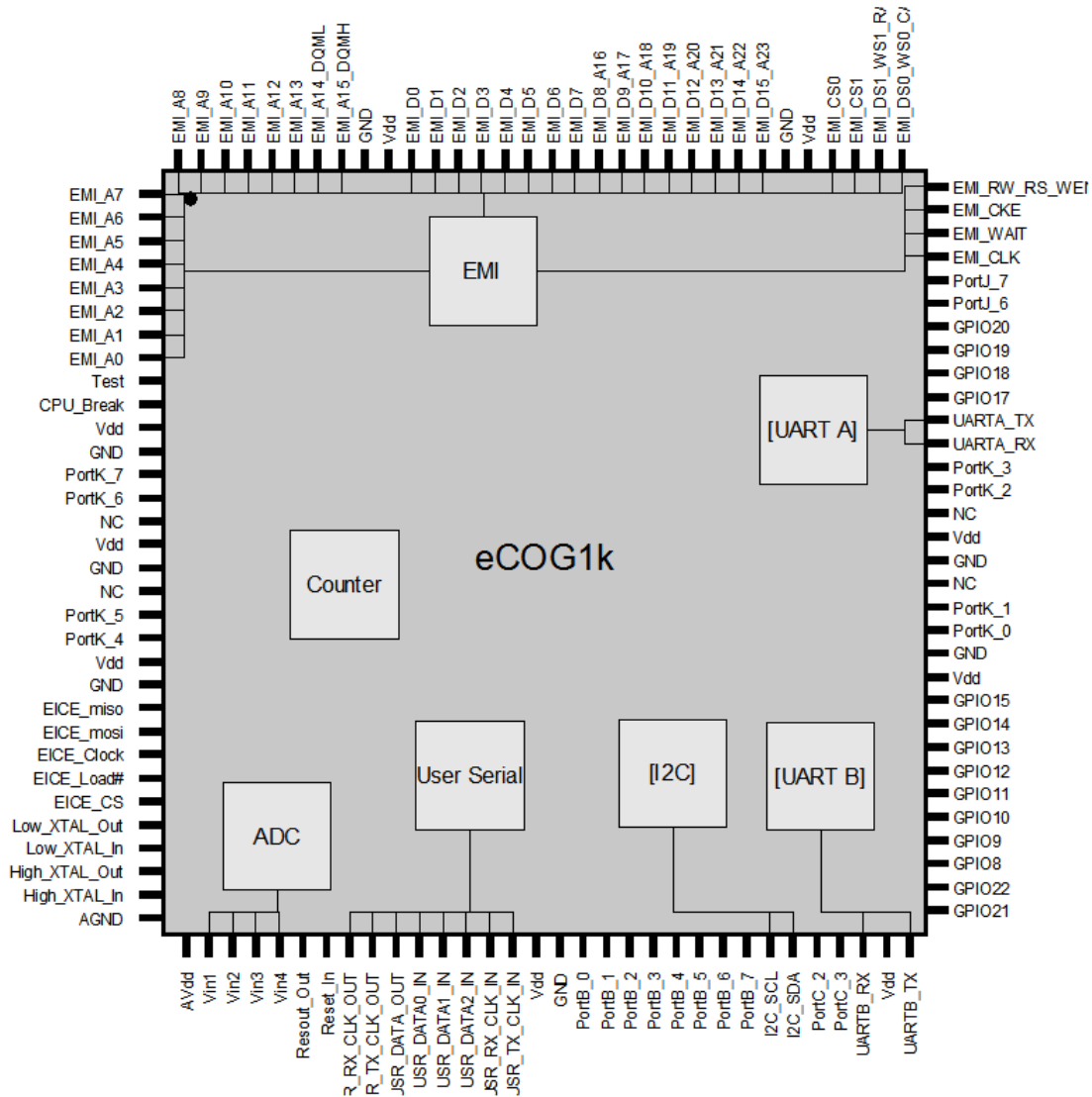




AN021 – Common Bus Serial Communications

Version 1.0

This application note describes how the Cyan Technology eCOG1k can be used to communicate on a common serial bus and control the transmitter on that bus.



