Michael Tangy

ECE 381 – Spring 2015

Lab 6

I2C SRAM

The purpose of this lab was for us to gain familiarity with the I2C protocol as well as how to use it to communicate with our two PCF8570 SRAM chips. The lab consisted of us asking the user to enter read and write commands in a certain format over the RS232 interface. The commands consists of R/W character as well as a slave address to determine which SRAM chip your writing to and a Memory chip address to tell SRAM what memory cell address you want to begin writing or reading to. Our commands also give the option to either write or read ASCII or hexadecimal characters

Parts Needed:

1- PSoC Microcontroller

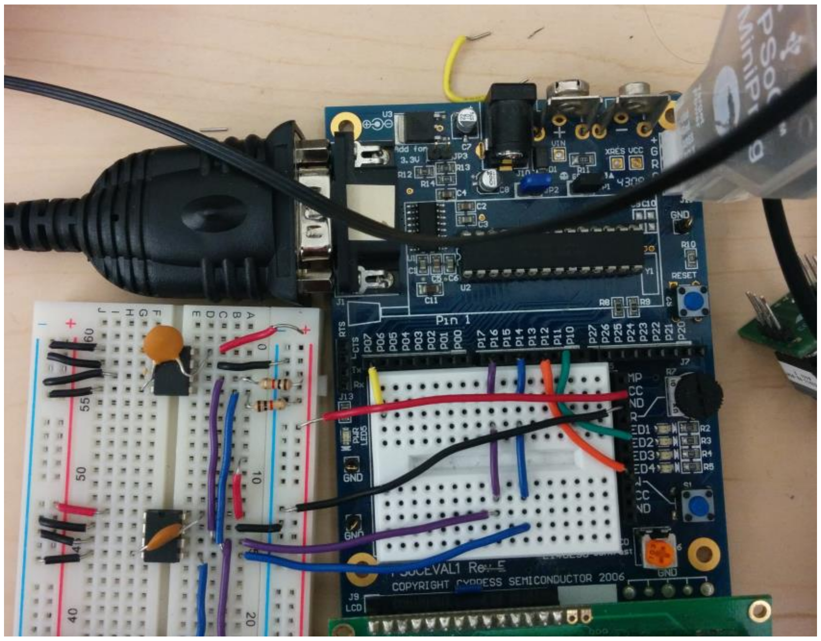
1- DB-9 to DB-9 cable or DB-9 to USB serial cable

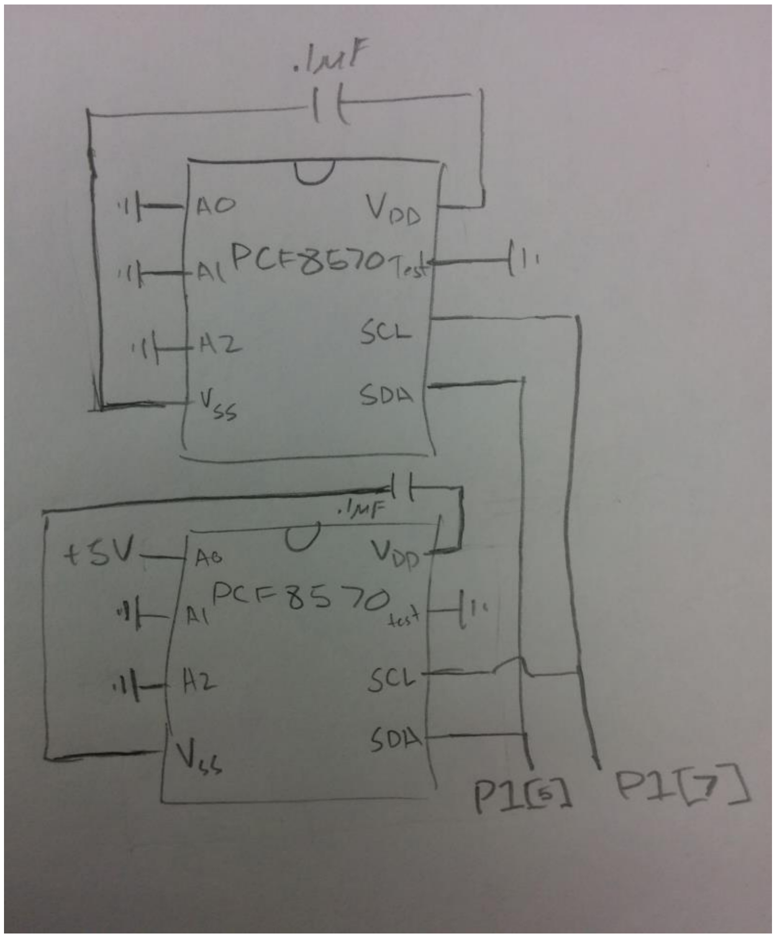
2- PCF8570 SRAM chips

2- .1μF ceramic capacitor

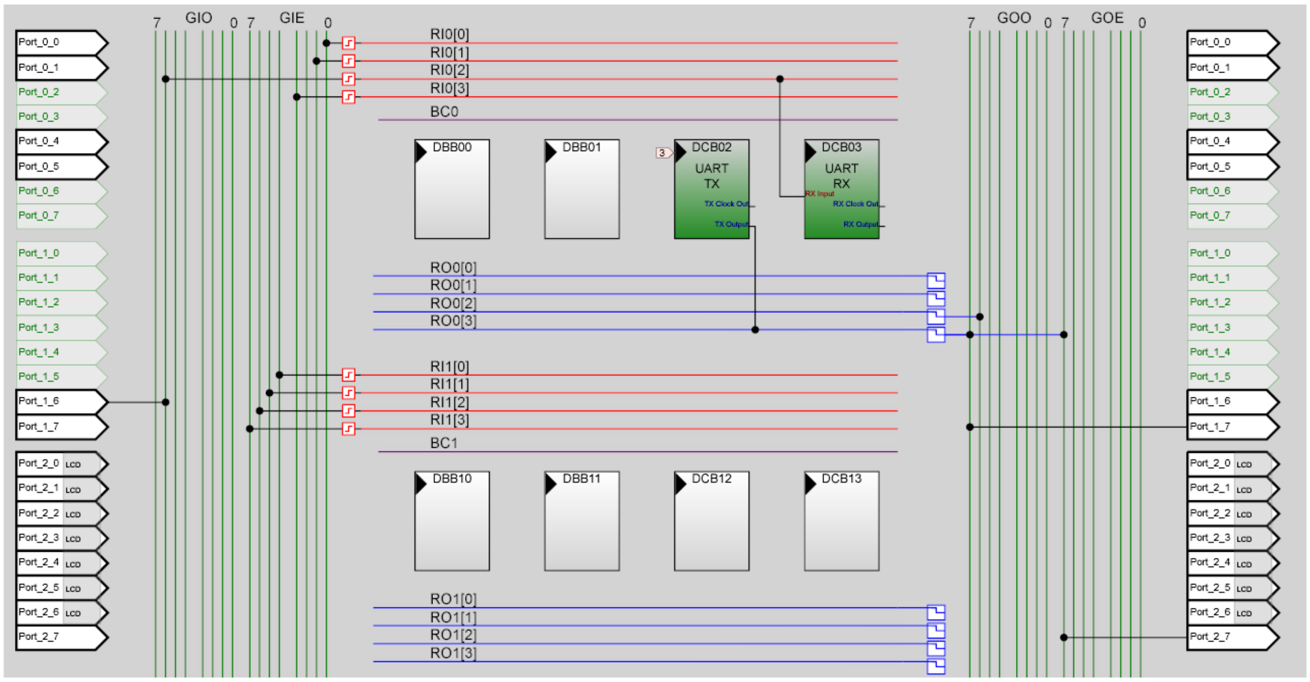
Hardware/Pin-Out Configuration:

After hooking your PSoC up with your computer you need to wire your SRAM chips to your PSoC. This involves setting your group two address with the three address pins on your SRAM chip. These three pins can be set to any combination of power and ground as long as the two are different (its preferred to set one to 000 and the other to 001). Next tie your power and ground pins to power and ground (pins 4 and 8 respectively) and also tie the test pin (#7) to ground. Next you want to connect your SCL and SDA I2C line's to your PSoC they go into port one pin five and port one pin seven respectively. After that you want to add your .1μF ceramic decupling capacitor to each chip, they should be placed in between the power and ground pins, and there leads should be cut as short as possible to reduce noise.

