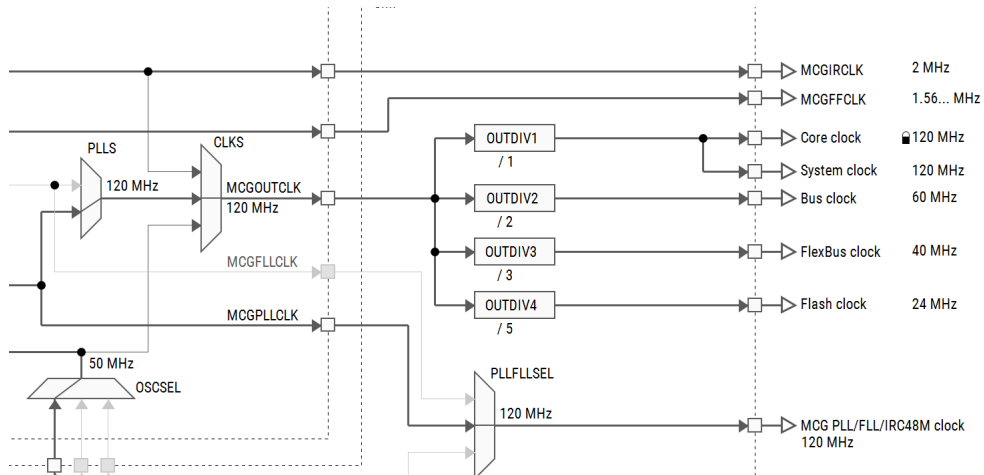

SPI - Ejemplo 2

Utilice el SDK para resolver el mismo problema.

Solución:

- Configure el reloj del sistema. Recuerde que muchos periféricos trabajan es con el clock del Bus.
- Habilite los puertos o pines a trabajar. Recuerde que esto implica activar el clock y ponerlos en la alternativa a trabajar
- Configure los periféricos que vaya a usar.
- Utilice el API de los drivers de cada periférico.

SPI - Ejemplo 2



SPI - Ejemplo 2

| | | | | | | | | | | | | |
|-----|---|---------------|-------------|--------------|-------|-------------|------------|-----------|----------|--------------|----------|-------|
| 91 | PTC17/UA... | J1[4] | TMR_1588... | J1[4] (D1) | PTC17 | UART3_TX | | | FB_CS4_b | | | VDDA |
| 92 | PTC18/UA... | J6[8]/RF_W... | WIFI_IRQ... | | PTC18 | UART3_RT... | | | FB_CS2_b | | | VREFH |
| 93 | <input checked="" type="checkbox"/> SPI0_PCS0 | J2[6] | | J2[6] (D10) | PTD0 | UART2_RT... | FTM3_CH0 | | FB_CS1_b | SPI0_PCS0 | LLWU_P12 | VREFL |
| 94 | <input checked="" type="checkbox"/> SPI0_SCK | J2[12] | | J2[12] (D13) | PTD1 | UART2_CT... | FTM3_CH1 | ADC0_SE5b | FB_CS0_b | SPI0_SCK | | VSSA |
| 95 | <input checked="" type="checkbox"/> SPI0_SOUT | J2[8] | UART2_RX | J2[8] (D11) | PTD2 | UART2_RX | FTM3_CH2 | | FB_AD4 | SPI0_SOUT | LLWU_P13 | |
| 96 | <input checked="" type="checkbox"/> SPI0_SIN | J2[10] | UART2_TX | J2[10] (D12) | PTD3 | UART2_TX | FTM3_CH3 | | FB_AD3 | SPI0_SIN | | |
| 97 | PTD4/LLW... | J6[4]/RF_W... | WIFI_CS | | PTD4 | UART0_RT... | FTM0_CH4 | | FB_AD2 | SPI0_PCS1... | LLWU_P14 | |
| 98 | ADC0_SE6... | J6[5]/RF_W... | WIFI_SCK | | PTD5 | UART0_CT... | FTM0_CH5 | ADC0_SE6b | FB_AD1 | SPI0_PCS2... | | |
| 99 | ADC0_SE7... | J6[6]/RF_W... | WIFI_MOSI | | PTD6 | UART0_RX | FTM0_CH... | ADC0_SE7b | FB_AD0 | SPI0_PCS3... | LLWU_P15 | |
| 100 | PTD7/CM... | J6[7]/RF_W... | WIFI_MISO | | PTD7 | UART0_TX | FTM0_CH... | | | SPI1_SIN | | |

[Routing Details](#)

Pins Signals 🔍 type filter text

Routing Details for... 5

[illegible]

SPI - Ejemplo 2

SPI0

Dual Serial Peripheral Interface (DSPI) [Peripheral drivers (Device specific)]

Name: SPI0 Custom name ☐

Mode: Polling Peripheral: SPI0

^ General configuration Preset: Custom...