Confidential and Proprietary Information

©Cyan Technology Ltd, 2008

This document contains confidential and proprietary information of Cyan Technology Ltd and is protected by copyright laws. Its receipt or possession does not convey any rights to reproduce, manufacture, use or sell anything based on information contained within this document.

Cyan Technology™, the Cyan Technology logo and Max-eICE™ are trademarks of Cyan Holdings Ltd. CyanIDE® and eCOG® are registered trademarks of Cyan Holdings Ltd. Cyan Technology Ltd recognises other brand and product names as trademarks or registered trademarks of their respective holders.

Any product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by Cyan Technology Ltd in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. Cyan Technology Ltd shall not be liable for any loss or damage arising from the use of any information in this guide, any error or omission in such information, or any incorrect use of the product.

This product is not designed or intended to be used for on-line control of aircraft, aircraft navigation or communications systems or in air traffic control applications or in the design, construction, operation or maintenance of any nuclear facility, or for any medical use related to either life support equipment or any other life-critical application. Cyan Technology Ltd specifically disclaims any express or implied warranty of fitness for any or all of such uses. Ask your sales representative for details.



Revision History

| Version | Date | Notes |
|---------|------------|-----------------|
| V1.0 | 18/01/2005 | Initial Release |
| | | |
| | | |
| | | |

Contents

| | | |
|------------|--------------------------|----|
| 1 | Introduction..... | 5 |
| 2 | Glossary | 5 |
| 3 | Overview..... | 6 |
| 4 | Data Buffers..... | 6 |
| 5 | Transmitter Control..... | 7 |
| 5.1 | Error Checking..... | 7 |
| Appendix A | Software Listing | 8 |
| A.1 | RS485.c..... | 8 |
| A.2 | irq.asm..... | 15 |

1 Introduction

Several serial communications standards in multi-drop applications use the same bus for both transmitting and receiving data. In such systems, each device on the bus has a transmitter and receiver which are connected to the same physical wiring. The transmitter has to be enabled and disabled at the start and end of transmission to allow other devices to send data on the bus.

This application note describes how the Cyan Technology eCOG1k can be used to communicate on a common serial bus and control the transmitter on that bus.

2 Glossary

A table of abbreviations used in this document.

| | |
|-------|---|
| eCOG1 | Cyan Technology target micro controller |
| UART | Universal Asynchronous Receiver / Transmitter |
| GPIO | General Purpose Input / Output |

The on-chip I/O registers may be accessed as complete registers, or as named bit fields within the registers. The include file *ecog1.h* contains structure definitions for all on-chip registers and bit fields. To access any register, use the structure prefix **rg**. To access a bit field within a register, use the structure prefix **fd**. For example:

```
// No timeouts on DUART A
rg.duart.a_tmr_cfg = 0;
// Enable DUART A transmitter
fd.duart.ctrl.a_tx_en = 1;
```

3 Overview

The example code in this application note uses channel A of the DUART within the eCOG1k. The transmit and receive signals are connected to the external transceiver device that provides the electrical interface to the serial bus.

Two GPIO lines are used to control the enable and disable of the transmit and receive sides of the transceiver. The transmit side is enabled only when the eCOG1k has data to transmit and is disabled at the end of transmission. The receive side is usually always enabled, but a disable control for the receiver is also provided, as some transceivers enter a low power mode when both the transmit and receive sides are disabled.

In the example code, macros are defined at the start of the code to control the enable and disable of the transmitter and receiver.

Interrupts are used to handle the flow of data in to and out of the UART. In combination with transmit and receiver buffers, these simplify the main application code.

4 Data Buffers

To improve the data flow and to reduce the load on the main application, two circular buffers are used to hold the transmit and receive data sent to and received from the UART.