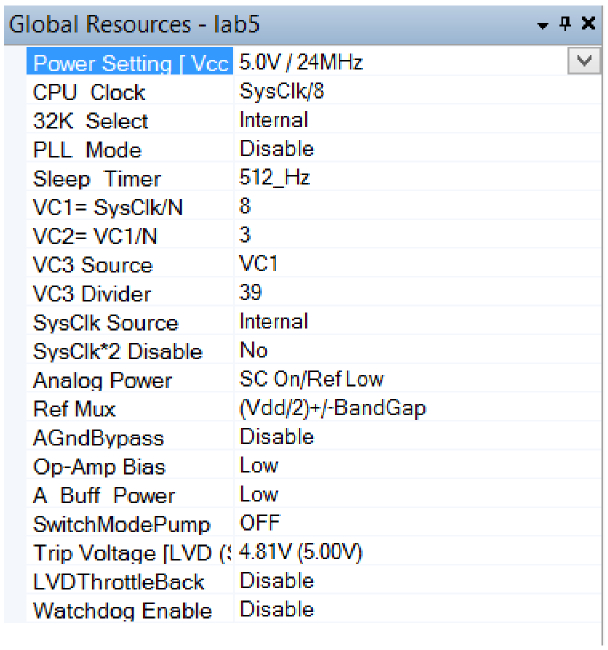



.1μF
A0
A1
A2
Vss
VDD
PCF8570
Test
SCL
SDA
.1μF
+5V
A0
A1
A2
Vss
VDD
PCF8570
test
SCL
SDA
P1[5]
P1[7]

Now you're ready to create your project, clone the previous project to inherit all the UART setting and configuration. If you chose not to clone the previous be sure to assign your PSoC port 1 pin 6 (P1[8]) to the UART's RX line and port 1 pin 7 (P1[7]) to the UART's TX line, also change the variable names at those pins. This allows you to view the binary bit streams you're sending and receiving from your PSoC. This can be done from the chip level view of your project by shift-clicking on the UART modules outputs and assigning them to the appropriate pins. Below is how the chip level view of your project should look.



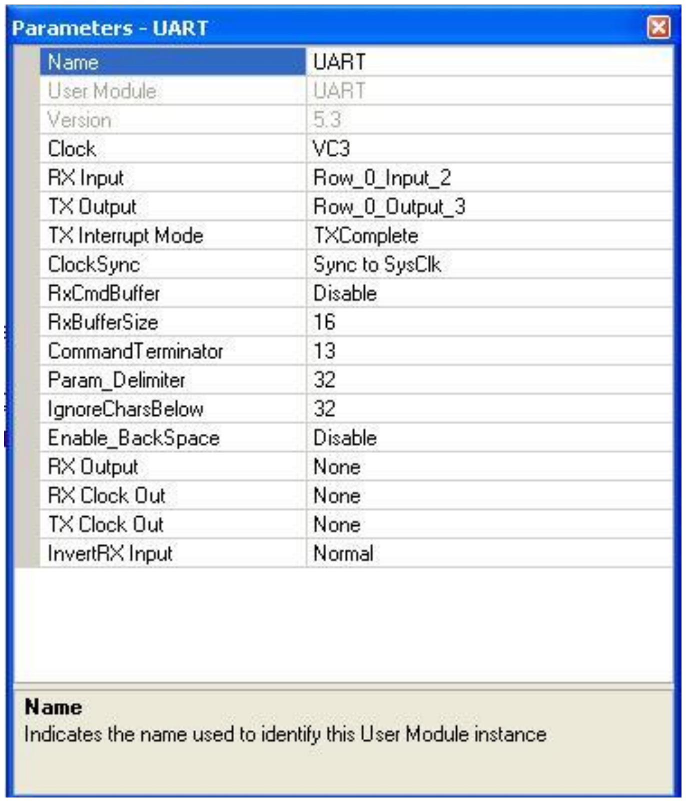
Next you need to configure a clock source to drive the UART. Since the required baud rate is 9600 we need to set our source clock to three times that (76,800Hz). In the UART data sheet we discovered the best way to do this was to divide the system clock by 8 and 39 and set it to VC3 as shown below:



Global Resources - lab5

Power Setting [Vcc	5.0V / 24MHz
CPU Clock	SysClk/8
32K Select	Internal
PLL Mode	Disable
Sleep Timer	512_Hz
VC1= SysClk/N	8
VC2= VC1/N	3
VC3 Source	VC1
VC3 Divider	39
SysClk Source	Internal
SysClk*2 Disable	No
Analog Power	SC On/Ref Low
Ref Mux	(Vdd/2)+/-BandGap
AGndBypass	Disable
Op-Amp Bias	Low
A Buff Power	Low
SwitchModePump	OFF
Trip Voltage [LVD (	4.81V (5.00V)
LVDThrottleBack	Disable
Watchdog Enable	Disable

Next you need to change the properties of your UART module to make sure everything's configured properly. The clock should be set to VC3 and the RxCmdBuffer should be disabled the UART properties window should look like this:



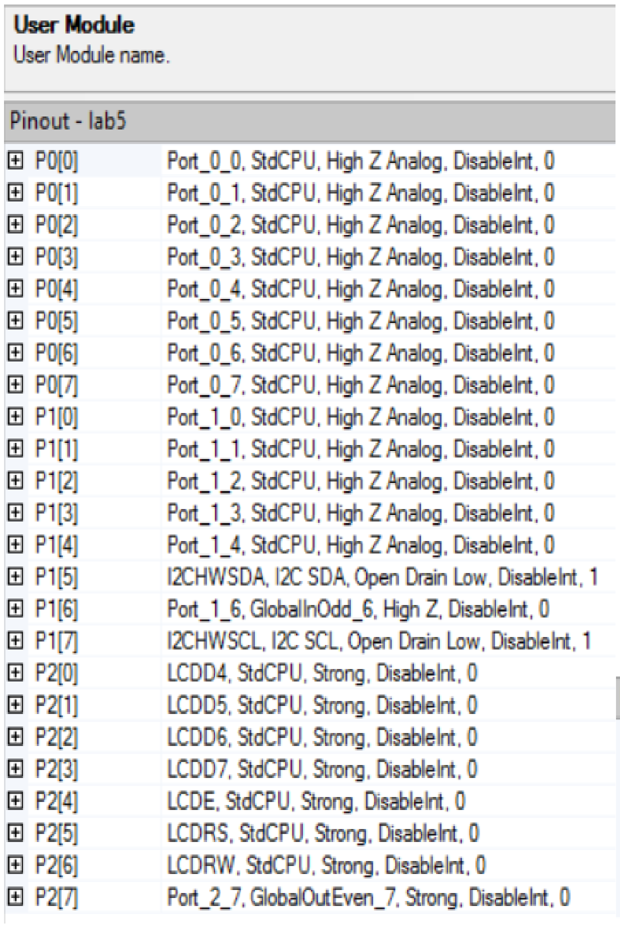
Parameters - UART

Name	UART
User Module	UART
Version	5.3
Clock	VC3
RX Input	Row_0_Input_2
TX Output	Row_0_Output_3
TX Interrupt Mode	TXComplete
ClockSync	Sync to SysClk
RxCmdBuffer	Disable
RxBufferSize	16
CommandTerminator	13
Param_Delimiter	32
IgnoreCharsBelow	32
Enable_BackSpace	Disable
RX Output	None
RX Clock Out	None
TX Clock Out	None
InvertRX Input	Normal