DSPI mode: Master mode

Clock source: Bus clock - BOARD_BootClockRUN: 60 MHz, BOARD_BootClockVLPB: 4 MHz

Clock source frequency: 60 MHz (BOARD_BootClockRUN)

^ Master configuration

Set Clock and Transfer Attributes (CTAR): CTAR0 selection option

^ The ctarConfig to the desired CTAR

| | |
|-------------------------------------|--|
| Baud rate | 1000 |
| Bits per frame | 16 |
| Clock polarity | CPOL=0: SCK inactive in low |
| Clock phase | CPHA=1: Data on leading edge/capture on following edge |
| MSB or LSB data shift direction | Data transfers start with most significant bit |
| PCS to SCK delay time [ns] | 1000 |
| The last SCK to PCS delay time [ns] | 1000 |
| After the SCK delay time [ns] | 1000 |

Peripheral Chip Select (pcs): PCS0/SS * [93] J2[6]

PCS active high or low: Active Low (idles high)

Continuous SCK: ☐

Receive FIFO overflow overwrite: ☐

Modified transfer format: ☐

Sample point: Zero system clocks between SCK edge and SIN sample

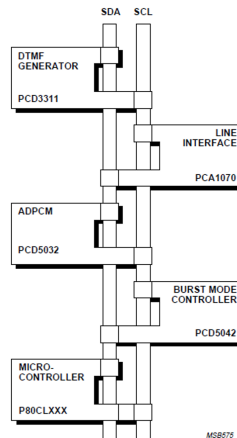
SPI - Ejemplo 2

```
1  int main(void) {
2
3      float temperature;
4
5      /* Init board hardware. */
6      BOARD_InitBootPins();
7      BOARD_InitBootClocks();
8      BOARD_InitBootPeripherals();
9      #ifndef BOARD_INIT_DEBUG_CONSOLE_PERIPHERAL
10     /* Init FSL debug console. */
11     BOARD_InitDebugConsole();
12     #endif
13
14     // Command configuration for SPI
15     dspicommand_data_config_t SPI0Config;
16     SPI0Config.isPcsContinuous=false;
17     SPI0Config.whichCtar = kDSPI_Ctar0;
18     SPI0Config.whichPcs = kDSPI_Pcs0;
19     SPI0Config.clearTransferCount = false;
20     SPI0Config.isEndOfQueue = false;
21
22     while(1) {
23         DSPI_MasterWriteDataBlocking(SPI0, \&SPI0Config, 0x00);
24         uint32_t DataSPI = DSPI_ReadData(SPI0);
25         temperature = ((float)(DataSPI>>3))/4.0;
26         PRINTF("La temperatura es: %f\n\r",temperature);
27     }
28     return 0 ;
29 }
```

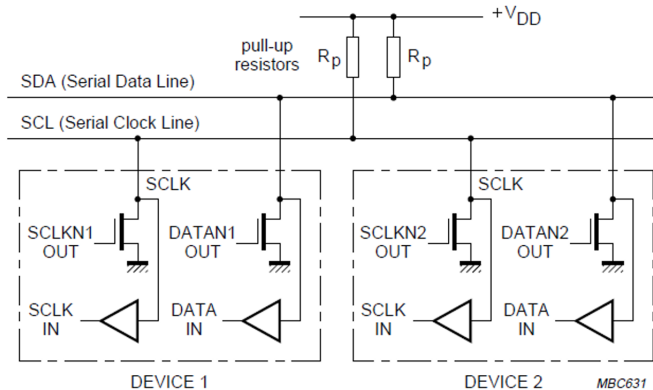
COMUNICACIONES I2C (Inter-Integrated Circuit)

I2C - Conceptos

- Múltiples dispositivos conectados al mismo bus serial.
- El bus es controlador por un dispositivo maestro, y los esclavos responden cuando se envían sus direcciones.
- El bus I2C tiene dos señales: SDA y SCL.
- Todos los detalles se encuentra en su protocolo de definición.

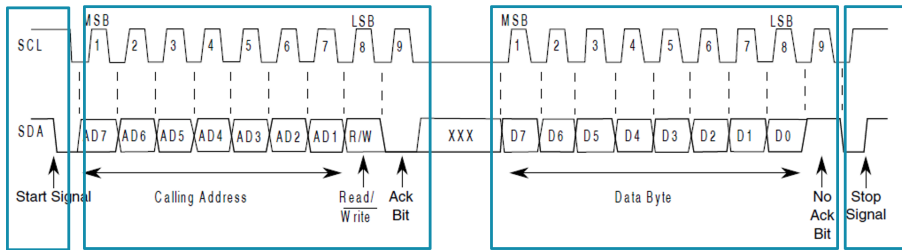


I2C - Conceptos



- Resistencia de pull-up a Vdd.
- Transistores en drenaje abierto llevan la linea a tierra
- El maestro genera la señal de clock que se distribuye a todos. Puede estar entre los rangos de 400 kHz, 1 MHz o más.

I2C - Formato de comunicación



- Transferencia de datos orientado a los mensajes con cuatro partes
 - Condición de start
 - Transmisión de la dirección del esclavo
 - Dirección
 - Escritura o Lectura
 - ACK por parte del receptor
 - Campo de datos: byte de datos y ACK
 - Condición de stop

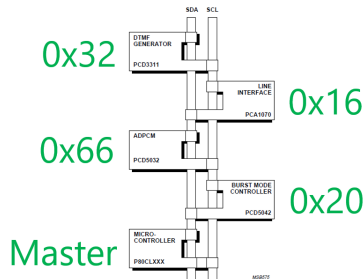
I2C - Direccionamiento

■ Direccionamiento a esclavo

- Cada esclavo tiene 7 bits de direccionamiento.
- Puede soportar hasta 128 dispositivos
- Diferentes dispositivos deben tener diferentes direcciones por defecto
- A veces se puede seleccionar una segunda dirección.

