Cika Electrónica

Nota de Aplicación: CAN-019
Título: **Aplicaciones de comunicaciones con interfaz serie asincrónica en Dynamic C 8**
Autor: Sergio R. Caprile, Senior Engineer

Revisiones	Fecha	Comentarios
0	15/12/03	

Analizamos las bibliotecas de funciones con soporte para comunicaciones via interfaz serie asincrónica que incluye Dynamic C, y la forma de aprovechar la mayor cantidad posible de este código para acortar nuestros tiempos de desarrollo.

No vamos a detallar cada una de las funciones que cada biblioteca incluye, pero sí haremos una enumeración de las más importantes, para luego ver unos sencillos ejemplos de su uso. Consulte el manual de referencia de Dynamic C para una descripción detallada de cada función en particular.

Bibliotecas de funciones

RS232.LIB

Está orientada a las comunicaciones full duplex, proveyendo un conjunto de funciones que envían y reciben bloques de datos de forma buffereada. Tanto transmisión como recepción incorporan buffers circulares, ubicados en el área root, de tamaño definido por el usuario. Para mayor performance, el tamaño debe ser 2^n-1.

Las funciones siguen la convención "serXfn" para su nomenclatura, donde 'X' es el port utilizado y 'fn' la función implementada.

La mayoría de las funciones no permiten la ejecución simultánea de otras tareas, retornando cuando terminan su cometido. Las funciones de recepción retornan al cumplir su misión o luego de transcurrido un tiempo predeterminado, retornando inmediatamente si no hay caracteres esperando en el buffer de recepción:

➔ *serXgetc()*: devuelve un caracter del buffer o retorna si éste está vacío
➔ *serXread()*: obtiene un número de bytes del buffer, retorna por timeout o buffer vacío.

Debido a la presencia del buffer circular, para el caso de transmisión, las funciones retornarán cuando hayan logrado colocar en el buffer todo el mensaje que se les pasa. Si se toma la precaución de definir el buffer de un tamaño mayor al requerido para transmitir el mensaje más largo, las funciones no bloquearán la ejecución de las demás tareas, para una frecuencia de mensajes compatible con la velocidad de la interfaz:

➔ *serXputc()*: escribe un caracter en el buffer
➔ *serXputs()*: escribe un string en el buffer
➔ *serXwrite()*: escribe un número determinado de bytes en el buffer

Existe además un set de cofunciones para un solo usuario que permiten la realización de otras tareas aún cuando el buffer circular esté lleno (o vacío en el caso de recepción) mientras cumplen su función. Éstas deben ser llamadas dentro de co-sentencias (costates) y son algo más difíciles de utilizar. Retornan cuando su tarea ha terminado, por lo que funcionan perfectamente dentro de un *waitfordone{}*.

Las funciones de recepción retornan al cumplir su misión o luego de transcurrido un tiempo predeterminado, quedando en espera y permitiendo la ejecución de los demás costates (yielding) mientras tanto:

➔ *cof_serXgetc()*: recibe un caracter
➔ *cof_serXgets()*: recibe un string (delimitado por null/cr/lf o tamaño máximo).
➔ *cof_serXread()*: recibe un número prefijado de bytes

Las funciones de transmisión retornan cuando logran colocar en el buffer todo el mensaje que se les pasa, permitiendo la ejecución de los demás costates (yielding) mientras tanto:

➔ *cof_serXputc()*: coloca un caracter en el buffer
➔ *cof_serXputs()*: coloca un string en el buffer
➔ *cof_serXwrite()*: coloca un número de bytes en el buffer

Para configurar la interfaz serie, disponemos de un conjunto de funciones adicionales:

- ➔ *serXopen()*: “abre” el port serie, configura velocidad de operación
- ➔ *serXclose()*: “cierra” el port serie
- ➔ *serXdatabits()*: configura largo de palabra
- ➔ *serXparity()*: configura paridad
- ➔ *serXgetError()*: reporta errores
- ➔ *serXflowcontrolOn()*: habilita control de flujo por hardware en los buffers (RTS/CTS)
- ➔ *serXflowcontrolOff()*: inhabilita control de flujo