Name
Indicates the name used to identify this User Module instance

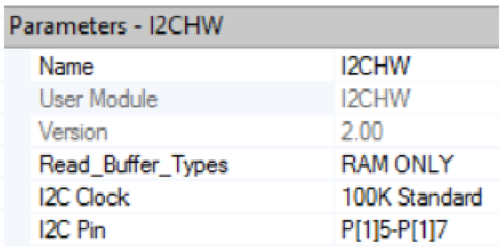
Next you should make sure your LCD is set to port two through its module properties. The Real Term software used in interface with the UART needs to configured with its Baud rate set to 9600 and its port set 1 if your UART is connected to COMA or COM1 or set to port 2 if its connect to COMB or COM2.

Next you'll set the CPU system clock to SysClk/1 then you'll have to add an I2C system module to the chip level view (however the module does not use a digital block so it's not visible) of your project in order to use the appropriate functions from the API. Refer to the tables below to confirm your settings:



User Module
User Module name.

Pinout - lab5	
P0[0]	Port_0_0, StdCPU, High Z Analog, DisableInt, 0
P0[1]	Port_0_1, StdCPU, High Z Analog, DisableInt, 0
P0[2]	Port_0_2, StdCPU, High Z Analog, DisableInt, 0
P0[3]	Port_0_3, StdCPU, High Z Analog, DisableInt, 0
P0[4]	Port_0_4, StdCPU, High Z Analog, DisableInt, 0
P0[5]	Port_0_5, StdCPU, High Z Analog, DisableInt, 0
P0[6]	Port_0_6, StdCPU, High Z Analog, DisableInt, 0
P0[7]	Port_0_7, StdCPU, High Z Analog, DisableInt, 0
P1[0]	Port_1_0, StdCPU, High Z Analog, DisableInt, 0
P1[1]	Port_1_1, StdCPU, High Z Analog, DisableInt, 0
P1[2]	Port_1_2, StdCPU, High Z Analog, DisableInt, 0
P1[3]	Port_1_3, StdCPU, High Z Analog, DisableInt, 0
P1[4]	Port_1_4, StdCPU, High Z Analog, DisableInt, 0
P1[5]	I2CHWSDA, I2C SDA, Open Drain Low, DisableInt, 1
P1[6]	Port_1_6, GlobalInOdd_6, High Z, DisableInt, 0
P1[7]	I2CHWSCL, I2C SCL, Open Drain Low, DisableInt, 1
P2[0]	LCDD4, StdCPU, Strong, DisableInt, 0
P2[1]	LCDD5, StdCPU, Strong, DisableInt, 0
P2[2]	LCDD6, StdCPU, Strong, DisableInt, 0
P2[3]	LCDD7, StdCPU, Strong, DisableInt, 0
P2[4]	LCDE, StdCPU, Strong, DisableInt, 0
P2[5]	LCDRS, StdCPU, Strong, DisableInt, 0
P2[6]	LCDRW, StdCPU, Strong, DisableInt, 0
P2[7]	Port_2_7, GlobalOutEven_7, Strong, DisableInt, 0



Parameters - I2CHW	
Name	I2CHW
User Module	I2CHW
Version	2.00
Read_Buffer_Types	RAM ONLY
I2C Clock	100K Standard
I2C Pin	P[1]5-P[1]7

<u>Software Description:</u>

After configuring the pin-out settings, chip level settings and the real term settings you’re ready to write your program. All your variables, pointers and constant strings should be declared globally above your main method. Your main method should initialize your strings to null as well as start you UART and set its parity to none as well as all other I2C startup functions. From there your program should print your initial message (asking if they want to R/W) to the UART’s console and read in the character they input, then the program should enter an infinite while loop. You’re infinite while loop calls the instruction function, read string and parse string function over and over again so the user can continuously read and write without restarting the program. The read function reads the entire string written by the user and prepares it to be parsed. The parse function assignees variables to prepare them to be written to RAM. The R/W character choices which parse is going to take place the slave address determines which ram chip will by assigned to the read and wright I2C functions. The address in should be loaded to the first bytes in the array and data in the string should loaded in the array byte by byte in order to write them to RAM.