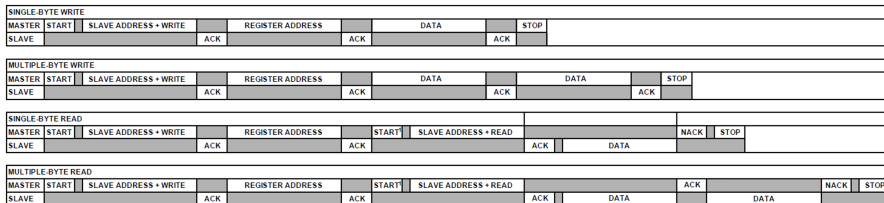
■ Direccionamiento a registro

- Algunos dispositivos tienen señales de control, registros de datos, o memoria, como se accede a esta?
- Usa el primer byte de datos como direccionamiento a registro



| Name | Type | Register Address | Comment |
|-----------------------------------|------|------------------|---|
| STATUS/F_STATUS ⁽¹⁾⁽²⁾ | R | 0x00 | FMODE = 0, real time status FMODE > 0, FIFO status |
| OUT_X_MSB ⁽¹⁾⁽²⁾ | R | 0x01 | [7:0] are 8 MSBs of 14-bit sample. Root pointer to XYZ FIFO data. |
| OUT_X_LSB ⁽¹⁾⁽²⁾ | R | 0x02 | [7:2] are 6 LSBs of 14-bit real-time sample |
| OUT_Y_MSB ⁽¹⁾⁽²⁾ | R | 0x03 | [7:0] are 8 MSBs of 14-bit real-time sample |
| OUT_Y_LSB ⁽¹⁾⁽²⁾ | R | 0x04 | [7:2] are 6 LSBs of 14-bit real-time sample |
| | | | [7:0] are 8 MSBs of 14-bit real-time sample |

I2C - Direccionamiento a registros



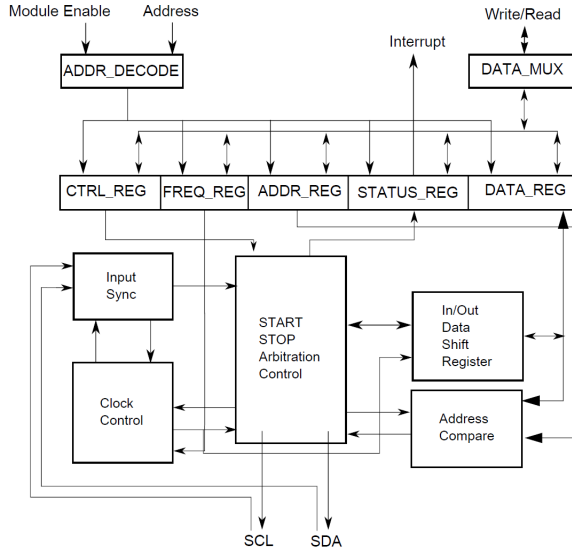
NOTES

1. THIS START IS EITHER A RESTART OR A STOP FOLLOWED BY A START.
2. THE SHADED AREAS REPRESENT WHEN THE DEVICE IS LISTENING.

■ La comunicación la dirige el maestro

- Se envía la condición de start, la dirección del esclavo, y el comando de lectura/es-
critura.
- Se espera por el ACK del esclavo.
- Se envía el direccionamiento a registro.
- Se espera el ACK del esclavo

I2C - Hardware



I2C - Configuración - I2Cx_F

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------|------|---|---|---|-----|---|---|---|
| Read | MULT | | | | ICR | | | |
| Write | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

■ I2Cx_F: registro de divisor de frecuencia

- MULT: especifica el multiplicador $mul=2^{MULT}$
- ICR: frecuencia del clock
- I2C baud rate: $f_{bus}/(2^{MULT} * ICR)$

I2C - Configuración - I2Cx_C1

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------|-------|-------|-----|----|------|------|------|-------|
| Read | IICEN | IICIE | MST | TX | TXAK | 0 | WUEN | DMAEN |
| Write | | | | | | RSTA | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

- IICEN - habilita el I2C
- IICIE - habilita la interrupción del I2C
- MST - selecciona como modo maestro
 - 0 a 1, genera la condición de start
 - 1 a 0, genera la condición de stop
- TX - selecciona 1 para transmitir desde el maestro y 0 para recibir
- TXAK - habilitar el ACK
- RSTA - repetir start
- WUEN - habilitar wakeup
- DMAEN - habilitar DMA

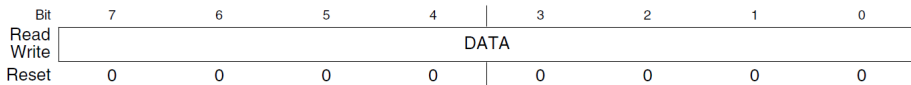
I2C - Configuración - I2Cx_S

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------|-----|------|------|------|-----|-----|-------|------|
| Read | TCF | IAAS | BUSY | ARBL | RAM | SRW | IICIF | RXAK |
| Write | | | | w1c | | | w1c | |
| Reset | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

- TCF - bandera de transferencia completa, después de transmitir byte y ACK
- IAAS - direccionamiento como esclavo
- BUSY - bus ocupado
- ARBL - se pierde arbitración
- RAM - rango de direccionamiento alcanzado
- RXAK - señal ACK recibida