Finalmente, encontramos las funciones que operan sobre los buffers:

- ➔ *serXrdFlush()*: borra el buffer de recepción
- ➔ *serXwrFlush()*: borra el buffer de transmisión
- ➔ *serXrdFree()*: retorna el espacio no utilizado del buffer de recepción
- ➔ *serXwrFree()*:retorna el espacio no utilizado del buffer de transmisión
- ➔ *serXrdUsed()*: retorna el espacio utilizado del buffer de recepción

PACKET.LIB

Está orientada a las comunicaciones half duplex, proveyendo un conjunto de funciones que envían y reciben bloques de datos. Para recepción se utiliza un conjunto de buffers, cada buffer puede alojar un paquete de información. Los buffers se ubican sobre *xmem*.

Dado el carácter de half duplex, las funciones de transmisión retornan a recepción una vez finalizado el envío del último byte, más un posible tiempo de guarda definible por el usuario. Ésto es configurable, pudiendo elegirse otros modos de operación como detección de bit de address o caracter de sincronismo. Al iniciarse la operación de recepción, se llama a una función provista por el usuario, y al iniciarse la operación de transmisión se llama a otra función complementaria. Estas llamadas a funciones tienen el objeto de poder configurar el medio para la función a ejecutar, como por ejemplo actuar sobre los transceivers en RS-485. Estas funciones son:

- ➔ *pktXinit()*: llamada al inicio para configurar el hardware
- ➔ *pktXtx()*: configurar para transmitir
- ➔ *pktXrx()*: configurar para recibir

Las funciones siguen la convención “pktXfn” para su nomenclatura, donde 'X' es el port utilizado y 'fn' la función implementada.

La mayoría de las funciones no permiten la ejecución simultánea de otras tareas, sin embargo el funcionamiento es diferente al observado en *RS232.LIB*. La función de recepción retorna inmediatamente, el valor retornado indica si había algún paquete en el buffer de recepción o no. La función de transmisión también retorna inmediatamente, indicando mediante el valor retornado si pudo iniciar la operación de transmisión o no. Se provee además una función adicional para chequear el estado de la transmisión:

- ➔ *pktXsend()*:
- ➔ *pktXreceive()*:
- ➔ *pktXsending()*:

Existe además un set de cofunciones para un solo usuario que permiten la realización de otras tareas mientras se espera que terminen las operaciones de comunicaciones. Éstas deben ser llamadas dentro de co-sentencias (costates) y son algo más difíciles de utilizar. Retornan cuando su tarea ha terminado, por lo que funcionan perfectamente dentro de un *waitfordone{}*.

La función de recepción retorna al obtener un paquete, quedando en espera y permitiendo la ejecución de los demás costates (yielding) mientras tanto. La función de transmisión retorna cuando la totalidad del paquete se ha transmitido, permitiendo la ejecución de los demás costates (yielding) mientras tanto:

- ➔ *cof_pktXsend()*:
- ➔ *cof_pktXreceive()*:

Para configurar la interfaz serie, disponemos de un conjunto de funciones adicionales:

- ➔ *pktXopen()*: inicializa velocidad de operación
- ➔ *pktXclose()*: libera recursos
- ➔ *pktXinitBuffers()*: reserva memoria extendida para los buffers de recepción

➔ *pktXsetParity()*: configura paridad
➔ *pktXgetError()*: reporta errores

Ejemplos

Desarrollamos un par de sencillos ejemplos de cómo utilizar las cofunciones en un entorno multitarea. Simulamos una tarea cualquiera y las operaciones de comunicaciones, operando a cada momento sobre los LEDs de un kit RCM2100 para observar, de forma didáctica, la operación