<u>CODE:</u>

```
//----------------------------------------------------------------------------
-
// C main line
//----------------------------------------------------------------------------
-

#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules
#include <string.h>
#include "string.h"

#include "stdlib.h"

//#include "stdbool.h"
//#include <stdbool.h>
```

```
#include "I2CHWMstr.h"

//char register [7];

//strings to read in and parse
char string0[17];
char string1[17];
char string2[17];
char string3[17];

//ram address vaibles
char RorW_address;
char RorW_address1;
char RorW_address2;

//global array's and vairbles
char address[2];
char messageString[79];
int parseCounter=0;
int stringSize=0;
int readSize=0;
int addressInt=0;
char addressHex;
int i;
char sizeOfString;

//system control flags
BOOL hexOrAscii = FALSE;
BOOL finishFlag = FALSE;
BOOL invaildFlag = FALSE;
BOOL readOrWrite = FALSE;
BOOL firstPass = TRUE;


/* Define slave address */
#define Ram1 0x50
#define Ram2 0x51  //NOTE:1 may need to be changed to a 2 to make room for
the R/W bit
char ramForUse = 0x00;


/* Define buffer size */
#define BUFFER_SIZE 0xFF


//read and write byte buffer arrays
BYTE txBuffer[BUFFER_SIZE];
BYTE rxBuffer[BUFFER_SIZE];
BYTE status;

BYTE* Address_Pointer;
//BYTE* Write_Address_Pointer;

BYTE Read_Address = 0x01 ;
BYTE Write_Address = 0x01;

char charIn = '\0';
```

```c
char charIn1= '\0';
char charIn2= '\0';
char charIn3= '\0';
char charIn4= '\0';

char instBuffer= '\0';
int placeCounter = 0;
int count;


char rawString[79];

const char welcomeMsg[] = "Welcome.";

//"W # XX A Mary had a little lamb"

//function prototypes
void startFunction(void);
void readFunction(void);
void stringParser(int placeCounter);
void instructions(void);
void readFromRam(void);
void writeToRam(void);
char asciiToHex(char addressByte);


void main(void)
{
//Enable Interupt calls and start LCD function call
// I2CHW_Start();
I2CHW_EnableMstr();
M8C_EnableGInt;
I2CHW_EnableInt();
LCD_Start();
UART_Start(UART_PARITY_NONE);

i =0;

//I2CHW_InitRamRead(txBuffer, BUFFER_SIZE);

//Write_Address = 0x00;
//Read_Address = 0x00;

// assign buffer values to write to RAM
txBuffer[0] = 0x00;
txBuffer[1] = 0x51;
txBuffer[2] = 0x33;
txBuffer[3] = 0x55;
txBuffer[4] = 0x33;
txBuffer[5] = 0x55;
txBuffer[6] = 0x33;
txBuffer[7] = 0x55;

//txBuffer[8] = 0x55;


// while to continuously write and read from RAM
```

```
startFunction();
while(count < 100)
{
writeToRam();
readFromRam();
count++;

//                  if(!firstPass)
//                  {
//                  instructions();
//                  }
//
//                  readFunction();

//                  stringParser(placeCounter);
//
//                  if (invaildFlag)
//                  {
//                                    UART_PutChar(0x07);
//                          UART_CPutString("This string was invalid");
//UART_CPutString("Invalid flag is true");
//UART_PutCRLF();
//UART_CPutString(instructMsg);

//                  }//end invalid flag if
//                  else
//                  {
//                          //This is where the good I2c stuff goes
//                          //Probably ought to write separate read and write
functions
//                          //and switch on a flag set in stringParser()
//
//                          if (readOrWrite)
//                          {

// }//end if read
//else
// {

//}//end else

// }// end invalid if else statement

//firstPass=FALSE;//Allows instructions to print again
//invaildFlag=FALSE;//Resets validator

}//End while(1)


/// Outputting values written to ram
UART_PutCRLF();
UART_CPutString("Output from RAM: ");
UART_PutCRLF();
UART_PutChar(rxBuffer[0]);
UART_PutCRLF();
UART_PutChar(rxBuffer[1]);
UART_PutCRLF();
```

```
UART_PutChar(rxBuffer[2]);
UART_PutCRLF();
UART_PutChar(rxBuffer[3]);
UART_PutCRLF();
UART_PutChar(rxBuffer[4]);
UART_PutCRLF();
UART_PutChar(rxBuffer[5]);
UART_PutCRLF();
UART_PutChar(rxBuffer[6]);
UART_PutCRLF();
UART_PutChar(rxBuffer[7]);

/// Outputting values read from ram
UART_PutCRLF();
UART_CPutString("Bytes written to RAM: ");
UART_PutCRLF();
UART_CPutString("ram chip 1 , memory location 00");
UART_PutCRLF();
UART_PutChar(txBuffer[1]);
UART_PutCRLF();
UART_PutChar(txBuffer[2]);
UART_PutCRLF();
UART_PutChar(txBuffer[3]);
UART_PutCRLF();
UART_PutChar(txBuffer[4]);
UART_PutCRLF();
UART_PutChar(txBuffer[5]);
UART_PutCRLF();
UART_PutChar(txBuffer[6]);
UART_PutCRLF();
UART_PutChar(txBuffer[7]);
UART_PutCRLF();
UART_PutChar(txBuffer[8]);



} //End Main

void startFunction (void)
{
UART_CPutString(welcomeMsg);
instructions();

}//end startFunction
//void readFunction (void)
//{
//         placeCounter = 0;
//         finishFlag= FALSE;
//
//         while (!finishFlag) {
//
//                 charIn = UART_cReadChar();
//                         while  (charIn == 0x00){charIn = UART_cReadChar();}
//                                 if (placeCounter<78){
//
//                             if (charIn == 0x0d){ //if carriage return
//                                                UART_CPutString("\r\n");
//                                                UART_CPutString("\r\n");
//                                            finishFlag = TRUE;
//                                    } //end if CR
//                                                    //backspace
//                                else if ((charIn == 0x08 || charIn ==0x7f)
&& placeCounter > 0 ){ //if backspace
//                                        placeCounter--;
//                                        UART_CPutString("\x8\x20\x8");
//                                } //end if backspace
//                                else {
//                                                rawString[placeCounter] =
charIn;
//
UART_PutChar(rawString[placeCounter]);
//                                                placeCounter++;
//                                        }// end else write into string
//
//                                } // end placeCounter if
//                        else
//                                {
//
//                                        UART_CPutString("String Full");
//
      UART_PutChar(0x07);
//                                        finishFlag = TRUE;
//                                                    //addbell
//                                }
//
//        }//end while (~finsihFlag)
//
//        finishFlag= FALSE ;
//
//}//end readFunction
//
//void stringParser(int parseCounter)
//{
//    if(rawString[0]>= 0x41 && rawString[0]<= 0x5a){
//          rawString[0] = rawString[0]+0x20;
//    }
//        switch (rawString[0])//Determines if operation is read or write
//        {
//                case  'r':
//
//                UART_CPutString("read");
//                UART_PutCRLF();
//                readOrWrite = TRUE;
//
//                break; //end case r
//
//                case 'w':
//
//                UART_CPutString("write");
//                UART_PutCRLF();
//                readOrWrite = FALSE;
//                break; //end case w
//
//                default:
```

```
//                  invaildFlag = TRUE;
//
//                  UART_CPutString("r/w fail");
//                  UART_PutCRLF();
//
//
//          }//end read/write switch
//
//          switch (rawString[2])//determines RAM to be written to
//          {
//                  case  '0':
//
//                  UART_CPutString("ram 1");
//                  UART_PutCRLF();
//                  ramForUse = Ram1;
//
//                  break; //end case 0
//
//                  case '1':
//
//
//                  UART_CPutString("ram 2");
//                  UART_PutCRLF();
//                  ramForUse = Ram2;
//
//                  break; //end case 1
//
//                  default:
//
//                  UART_CPutString("ram fail");
//                  UART_PutCRLF();
//                  invaildFlag = TRUE;
//
//          }//end address switch
//
//          //This writes the address to a string called 'address' and an int
'addressInt'
//          //We need to make this check if the hex is valid
//
//
//            RorW_address1 = asciiToHex(rawString[4]);
//            RorW_address2 = asciiToHex(rawString[5]);
//
//            RorW_address1 = RorW_address1 << 4;
//            RorW_address = RorW_address1 + RorW_address2;
//
//
//          //Should change the 2 byte ascii address string into a one byte hex
address char
//          //addressHex= asciiToHex(address[0])*16 + asciiToHex(address[1]);
//
//            UART_PutCRLF();
//          UART_CPutString("address: ");
//          UART_PutChar(RorW_address);
//          UART_PutCRLF();
//
//
```

```
//          if(rawString[7]>= 0x41 && rawString[7]<= 0x5a)
//          {
//                rawString[7] = rawString[7]+0x20;
//          }
//
//          switch (rawString[7])
//
//                {
//                case  'a':
//
//                UART_CPutString("ascii");
//                hexOrAscii = FALSE;
//                UART_PutCRLF();
//
//                break; //end case 0
//
//                case 'h':
//
//                UART_CPutString("hex");
//                hexOrAscii = TRUE;
//                UART_PutCRLF();
//
//                break; //end case 1
//
//                default:
//                UART_CPutString("ascii/hex fail");
//                UART_PutCRLF();
//
//                invaildFlag = TRUE;
//
//        }//end ascii/hex switch
//        parseCounter = 1;
//
//        UART_PutCRLF();
//        UART_CPutString("placeCounter set ");
//        UART_PutCRLF();
//
//        messageString[0] = addressHex;
//
//        while((parseCounter+8) < placeCounter)
//        {
//                messageString[parseCounter] = rawString[(parseCounter+8)];
//                parseCounter++;
//
//        }//end parseCounter while
//
//          if( (parseCounter+8) == placeCounter){
//
//          messageString[placeCounter + 3] = '\0';  // something like this -
needs to be null terminated
//
//    }
//
//        UART_PutString(messageString);
//        UART_PutCRLF();
//        UART_PutCRLF();
//
```