These are implemented as circular buffers with fill and empty indices that follow each other around in a circle. Any data added to the buffer is placed in at the current fill index and the index is incremented. If the index has exceeded the size of the buffer, it is reset to the start of the buffer at zero. When data is removed from the buffer, it is taken from the empty index and the index is incremented. Again, if the index has exceeded the size of the buffer, it is reset to the start of the buffer at zero.

The amount of data in the buffer can be determined by subtracting the empty index from the fill index. If the result is negative, then the fill index has wrapped around, but the empty index has not, so by adding the total buffer size the amount of data can be calculated.

In this example, the `txchar` function checks the size of the transmit buffer, and if it is full, it waits until there is enough space for the byte. This may not be desirable behaviour in some applications, as this will hold up the execution of the application until a byte has been transmitted.

In the `rxchar` function, the code waits for data to become available before it returns.

In the receiver interrupt service routine, if the receive buffer is full then the data is discarded as waiting in the interrupt routine would lock up the whole system.

5 Transmitter Control

For this application, the transmitter of the serial bus transceiver needs to be enabled at the start of a block of data and disabled again at the end. The UART in the eCOG1k does not provide an interrupt at the end of transmission of a frame (data byte with start bit, stop bit and optional parity). However, it is possible to use the receiver to determine when the character has been transmitted and then to disable the transmitter at the end of a frame.

As there is buffering in the transmit side of the UART itself, a count must be kept of how many bytes are in the process of being transmitted. The transmitter should be disabled only when the last transmitted character has been received back by the receiver.

An advantage of this method is that the integrity of the data on the bus can be checked and errors caused by noise, data collisions, etc. can be detected.

The transmitter is enabled by the transmitter interrupt service routine when data is available to be transmitted. In this example, the variable `txframecount` is used for keeping track of the number of bytes that have been put into the UART transmitter and it is incremented every time a new byte is placed into the transmitter.

When a byte is received, the receiver interrupt service routine tests `txframecount` and if it is not zero then the data is assumed to be data from our own transmitter. The value of `txframecount` is then decremented and if it is now zero, the transmitter is disabled as we know that there is no pending data in the UART transmitter.

5.1 Error Checking

By comparing the received data against the transmitted data, transmission errors can be detected.

When a byte is received and `txframecount` is not zero, it can be used to look back in the transmitter buffer relative to the empty index and check what character was transmitted. The received and transmitted byte can be compared and an error flagged if they are not the same.

As this history of transmitted data must be kept in the transmit data buffer, the maximum size of the data buffer is effectively reduced. This is a maximum of three bytes (one in the receive register, one in the transmit shift register and one in the transmit buffer); a macro `BUFFER_OVERLAP` is used to provide this offset in the buffer size calculations and to control the error checking. It should be set to three to enable the error checking and zero to disable it.

It may be necessary to disable the error checking at high data rates when the interrupt processing time becomes significant.

Any errors detected are indicated by setting a flag which should be tested by the main application. The clearing of this flag and any error recovery are dependent on the higher levels of the communications protocol that is used across the bus. Details of this are not shown here as it is outside the scope of this document.

Appendix A Software Listing

A.1 RS485.c

```

/*=====
Cyan Technology Limited

FILE - RS485.c
      Example eCOG1 application - buffered interrupt driven serial port
      for two wire RS485 serial bus.

DESCRIPTION
      This example uses duart channel A

MODIFICATION DETAILS
=====*/

/*****
Project level include files.
*****/

#include <ecog.h>
#include <ecog1.h>
#include "driver_lib.h"

/*****
Include files for public interfaces from other modules.
*****/

/*****
Declaration of public functions.
*****/

/*****
Private constants and types.
*****/

// Buffer sizes
#define TXBUFSIZE          20
#define RXBUFSIZE          20

// Transmit buffer overlap
// Set to 0 to disable error checking
// Set to 3 to enable error checking
#define BUFFER_OVERLAP      3

// Enable/Disable macros for the RS485 Transceiver
// Transmitter enable is assumed to be active high
#define RS485_TX_ENABLE      fd.io.gp0_3_out.set0 = 1
#define RS485_TX_DISABLE    fd.io.gp0_3_out.clr0 = 1
// Receiver enable is assumed to be active low.
#define RS485_RX_ENABLE      fd.io.gp0_3_out.clr1 = 1
#define RS485_RX_DISABLE    fd.io.gp0_3_out.set1 = 1

/*****
Declaration of static functions.
*****/

void txchar(int c);
void txstring(char *s);
int rxchar(void);

int txcount(void);
int rxcount(void);