I2C - Configuración - I2Cx_D

Address: Base address + 4h offset



- Registro de datos de 8 bits.
- Modo de transmisión maestro
 - Escribir a este registro inicia la transferencia de datos.
- Modo de recepción del maestro
 - Leer este registro comienza la recepción del siguiente byte

I2C - Macros útiles

```
1 #define I2C_M_START    I2C0->C1 |= I2C_C1.MST_MASK
2 #define I2C_M_STOP     I2C0->C1 &= ~I2C_C1.MST_MASK
3 #define I2C_M_RSTART   I2C0->C1 |= I2C_C1.RSTA_MASK
4
5 #define I2C_TRAN       I2C0->C1 |= I2C_C1.TX_MASK
6 #define I2C_REC        I2C0->C1 &= ~I2C_C1.TX_MASK
7
8 #define BUSY_ACK       while( I2C0->S & 0x01)
9 #define TRANS_COMP     while( !( I2C0->S & 0x80))
10 #define I2C_WAIT       while( ( I2C0->S & I2C_S.IICIF_MASK )==0){} \
11                        I2C0->S |= I2C_S.IICIF_MASK ;
12
13 #define NACK           I2C0->C1 |= I2C_C1.TXAK_MASK
14 #define ACK            I2C0->C1 &= ~I2C_C1.TXAK_MASK
15
16
```

I2C - Funciones

Mandar un byte

```
1 I2C_TRAN; /*set to transmit mode */
2 I2C_M_START; /*send start */
3 I2C0->D = dev; /*send dev address */
4 I2C_WAIT; /*wait for ack */
5
6 I2C0->D = address; /*send write address
7 */
8 I2C_WAIT;
9
10 I2C0->D = data; /*send data */
11 I2C_WAIT;
12 I2C_M_STOP;
```

Leer un byte

```
1 I2C_TRAN; /*set to transmit mode */
```

```
2 I2C_M_START; /*send start */
3 I2C0->D = dev; /*send dev address */
4 I2C_WAIT; /*wait for completion */
5 I2C0->D = address; /*send read address */
6 I2C_WAIT; /*wait for completion */
7 I2C_M_RSTART; /*repeated start */
8 I2C0->D = (dev|0x1); /*send dev address (
9 read) */
10 I2C_WAIT; /*wait for completion */
11 I2C_REC; /*set to receive mode */
12 NACK; /*set NACK after read */
13 data = I2C0->D; /*dummy read */
14 I2C_WAIT; /*wait for completion */
15 I2C_M_STOP; /*send stop */
16 data = I2C0->D; /*read data */
17
```

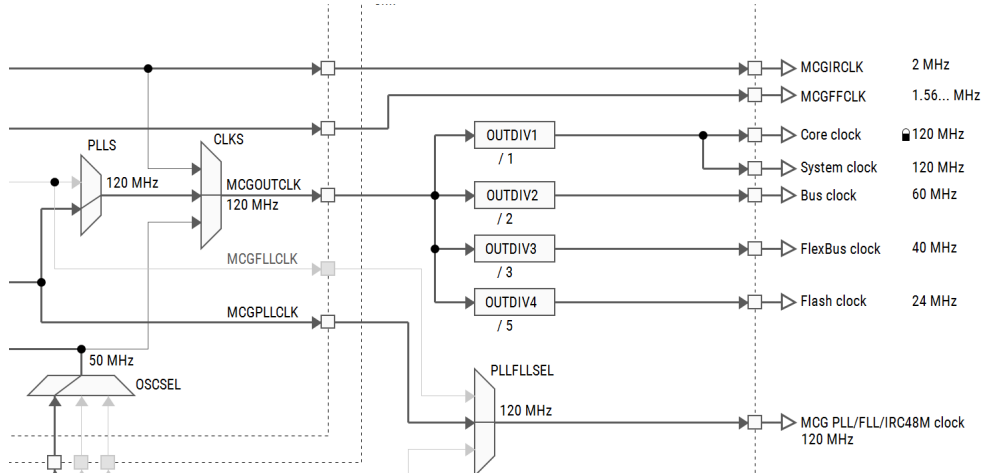
I2C - Ejemplo

Se requiere tomar la temperatura y la humedad utilizando el sensor SHT3X cada segundo.

Solución:

- Configurar Clocks del sistema
- Configurar pines a utilizar
- Configurar los periféricos tanto del I2C como del PIT
- Configurar interrupción
- Hacer funciones para generar comandos al sensor.