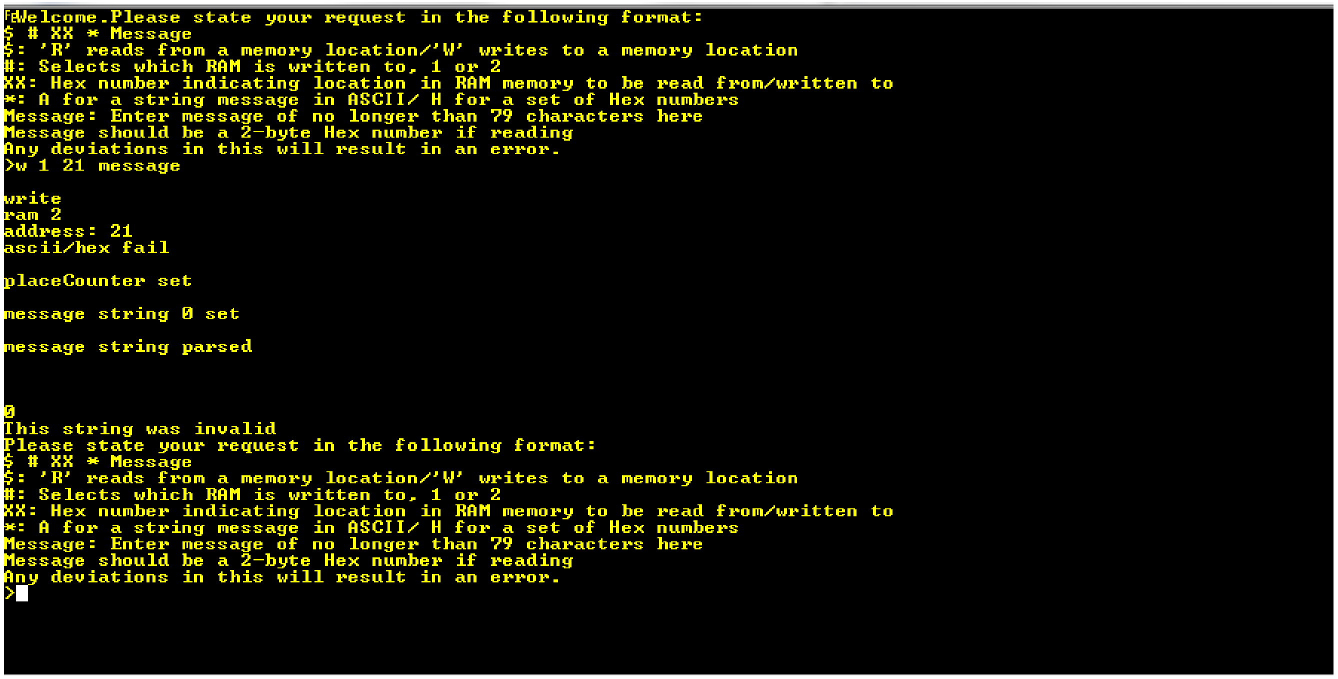
```
//        placeCounter=0;
//        parseCounter=0;
//
//        stringSize = strlen(messageString);
//        itoa(&sizeOfString, stringSize, 10);
//
//        UART_PutCRLF();
//        UART_PutString(&sizeOfString);
//        UART_PutCRLF();
//
//        //Populate Write buffer
//
//        //Address_Pointer =
//        //strcat(&address[0],&address[1]);
//
//        //txBuffer[0] = address[0];  // is this assignment correct?
//
//        for(i = 0; i <= stringSize; i++){
//                txBuffer[i+1] = messageString[i];
//
//        }//end for loop
//
//
//
//        /// add Condition for spaces
//}//end stringParser

//user instruction print function

void instructions(void)
{
//const char instructMsg[] = "Type in the form we like (I swear I'll fix
this)";
//"W # XX A Mary had a little lamb"
UART_CPutString("Please state your request in the following format: ");
UART_PutCRLF();

UART_CPutString("$ # XX * Message");
UART_PutCRLF();

UART_CPutString("$: 'R' reads from a memory location/'W' writes to a memory
location");
UART_PutCRLF();

UART_CPutString("#: Group 2 address of Ram to be read from/written to ");
UART_PutCRLF();

UART_CPutString("XX: Hex number indicating location in RAM memory to be read
from/written to");
UART_PutCRLF();

UART_CPutString("*: A for a string message in ASCII/ H for a set of Hex
numbers");
UART_PutCRLF();

UART_CPutString("Message: Enter message of no longer than 79 characters
here");
```

```
UART_PutCRLF();

UART_CPutString("Message should be a 2-byte Hex number if reading");
UART_PutCRLF();

UART_CPutString("Any deviations in this will result in an error.  ");
UART_PutCRLF();

UART_CPutString(">");

}//end instructions

//Read Ram function to read hard coded addresses
void readFromRam(void)
{
//VVV From sample program in datasheet VVV
/*
I2CHW_fReadBytes(SLAVE_ADDRESS, rxBuffer, BUFFER_SIZE, I2CHW_CompleteXfer);
// Wait until the data is read
while(!(I2CHW_bReadI2CStatus() & I2CHW_RD_COMPLETE));
// Clear Read Complete Status bit
I2CHW_ClrRdStatus();
*/
//need to parse the message into HEX and set to readSize


I2CHW_bWriteBytes(Ram1,0x00,1,I2CHW_NoStop);
I2CHW_fReadBytes(Ram1,rxBuffer, 8, I2CHW_RepStart);
while(!(I2CHW_bReadI2CStatus() & I2CHW_RD_COMPLETE));
I2CHW_ClrRdStatus();



}//end readFromRam
//write Ram function to read hard coded addresses
void writeToRam(void)
{

//Write to RAM

I2CHW_bWriteBytes(Ram1, txBuffer , 8, I2CHW_CompleteXfer);
while(!(I2CHW_bReadI2CStatus() & I2CHW_WR_COMPLETE));
I2CHW_ClrWrStatus();



}//end writeToRam

//char asciiToHex(char addressByte)
//char asciiToHex(char addressByte)
//{
//
//
//
//        char output;
//        BOOL lowerCase = FALSE;
//
```

```
//
//          UART_CPutString("staringAsciiToHex");
//          UART_PutCRLF();
//
//          if(addressByte>= 0x41 && addressByte<= 0x5a)
//          {
//
//                UART_CPutString("Lower case");
//                UART_PutCRLF();
//                rawString[0] = rawString[0]+ 0x20;
//          }
//
//        if (addressByte >= 0x30  && addressByte <= 0x39){
//
//                                UART_CPutString("integer");
//                                UART_PutCRLF();
//                      /// Add conditional if
//                        output = addressByte - 0x30;
//          }
//           if (addressByte >= 0x61  && addressByte <= 0x66){
//
//                                /// Add conditional if
//                                UART_CPutString("Upper case");
//                                UART_PutCRLF();
//                        output = addressByte - 0x51;
//          }
//
////        if (!(addressByte >= 0x30  && addressByte <= 0x39)&&
//!(addressByte >= 0x61  && addressByte <= 0x7a))
////        {
////
//////                          invaildFlag=TRUE;
//////                  UART_CPutString("Not valid Hex address");
//////                          UART_PutChar(0x07);
////        }
////
//
//        return output;
//
//}//end asciiToHex

/*
//end addressByte switch

//        if (lowerCase)
//        {
//                switch (output)
//                {
//                        case 'a':
//
//                                output = 0x61;
//
//                        break;
//
//                        case 'b':
//
//                                output = 0x62;
//
//                              break;
//
//                              case 'c':
//
//                                        output = 0x63;
//
//                              break;
//
//                              case 'd':
//
//                                        output = 0x64;
//
//                              break;
//
//                              case 'e':
//
//                                        output = 0x65;
//
//                              break;
//
//                              case 'f':
//
//                                        output = 0x66;
//
//                              break;
//
//                              default:

//end of switch output

// }//end if lowerCase
```