```

```

// Called from irq service routines
int __irq_code itxcount(void); // Read number of characters in transmit buffer
int __irq_code irxcount(void); // Read number of characters in receive buffer

/*****
Global variables.
*****/

char txbuf[TXBUFSIZE]; // Transmit buffer
char rxbuf[RXBUFSIZE]; // Receive buffer

// Buffer index values
int txepr; // Transmit buffer empty index
int txfpr; // Transmit buffer fill index
int rxepr; // Receive buffer empty index
int rxfpr; // Receive buffer fill index

int txframecount; // Transmitter frame counter

// TX error flag
int txerrorflag;

/*****
Module global variables.
*****/

/*****
Definition of API functions.
*****/

/*****
NAME
    main

SYNOPSIS
    int main(int argc, char* argv[])

FUNCTION
    Simple main function to demonstrate the use of the RS485 code.

RETURNS
    Exit Code
*****/

int main(int argc, char* argv[])
{
    int count = 0;
    int c;

    // Initialise the Serial Interface
    initialise();
    // Initialise the LCD on the Evaluation Board
    lcd_rst();
    lcd_puts("Example: RS485");
    lcd_xy(1, 2);

    // Send a "Hello World" sign-on message to the bus
    txstring("Hello World");

    // Main loop - read any received characters and echo them to the LCD
    while (1)
    {
        // Has a collision occurred?
        if (txerrorflag)
        {
            // Clear the flag
            txerrorflag = 0;
            // Print a message
            lcd_xy(1,1);
            lcd_puts("***COLLISION***");
            // Restore the current data position
            lcd_xy(c+1, 2);
        }
    }
}

```

```

        // Any characters received?
        if (rxcount() > 0)
        {
            c = rxchar();

            // Count character (for lcd) wrapping 16 characters
            count++;
            count %= 16;
            if (count == 0)
            {
                lcd_xy(1, 2);
            }
            lcd_putc(c);        // Send character to lcd
        }
    }

    return 0;
}

/*****
NAME
    initialise

SYNOPSIS
    void initialise(void)

FUNCTION
    Initialises all the hardware for the DUART channel A

RETURNS
    Nothing
*****/

void initialise(void)
{
    // Initialise the buffer pointers the frame counter and the error flag
    txeptr = 0;
    txfptr = 0;
    rxeptr = 0;
    rxfptr = 0;
    txframecount = 0;
    txerrorflag = 0;

    // Disable the RS485 transmitter
    RS485_TX_DISABLE;
    // Enable the RS485 receiver
    RS485_RX_ENABLE;

    // Set duart clock to 100MHz / 8 = 12.5 MHz
    ssm_duart_clk(SSM_HIGH_PLL, 1);

    // Configure duart B
    rg.duart.frame_cfg = 0;
    rg.duart.a_tmr_cfg = 0;        // No timeouts
    rg.duart.a_baud = 80;        // 9600 baud

    // Enable uart
    fd.duart.ctrl.a_tx_en = 1;    // Enable tx
    fd.duart.ctrl.a_rx_en = 1;    // Enable rx

    // Enable receive interrupt
    fd.duart.a_int_en.rx_lb_rdy = 1;
}

//-----
// Put character in transmit buffer
void txchar(int c)
{
    // Wait for space in output buffer
    while (txcount() >= (TXBUFSIZE - BUFFER_OVERLAP - 1));
}