I2C - Ejemplo - Solución





I2C - Ejemplo - Solución

The screenshot displays a PCB design tool interface. The top section shows a component placement grid with various components like VREF_OUT, DAC0_OU, XTAL32, EXTAL32, VBAT, I2C0_SCL, I2C0_SDA, PTE26/EN, PTE26, UART4_TX, UART4_RX, UART4_CT, PTA0, PTA1, PTA2, PTA3, PTA4, PTA5, FTM0_CH5, FTM0_CH6, FTM0_CH7, FTM0_CH0, FTM0_CH1, FTM0_CH2, I2S0_TX_B, LLWU_P3, CMP2_OUT, RMII0_RX, VDD, and VREF_OUT. The right side shows a component footprint for MK64FN1M with pins VREF_OU, DAC0_OU, XTAL32, EXTAL32, VBAT, I2C0_SCL, I2C0_SDA, PTE26, and PTA0. The bottom section shows the Routing Details table.

| # | Peripheral | Signal | Arrow | Routed pin/signal | Label | Identifier | Direction | GPIO initial state | GPIO interrupt | Slew rate | Open drain | Drive strength | Pull select | Pull enable | Passive filter | Digital filter |
|----|------------|--------|-------|-------------------|------------------------|---------------|--------------|--------------------|----------------|-----------|------------|----------------|-------------|-------------|----------------|----------------|
| 36 | TPIU | SWO | -> | [36] TRACE_SWO | J1[12]/J9[6]/TRACE_SWO | n/a | Output | n/a | n/a | Fast | Disabled | Low | Pulldown | Disabled | Disabled | n/a |
| 31 | I2C0 | SCL | <-> | [31] I2C0_SCL | J2[20]/U8[4]/I2C0_SCL | Not Specified | Input/Output | n/a | n/a | Fast | Disabled | Low | Pulldown | Disabled | Disabled | n/a |
| 32 | I2C0 | SDA | <-> | [32] I2C0_SDA | J2[18]/U8[6]/I2C0_SDA | Not Specified | Input/Output | n/a | n/a | Fast | Disabled | Low | Pulldown | Disabled | Disabled | n/a |

I2C - Ejemplo - Solución

Inter-Integrated Circuit (I2C) *(Peripheral drivers (Device specific))*  

Name: I2C0 Custom name ☐

Mode: Polling Peripheral: I2C0

General configuration Preset: Custom...

I2C mode: Master mode

Clock source: Bus clock - BOARD_BootClockRUN: 60 MHz, BOARD_BootClockVLP: 4 MHz

Clock source frequency: 60 MHz (BOARD_BootClockRUN)

Master configuration

Enable the peripheral at initialization time ☒

Controls the stop hold ☐

Baud rate: 100000

Controls the width of the glitch: 0

I2C - Ejemplo - Solución

Periodic Interrupt Timer (PIT) *[Peripheral drivers (Device specific)]*

Name: PIT Custom name: ☐

Mode: General Peripheral: PIT

^ **General PIT configuration** Preset: Custom...

Run PIT in debug mode: ☐

^ **Clock setting**

Clock source: Bus clock - BOARD_BootClockRUN: 60 MHz, BOARD_BootClockVLP: 4 MHz

Clock source frequency: 60 MHz (BOARD_BootClockRUN)

^ **PIT channels** + x

| CHANNEL_0 | |
|--------------------------------|-------------------------------------|
| Channel ID | CHANNEL_0 |
| Channel number | Channel 0 |
| Chained | <input type="checkbox"/> |
| Channel period/frequency | 1s |
| Resulting period | 1 s (60000000 ticks) |
| Start channel | <input checked="" type="checkbox"/> |
| Interrupt | <input checked="" type="checkbox"/> |
| Interrupt | PIT0_IRQn |
| Interrupt request | Enabled in initialization |
| Enable priority initialization | <input type="checkbox"/> |
| Priority | 0 |
| Enable custom handler name | <input type="checkbox"/> |
| Interrupt handler name | PIT_CHANNEL_0_IRQHANDLER |
| Handler template | Copy to clipboard |

I2C - Ejemplo - Solución

Creación de algunas MACROS

```
1 #define SHT31_DEFAULT_ADDR      0x44U
2 #define SHT31_MEAS_HIGHREP      0x2400U
3 #define SHT31_MEAS_MEDREP       0x240BU
4 #define SHT31_MEAS_LOWREP       0x2416
5 #define SHT31_READSTATUS        0xF32D
6 #define SHT31_SOFTRESET         0x30A2U
7 #define I2C_Wait()               while ((I2C_MasterGetStatusFlags(I2C0) & I2C_S_IICIF_MASK)==0) {} \
8                                 I2C0->S |= I2C_S_IICIF_MASK;
9
```

Creación de función para envío de comando.

```
1 void writeCommand(uint16_t cmd, uint8_t address) {
2     uint8_t buf[2];
3     buf[0] = (cmd >> 8);
4     buf[1] = (cmd & 0xFF);
5
6     I2C_MasterStart(I2C0, address, kI2C_Write);           // Start transmission with Slave, write mode
7     I2C_Wait();                                           // Wait for ACK
8     I2C_MasterWriteBlocking(I2C0, buf, 2, kI2C_TransferDefaultFlag); // Send command
9 }
10
```

Creación de función para soft reset.

```
1 void reset(void) {
2     writeCommand(SHT31_SOFTRESET, SHT31_DEFAULT_ADDR);
3     delaySimple(10000);
4 }
5
```


I2C - Ejemplo - Solución

Lectura de status

```
1 uint16_t readStatus(uint8_t address) {
2     //Send command read status
3     writeCommand(SHT31.READSTATUS,SHT31.DEFAULT_ADDR);
4
5     // Read status
6     uint8_t readbuffer[3];
7     I2C_MasterStart(I2C0, address, kI2C_Read); // Start transmission with Slave, read mode
8     I2C_Wait(); // Wait for ACK
9     I2C_MasterReadBlocking(I2C0, &readbuffer[0], 3, kI2C_TransferDefaultFlag); // read 3 data
10
11     uint16_t stat = readbuffer[0];
12     stat <<= 8;
13     stat |= readbuffer[1];
14
15     if (readbuffer[2] != crc8((uint8_t *) readbuffer, 2)) {
16         PRINTF("CRC erroneo \r\n", stat);
17         return false;
18     }
19
20     PRINTF("%d\r\n", stat);
21     return stat;
22 }
23
```