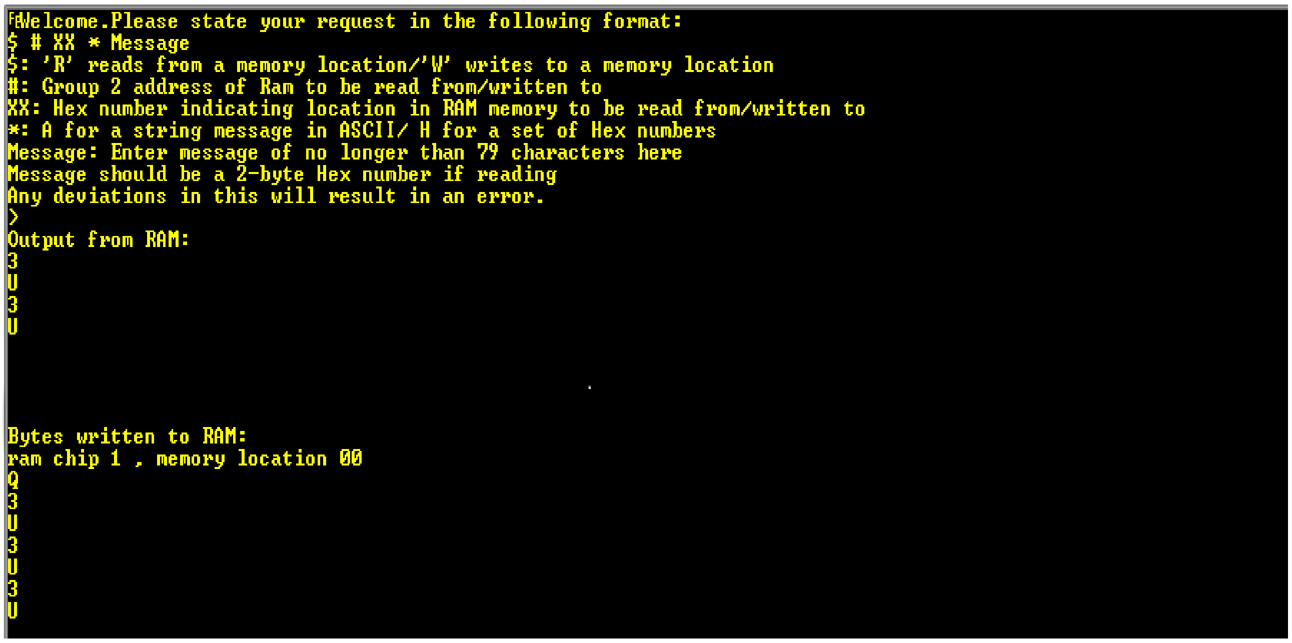
Testing:

After the program is written is should be built and uploaded to the PSoC. Turn the Psoc on and go to the Real Term console and look for the initial message. If your program was written properly then it should work as described however we encountered problems while trying to properly parse the string and were not able to isolate the proper value's to upload to our buffers for reading and writing. But if the program worked properly it should look something like this:



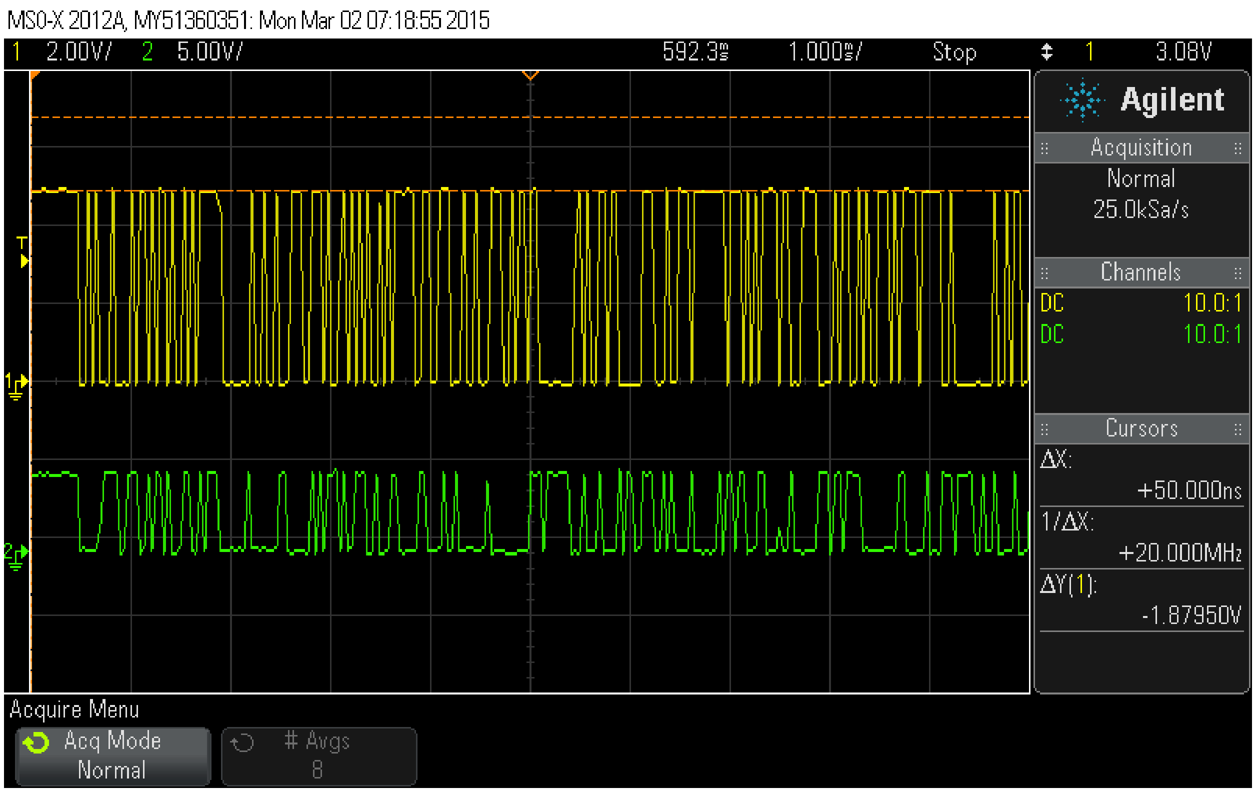
```
FΔWelcome.Please state your request in the following format:
$ # XX * Message
$: 'R' reads from a memory location/'W' writes to a memory location
#: Selects which RAM is written to, 1 or 2
XX: Hex number indicating location in RAM memory to be read from/written to
*: A for a string message in ASCII/ H for a set of Hex numbers
Message: Enter message of no longer than 79 characters here
Message should be a 2-byte Hex number if reading
Any deviations in this will result in an error.
>w 1 21 message

write
ram 2
address: 21
ascii/hex fail

placeCounter set

message string 0 set

message string parsed



0
This string was invalid
Please state your request in the following format:
$ # XX * Message
$: 'R' reads from a memory location/'W' writes to a memory location
#: Selects which RAM is written to, 1 or 2
XX: Hex number indicating location in RAM memory to be read from/written to
*: A for a string message in ASCII/ H for a set of Hex numbers
Message: Enter message of no longer than 79 characters here
Message should be a 2-byte Hex number if reading
Any deviations in this will result in an error.
>█
```