```

```

    // Put character into buffer
    txbuf[txfptr++] = (char)c;
    // Check for index pointer wraparound
    if (txfptr >= TXBUFSIZE)
    {
        txfptr = 0;
    }

    // Enable transmit ready interrupt
    fd.duart.a_int_en.tx_rdy = 1;
}

//-----
// Transmit a string of characters
void txstring(char *s)
{
    // Send a string of characters until we hit the terminating /0
    while(*s)
    {
        txchar(*s++);
    }
}

//-----
// Read character from receive buffer
int rxchar(void)
{
    int c;

    // Wait for data to become available
    while (0 == rxcount());

    // Read character from buffer
    c = (int)(rxbuf[rxeptr++]);
    // Check for index pointer wraparound
    if (rxeptr >= RXBUFSIZE)
    {
        rxeptr = 0;
    }

    return (c);
}

//-----
// Returns number of characters in transmit buffer
int txcount(void)
{
    int chars_pending = txfptr - txeptr;

    if (chars_pending < 0)
    {
        chars_pending += TXBUFSIZE;
    }

    return (chars_pending);
}

//-----
// Returns number of characters in receive buffer
int rxcount(void)
{
    int chars_received = rxfptr - rxeptr;

    if (chars_received < 0)
    {
        chars_received += RXBUFSIZE;
    }

    return (chars_received);
}

```

```

/*****
NAME
    RS485_tx_handler

SYNOPSIS
    void __irq_entry RS485_tx_handler(void)

FUNCTION
    Interrupt handler for RS485 transmit ready

RETURNS
    Nothing.
*****/

void __irq_entry RS485_tx_handler(void)
{
    // Read tx interrupt status register
    int test = fd.duart.a_sts.tx_rdy;

    // Transmitter ready?
    if (1 == test)
    {
        // Turn the transmitter on and increment the frame count
        RS485_TX_ENABLE;
        txframecount++;

        // Yes, any characters to send?
        if (itxcount() > 0)
        {
            // Yes, get character from buffer
            rg.duart.a_tx8 = txbuf[txeptr++];
            // Check for index pointer wraparound
            if (txeptr >= TXBUFSIZE)
            {
                txeptr = 0;
            }
        }
        else
        {
            // No characters left in transmit buffer
            // Disable transmitter ready interrupt
            fd.duart.a_int_dis.tx_rdy = 1;
        }
    }
}

/*****
NAME
    RS485_rx_handler

SYNOPSIS
    void __irq_entry RS485_rx_handler(void)

FUNCTION
    Interrupt handler for RS485 receive ready

RETURNS
    Nothing.
*****/

void __irq_entry RS485_rx_handler(void)
{
    // Read rx interrupt status register
    int test = fd.duart.a_sts.rx_lb_rdy;

    // Received character ready?
    if (1 == test)
    {
        // Yes, read receive register
        char c = rg.duart.a_rx;

        // Check the frame counter to see if this was data we transmitted
        if (0 != txframecount)
        {
            #if BUFFER_OVERLAP

```

```

        int index = txeptr - txframecount;

        if (index < 0)
        {
            index += TXBUFSIZE;
        }

        // Check the data against the buffer
        if (c != txbuf[index])
        {
            // The data is not the same so flag an error
            txerrorflag = -1;
        }
#endif

        // Discard the data, decrement the frame count and if it is
        // now zero, then disable the RS485 transmitter.
        txframecount--;
        if (0 == txframecount)
        {
            RS485_TX_DISABLE;
        }
    }
    // Is there space in receive buffer?
    else if (irxcount() < (RXBUFSIZE - 1))
    {
        // Yes, put character into input buffer
        rxbuf[rxfptr++] = c;
        // Check for index pointer wraparound
        if (rxfptr >= RXBUFSIZE)
        {
            rxfptr = 0;
        }
    }
}

// Duplicate buffer count routines for use by interrupt service routines
//-----
// Return number of characters in transmit buffer
int __irq_code itxcount(void)
{
    int chars_pending = txfptr - txeptr;

    if (chars_pending < 0)
    {
        chars_pending += TXBUFSIZE;
    }

    return (chars_pending);
}

//-----
// Return number of characters in receive buffer
int __irq_code irxcount(void)
{
    int chars_received = rxfptr - rxeptr;

    if (chars_received < 0)
    {
        chars_received += RXBUFSIZE;
    }

    return (chars_received);
}