I2C - Ejemplo - Solución

Inicialización del sensor y CRC8

```
1 void Init_SHT3(void){
2     reset();
3     readStatus(SHT31-DEFAULT_ADDR);
4 }
5
6 uint8_t crc8(const uint8_t *data, int len) {
7     uint8_t crc = 0xFF;
8     size_t i, j;
9     for (i = 0; i < len; i++) {
10         crc ^= data[i];
11         for (j = 0; j < 8; j++) {
12             if ((crc & 0x80) != 0)
13                 crc = (uint8_t)((crc << 1) ^ 0x31);
14             else
15                 crc <<= 1;
16         }
17     }
18     return crc;
19 }
20
21
```

I2C - Ejemplo - Solución

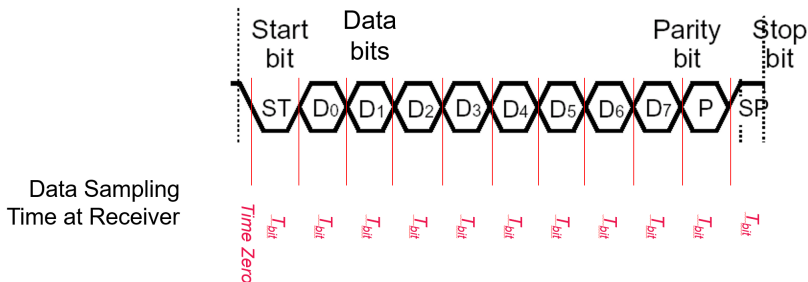
Lectura de temperatura y humedad

```
1  _Bool readTempHum(void) {
2      uint8_t readbuffer[6];
3
4      writeCommand(SHT31_MEAS_HIGHREP,
5                   SHT31_DEFAULT_ADDR);
6
7      delaySimple(500000);
8
9      I2C_MasterStart(I2C0, SHT31_DEFAULT_ADDR,
10                      kI2C_Read);
11
12      I2C_Wait(); // Wait for ACK
13
14      I2C_MasterReadBlocking(I2C0, &readbuffer[0], 6,
15                             kI2C_TransferDefaultFlag);
16
17      uint16_t ST, SRH;
18      ST = readbuffer[0];
19      ST <<= 8;
20      ST |= readbuffer[1];
21
22      if (readbuffer[2] != crc8((uint8_t *) readbuffer
23                               , 2)) {
24          return false;
25      }
26
27      SRH = readbuffer[3];
28      SRH <<= 8;
29      SRH |= readbuffer[4];
30
31      if (readbuffer[5] != crc8((uint8_t *) readbuffer
32                               +3, 2)) {
33          return false;
34      }
35
36      double stemp = ST;
37      stemp *= 175;
38      stemp /= 0xffff;
39      stemp = -45 + stemp;
40
41      double shum = SRH;
42      shum *= 100;
43      shum /= 0xFFFF;
44
45      PRINTF("La temperatura fue de: %f y la humedad
46             de %f\n\r", stemp, shum);
47      return true;
48 }
```

SERIAL ASINCRÓNICO UART

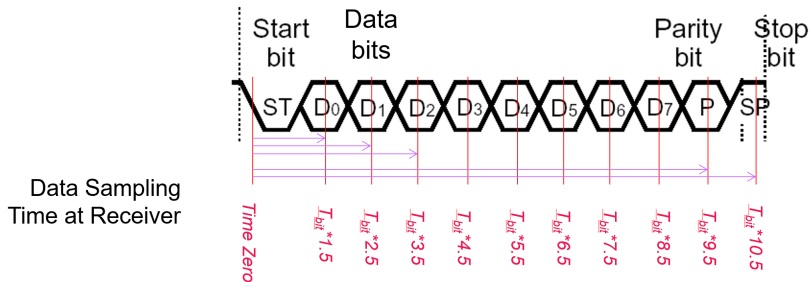
(Universal Asynchronous Receiver-Transmitter)

UART - Conceptos básicos transmisión



- Si no hay datos para enviar, la línea se mantiene en uno.
- Cuando se necesita enviar un datos, entonces:
 - Se envia un cero, o start bit; para indicar el inicio de una transmisión.
 - Enviar cada bit en la palabra (utiliza registros de desplazamiento).
 - Envía un 1 para indicar el paro de la transmisión.