To read and write and test your ram chip run the uncommented code above. This code will populate a write buffer with the values that are written to ram. After that it reads those values back. Below is a screen shot of how the UART realterm console window should look if ran poperly.



```
FΔWelcome.Please state your request in the following format:
$ # XX * Message
$: 'R' reads from a memory location/'W' writes to a memory location
#: Group 2 address of Ram to be read from/written to
XX: Hex number indicating location in RAM memory to be read from/written to
*: A for a string message in ASCII/ H for a set of Hex numbers
Message: Enter message of no longer than 79 characters here
Message should be a 2-byte Hex number if reading
Any deviations in this will result in an error.
>
Output from RAM:
3
U
3
U




Bytes written to RAM:
ram chip 1 , memory location 00
Q
3
U
3
U
3
U
```

After checking the values on the console observe the the two I2C signals on the osciloscope with your two probes and see if the protocol acts as expected. Between each byte you should see the start, R/W and Acknoledge bits you should also noticed how signals and synchronized to the clock. Below are Osciloscope screen shots of how our signals should look however we had a lot of noise while redoing this part to get the screenshots:


MSO-X 2012A, MY51360351: Mon Mar 02 07:18:55 2015

1 2.00V/ 2 5.00V/ 592.3ms 1.000ms/ Stop 1 3.08V

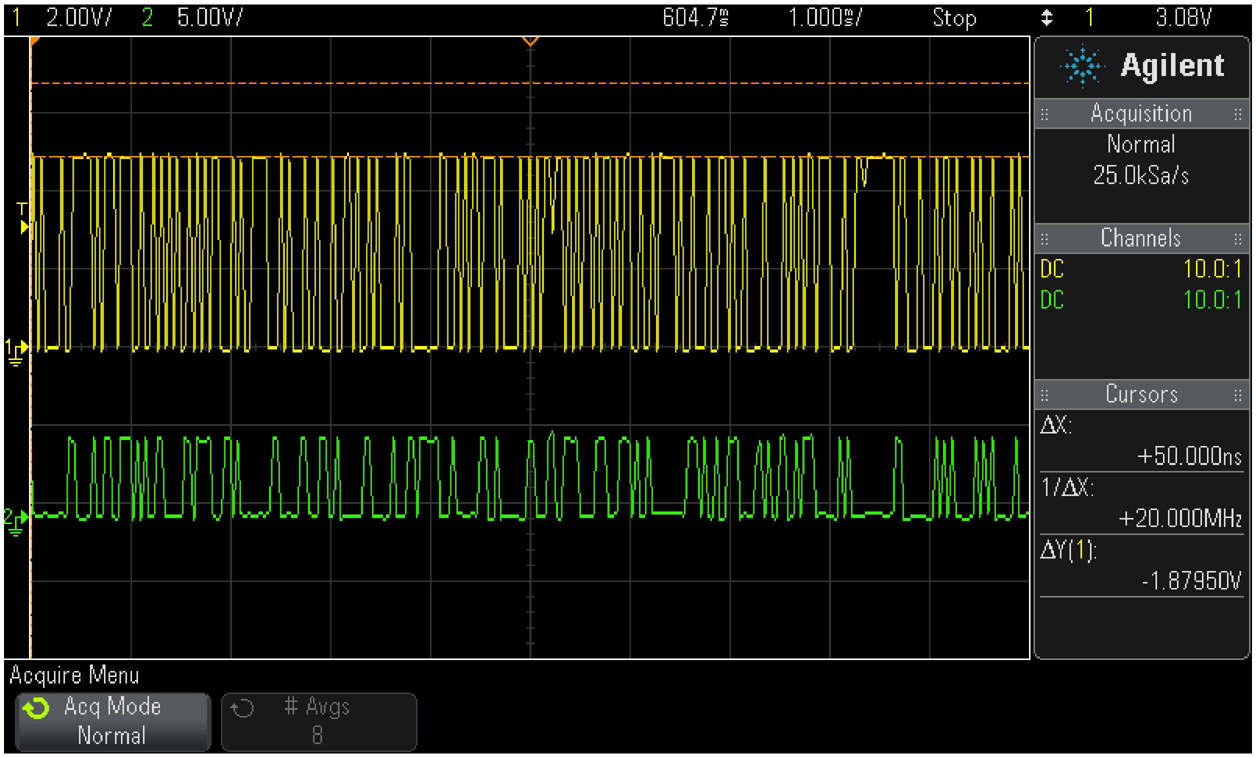
Acquisition: Normal, 25.0kSa/s

Channels: DC 10.0:1, DC 10.0:1

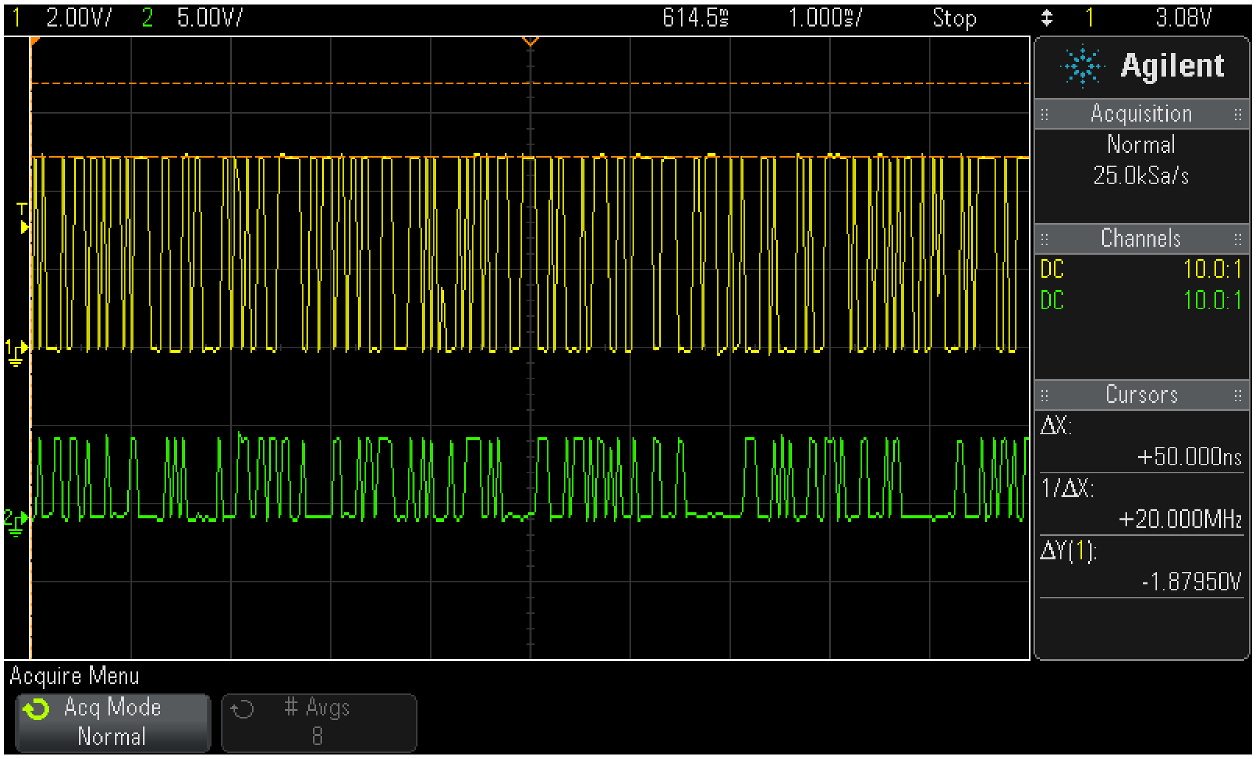
Cursors: ΔX: +50.000ns; 1/ΔX: +20.000MHz; ΔY(1): -1.87950V