```

```

/*****
NAME
    RS485_ex_handler

SYNOPSIS
    void __irq_entry usr_ex_handler(void)

FUNCTION
    Interrupt handler for usr exceptions

RETURNS
    Nothing.
*****/

void __irq_entry RS485_ex_handler(void)
{
    // Clear any exceptions
    rg.duart.a_int_clr = 0xffff2;
    // Disable any exceptions
    rg.duart.a_int_dis = 0xffff2;
}

```

A.2 irq.asm

```

;=====
; Cyan Technology Ltd
; Example application software for eCOG
;
; FILE - irq.asm
; Contains the interrupt vectors for the RS485 example code.
;=====

                MODULE    irq
                .ALL

; Interrupt vectors are sign-extended 24 bit absolute branch addresses.
; For example a vector containing H'1234 will start executing at H'1234
; and a vector containing H'8765 will start executing at H'ff8765.
; All unused interrupts are handled by the default code in minimal_handler.

                ORG      H'0
                bra      $?cstart_code

                ORG      H'4
ex_debug       DC $minimal_handler
ex_tim_wdog_ufl DC $minimal_handler
ex_adr_err     DC $minimal_handler
reserved      DC $minimal_handler
ex_tim        DC $minimal_handler
ex_reserved   DC $minimal_handler
ex_usart_a    DC $minimal_handler
ex_usart_b    DC $minimal_handler
ex_uart_a    DC $RS485_ex_handler??
ex_uart_b    DC $minimal_handler
int_tim_tmr_ufl DC $minimal_handler
int_tim_cnt1_ufl DC $minimal_handler
int_tim_cnt2_ufl DC $minimal_handler
int_tim_cnt1_match DC $minimal_handler
int_tim_cnt2_match DC $minimal_handler
int_tim_pwm1_ufl DC $minimal_handler
int_tim_pwm2_ufl DC $minimal_handler
int_tim_pwm1_match DC $minimal_handler
int_tim_pwm2_match DC $minimal_handler
int_tim_cap_ofl DC $minimal_handler
int_tim_cap1  DC $minimal_handler
int_tim_cap2  DC $minimal_handler
int_tim_cap3  DC $minimal_handler
int_tim_cap4  DC $minimal_handler
int_tim_cap5  DC $minimal_handler
int_tim_cap6  DC $minimal_handler
int_tim_ltmr_ufl DC $minimal_handler
int_reserved1 DC $minimal_handler
int_reserved2 DC $minimal_handler
int_reserved3 DC $minimal_handler
int_reserved4 DC $minimal_handler
int_reserved5 DC $minimal_handler
int_reserved6 DC $minimal_handler
int_reserved7 DC $minimal_handler
int_reserved8 DC $minimal_handler
int_usart_a_rx DC $minimal_handler
int_usart_a_tx DC $minimal_handler
int_usart_b_rx DC $minimal_handler
int_usart_b_tx DC $minimal_handler
int_sci_tx_done DC $minimal_handler
int_sci_tx_err DC $minimal_handler
int_sci       DC $minimal_handler
int_ifr_tx_done DC $minimal_handler
int_ifr_rx_done DC $minimal_handler
int_ifr_rx_err DC $minimal_handler
int_ifr_frame_done DC $minimal_handler
int_uart_a_tx_rdy DC $RS485_tx_handler??
int_uart_a_rx_rdy DC $RS485_rx_Handler??
int_uart_b_tx_rdy DC $minimal_handler
int_uart_b_rx_rdy DC $minimal_handler
int_ehi       DC $minimal_handler
int_gpio      DC $minimal_handler
int_adc       DC $minimal_handler

                ENDMOD

```