UART - Conceptos básicos recepción

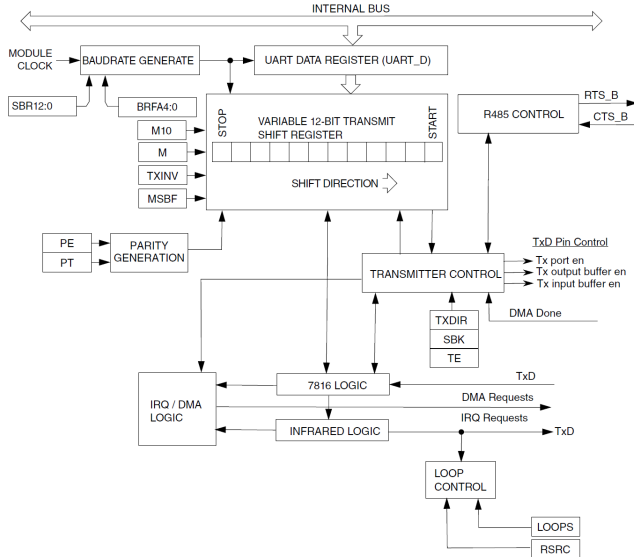


- Se espera por un flanco de bajada.
 - Se espera medio bit para tomar el dato
 - Hacer lo siguiente de acuerdo a la cantidad de bits en la palabra
 - Esperar un tiempo de un bit
 - Leer el bit y desplazarlo al buffer de entrada
 - Esperar un bit
 - Leer el bit: si es uno, entonces se acaba la transmisión. Si es cero, hay problemas.

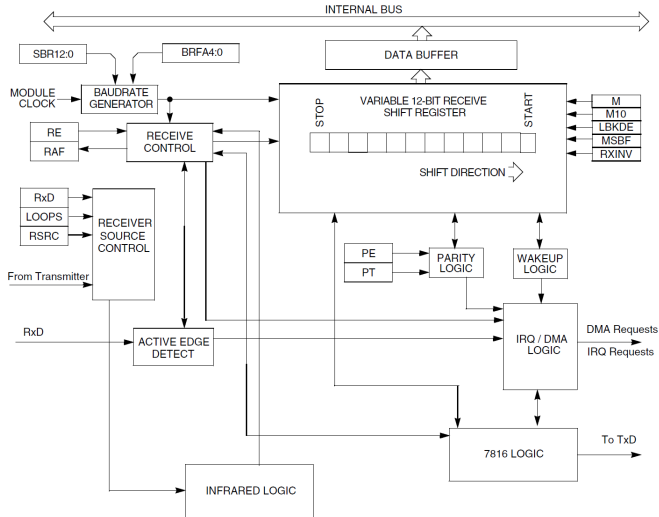
UART - Que se debe cumplir para que funcione?

- El transmisor y el receptor deben coincidir en las mismas configuraciones:
 - El orden de los bit de datos.
 - Numero de bits de datos a transmitir
 - Como es el bit de start y el bit de stop
 - Cuanto dura un bit
 - El transmisor y receptor deben tener clock muy parecidos ya que la única referencia es cuando se da el flanco en la bit de start
 - En el K64 de 100 pines se tienen 5 UART

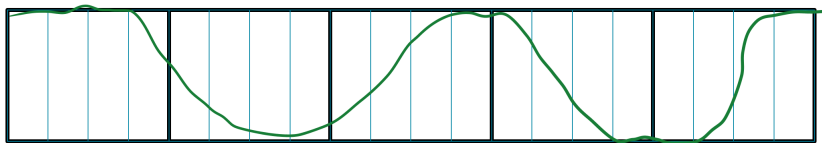
UART - Transmisor



UART - Receptor

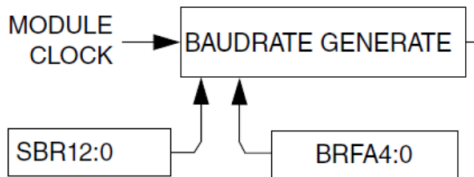


UART - Sobremuestreo



- Cuando se recibe, el UART sobremuestra la línea de datos
 - Mas muestras permite botar y mejorar la inmunidad al ruido
 - Mejor sincronización para la línea de datos
- En el K64, el receptor muestrea la línea de acuerdo al RT clock, este RT clock es 16 veces el baud rate de la comunicación.

UART - Generador de clock



$$\text{UART baud rate} = \text{UART module clock} / (16 \times (\text{SBR}[12:0] + \text{BRFD}))$$

- UART0 y UART 1 utilizan el clock del sistema (120 MHz), y UART2-4 el clock del bus (60 MHz)
- Se requiere dividir la frecuencia del Bus para lograr el baud rate deseado
- Ejemplo: se usa el UART2 con frecuencia de bus de 60MHz. Para lograr 9600 baud rate entonces

$$SBR = 60E6 / (9600 * 16) = 390$$

UART - Registro UARTx_C1

Address: Base address + 2h offset

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------|-------|----------|------|---|------|-----|----|----|
| Read | LOOPS | UARTSWAI | RSRC | M | WAKE | ILT | PE | PT |
| Write | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

- LOOPS: habilita un loop interno entre Tx y Rx. 0 Operación normal
- UARTSAWI: UART para en modo espera
- RSRC: seleccionar la fuente en loop
- M: Seleccionar datos de 9bits
- WAKE: método de despertar
- ILT: tipo de idle
- PE: paridad habilitada con 1
- PT: paridad impar con 1 y par con 0

UART - Registro UARTx_C2

Address: Base address + 3h offset

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------|-----|------|-----|------|----|----|-----|-----|
| Read | TIE | TCIE | RIE | ILIE | TE | RE | RWU | SBK |
| Write | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

- Habilitar las interrupciones
 - TIE: Interrupción cuando Tx está vacío.
 - TCIE: interrupción cuando Tx está completo.
 - RIE: interrupción cuando Rx tiene dato
- Habilitar el módulo
 - TE: Habilitar Tx
 - RE: Habilitar Rx
- Otros
 - RWU: poner el receptor en modo espera, se despierta cuando ocurra el evento de despertar
 - SBK: mandar un caracter de break (todos ceros)