RS232.LIB

Enviamos constantemente un “fox” por la interfaz serie, haciendo parpadear un LED cada vez que la cofunción termina de ejecutarse (cuando logra poner todo el paquete en el buffer de salida). Mientras tanto, una tarea ficticia hace parpadear otro LED, demostrando la operación multitarea. Los caracteres recibidos son buffereados hasta llenar el buffer de 32 bytes e impresos en la consola de Dynamic C. Si pasan más de 10 segundos sin recibir un caracter (una vez que ya se ha recibido uno), se indica en consola. Si pasan más de 30 segundos desde el inicio del programa y no se ha recibido ningún caracter, se muestra el texto “Inactivo” en consola. Elegimos intencionalmente una velocidad baja, a modo de poder observar la interrelación de los buffers y los caracteres transmitidos.

```
#class auto
// Define tamaño de buffers (2^n-1)
#define BINBUFSIZE  15
#define BOUTBUFSIZE 15
// Timeout de función (10 segs desde el último caracter recibido)
#define MSSG_TMOUT 10000UL
// Timeout de programa (30 segundos sin recibir nada)
#define IDLE_TMOUT 30000UL

void main()
{
unsigned long t;
int n,i;
unsigned char indata[32];
static const char Fox[]="The quick brown fox jumps over the lazy dog 01234567890\n";

        WrPortI ( SPCR, &SPCRShadow, 0x84 );         // PA0-7 = Outputs
        n = 0;
        serBopen(300);
        for (;;) {
            costate {                                // Recepción
                t = MS_TIMER;
                waitfordone {
                  n = cof_serBread(indata, sizeof(indata), MSSG_TMOUT);
                }
                if(n!=sizeof(indata))
                    printf ("\nTimeout, %d characteres recibidos: %s\n", n,indata);
                else printf ("Rx: %s\n",indata);
            }
            costate {                                // Transmisión
                waitfordone {
                  cof_serBwrite(Fox, sizeof(Fox));
                }
                BitWrPortI( PADR, &PADRShadow, (PADRShadow&2)^2, 1 ); // toggle LED 3
            }
            costate {                                // Rx timeout
                if (MS_TIMER > t + IDLE_TMOUT){
                        t = MS_TIMER;
                        printf("\nInactivo\n");
                }
            }
            costate {                                // Tarea ficticia
                waitfor(DelayMs(250));
```

```
                BitWrPortI( PADR, &PADRShadow, (PADRShadow&1)^1, 0); // toggle LED 2
            }
        }
        while (serBwrFree() != BOUTBUFSIZE);         // espera vaciar buffer de Tx
        serBclose();                                 // cierra port
}
```

PACKET.LIB

Enviamos cada dos segundos un paquete cualquiera, haciendo parpadear un LED cada vez que la cofunción termina de ejecutarse (cuando logra enviar todo el paquete). Mientras tanto, una tarea ficticia hace parpadear otro LED, demostrando la operación multitarea.

Mediante otro juego de LEDs (el RCM2100, tiene cuatro LEDs) podemos visualizar el momento en que finaliza la transmisión del último bit del último byte del mensaje, cuando la biblioteca de funciones llama a nuestra función para poner los transceivers en modo recepción.

```
#class auto
#use packet.lib

void main()
{
char rx_packet[51];
char tx_packet[24];
int packet_size;
char errors;
long t;

        pktCinitBuffers(5, 50);
        pktCopen(2400, PKT_GAPMODE, 3, NULL);
        WrPortI ( SPCR, &SPCRShadow, 0x84 );         // PA0-7 = Outputs

        for (;;) {
            costate {                                // Tarea ficticia
                waitfor(DelayMs(250));
                BitWrPortI( PADR, &PADRShadow, (PADRShadow&1)^1, 0); // toggle LED 2
            }
            costate {                                // Transmisión
                waitfor(DelaySec(2));
                waitfordone {
                        cof_pktCsend(tx_packet, sizeof(tx_packet), 3);
                }
                BitWrPortI( PADR, &PADRShadow, (PADRShadow&2)^2, 1 ); // toggle LED 3
            }
        }

}

// inicialización de transceivers
void pktCinit()
{
}
// configura transceivers para recepción
void pktCrx()
{
         BitWrPortI( PADR, &PADRShadow, 1, 2 ); // set LED 4
         BitWrPortI( PADR, &PADRShadow, 0, 3 ); // reset LED 5
}
// configura transceivers para transmisión
void pktCtx()
{
         BitWrPortI( PADR, &PADRShadow, 1, 3 ); // set LED 5
         BitWrPortI( PADR, &PADRShadow, 0, 2 ); // reset LED 4
}
```