UART - Registro UARTx_S1

Address: Base address + 4h offset

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------|------|----|------|------|----|----|----|----|
| Read | TDRE | TC | RDRF | IDLE | OR | NF | FE | PF |
| Write | | | | | | | | |
| Reset | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

- TDRE: el registro de Tx está vacío, puede escribir mas datos
- TC: Tx completada
- RDRF: Receptor lleno en el registro, puede recibir mas datos
- IDLE: UART se pone en idle durante un caracter
- OR: Se ha borrado el dato de recepción por uno nuevo
- NF: Bandera de ruido
- FE: Recibido un cero para parada y esperaba un uno
- PF: Error de paridad

UART - Registro UARTx_S2

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------|--------|---------|------|-------|-------|-------|-------|-----|
| Read | LBKDIF | RXEDGIF | MSBF | RXINV | RWUID | BRK13 | LBKDE | RAF |
| Write | w1c | w1c | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

- LBDKIF: detecta la bandera de interrupción
- RXEDGIF: detecta flanco activo en el pin de recepción.
- MSBF: mandar el bit más significativo primero.
- RXINV: invertir la polaridad de las señales de recepción.
- RWUID: activa bit de recepción para despertar el MCU.
- BRK13: envía carácter de break de acuerdo a la longitud.
- LBKDE: habilitar la linea break
- RAF: no hay linea idle

UART - Polling Serial Comm

```
1 void Init_UART2(uint32_t baud_rate) {
2     uint32_t divisor;
3     // enable clock to UART and Port A
4     SIM->SCGC4 |= SIM_SCGC4_UART2_MASK;
5     SIM->SCGC5 |= SIM_SCGC5_PORTE_MASK;
6
7     // connect UART to pins for PTE22, PTE23
8     PORTE->PCR[22] = PORT_PCR_MUX(4);
9     PORTE->PCR[23] = PORT_PCR_MUX(4);
10    // ensure tx and rx are disabled before configuration
11    UART2->C2 &= ~(UARTLP_C2_TE_MASK | UARTLP_C2_RE_MASK);
12
13    // Set baud rate
14    divisor = BUS_CLOCK/(baud_rate*16);
15    UART2->BDH = UART_BDH_SBR(divisor>>8);
16    UART2->BDL = UART_BDL_SBR(divisor);
17
18    // No parity, 8 bits, two stop bits, other settings;
19    UART2->C1 = UART2->S2 = UART2->C3 = 0;
20
21    // Enable transmitter and receiver
22    UART2->C2 = UART_C2_TE_MASK | UART_C2_RE_MASK;
23 }
24
```


UART - Transmisión serial

```
1 void UART2_Transmit_Poll(uint8_t data) {  
2     // wait until transmit data register is empty  
3     while (!(UART2->S1 & UART_S1_TDRE_MASK));  
4     UART2->D = data;  
5 }  
6 void main(void) {  
7     char c;  
8     // Initialization goes here  
9     while (1) {  
10         for (c='a'; c<='z'; c++) {  
11             UART2_Transmit_Poll(c);  
12         }  
13     }  
14 }  
15
```

UART - Recepción serial

```
1 uint8_t UART2_Receive_Poll(void) {
2     // wait until receive data register is full
3     while (!(UART2->S1 & UART_S1.RDRF_MASK));
4     return UART2->D;
5 }
6 void main(void) {
7     char c;
8     // Initialization goes here
9     while (1) {
10         c = UART2_Receive_Poll();
11         UART2_Transmit_Poll(c);
12     }
13 }
14
```

UART - Inicialización e Interrupciones

Usar interrupciones e inicializar periféricos para el MCU. El ISR debe identificar porque se genera la interrupción.

```
1 void Init_UART2(uint32_t baud_rate) { 18
2     NVIC_SetPriority(UART2_IRQn, 2); 19
3     NVIC_ClearPendingIRQ(UART2_IRQn); 20
4     NVIC_EnableIRQ(UART2_IRQn); 21
5 22
6     UART2->C2 |= UART_C2_TIE_MASK | 23
7     UART_C2_RIE_MASK; 24
8     UART2->C2 |= UART_C2_RIE_MASK; 25
9     Q_Init(&TxQ); 26
10    Q_Init(&RxQ); 27
11 } 28
12 void UART2_IRQHandler(void) { 29
13     NVIC_ClearPendingIRQ(UART2_IRQn); 30
14     if (UART2->S1 & UART_S1_TDRE_MASK) { 31
15         // can send another character 32
16         if (!Q_Empty(&TxQ)) { 33
17             UART2->D = Q_Dequeue(&TxQ); 34
18         } else { 35
19             // queue is empty so disable tx
20             UART2->C2 &= ~UART_C2_TIE_MASK;
21         }
22     }
23     if (UART2->S1 & UART_S1_RDRF_MASK) {
24         // received a character
25         if (!Q_Full(&RxQ)) {
26             Q_Enqueue(&RxQ, UART2->D);
27         } else {
28             // error - queue full.
29             while (1);
30         }
31     }
32 }
33 }
```

COMUNICACIONES SERIALES

GRACIAS