Acquire Menu: Acq Mode Normal; # Avgs 8

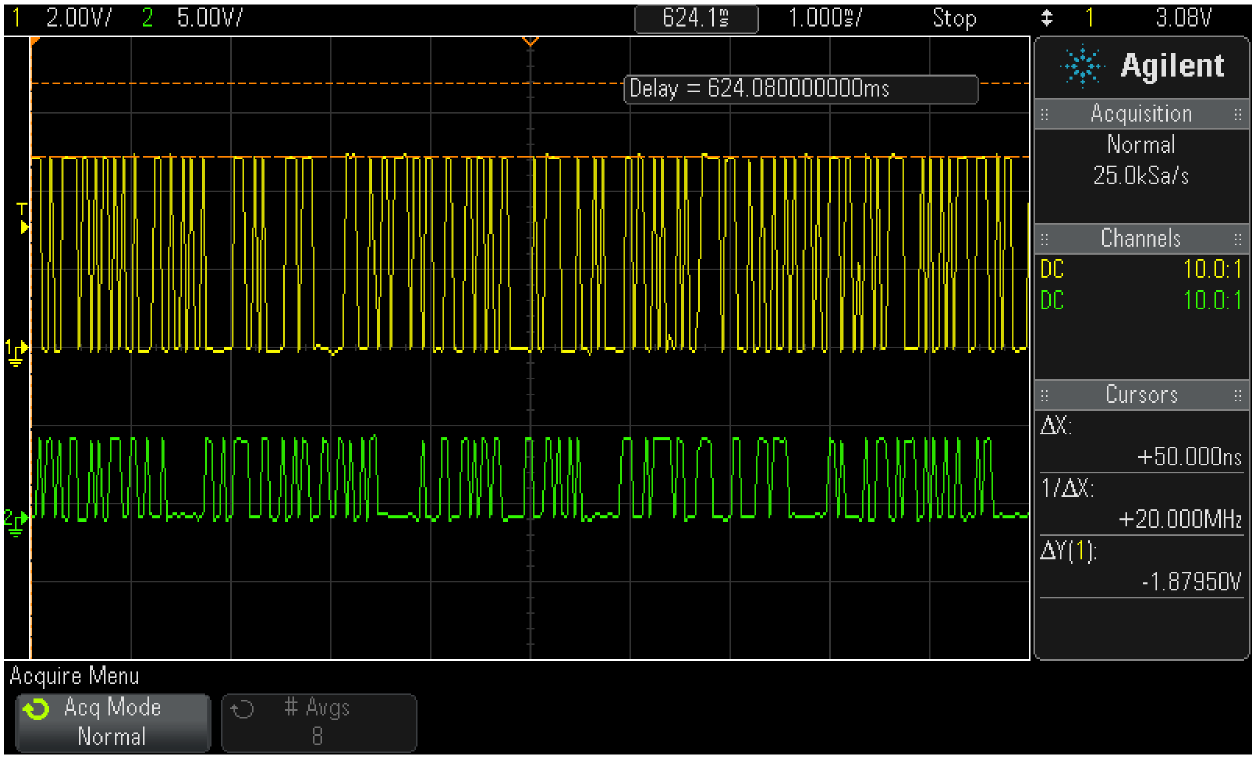
MSO-X 2012A, MY51360351: Mon Mar 02 07:19:29 2015



MSO-X 2012A, MY51360351: Mon Mar 02 07:19:41 2015



MSO-X 2012A, MY51360351: Mon Mar 02 07:19:54 2015



Below are functions used from the LCD & UART data sheet's:

I2CHW_fWrite

Description:

Sends a single-byte I^{2}C bus write and ACK. This function does not generate a start or stop condition. This routine should ONLY be called when a previous start and address has been generated on the I2C bus. It should only be used when I2C_BYTE_COMPL is set in the I2C_SCR register.

C Prototype:

```
BYTE I2CHW_fWrite( BYTE bData );
```

Assembler:

```
mov   A,[bRamData]              ; Load data to send to slave
lcall  I2CHW_fWrite              ; Initiate I2C write
```

Parameters:

bData: Byte to be sent to slave.

Return Value:

The return value is nonzero, if the slave acknowledged the master. The return value is zero, if the slave did not acknowledge the master. If the Slave failed to acknowledged the Master, the value of bStatus is 0xff.

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently, only the CUR_PP page pointer register is modified. The I2CHW interrupt is disabled if previously enabled.

I2CHW_bRead

Description:

Initiates a single-byte I^{2}C bus read and ACK phase. This function does not generate a start or stop condition. The fACK flag determines whether the slave is acknowledged upon receiving the data. This routine should ONLY be called when a previous start and address has been generated on the I2C bus. It should only be used when I2C_BYTE_COMPL is set in the I2C_SCR register. If fACK is set, it should be followed by next I2CHW_bRead call. To finish the read transaction Master should call this function with I2CHW_NAKslave parameter.

C Prototype:

```
BYTE I2CHW_bRead( BYTE fACK );
```

Assembler:

```
mov   A,I2CHW_ACKslave         ; Set flag to ACK slave
lcall  I2CHW_bRead              ; Read single byte from slave
                               ; Return data is in reg A
```

Parameters:

fACK: Set to I2CHW_ACKslave if master should ACK the slave after receiving the data; otherwise, flag should be set to I2CHW_NAKslave. In general, the ACK from master means request for the next data byte from the Slave. If set to I2CHW_ACKslave, the master after receiving current data byte and ACKing will immediately clock in the next byte from the slave. This next byte will be returned the next time I2CHW_bRead() is called. If set to I2CHW_NAKslave, the master will only NAK the current byte and will not clock in a the next byte of data.

Return Value:

Byte received from slave.

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently, only the CUR_PP page pointer register is modified. The I2CHW interrupt is disabled if previously enabled.

UART_Start

Description:

Sets the parity and enables the UART receiver and transmitter. When enabled, data can be received and transmitted.

C Prototype:

```
void  UART_Start(BYTE bParitySetting)
```

Assembly:

```
mov   A, UART_PARITY_NONE
lcall  UART_Start
```

Parameters:

bParitySetting: One byte that specifies the transmit parity. Symbolic names are given in C and assembly, and their associated values are listed in the following table:

TX Parity	Value
UART_PARITY_NONE	0x00
UART_PARITY_EVEN	0x02
UART_PARITY_ODD	0x06

Return Value:

None

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_CPutString

Description:

Sends constant (ROM), null terminated string out the UART TX port.

C Prototype:

```
void UART_CPutString(const char * szROMString)
```

Assembler:

```
mov  A,>szRomString   ; Load MSB part of pointer to ROM based null
                      ; terminated string.
mov  X,<szRomString   ; Load LSB part of pointer to ROM based null
                      ; terminated string.
lcall UART_CPutString  ; Call function to send constant string out
                      ; UART TX
```

Parameters:

const char * szROMString: Pointer to string to be sent to the UART. MSB of string pointer is passed in the Accumulator and LSB of the pointer is passed in the X register.

Return Value:

None

Side Effects:

Program flow stays in this function, until the last character is loaded into the UART transmit buffer. The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_PutChar

Description:

Writes a single character to the UART TX port when the port buffer is empty.

C Prototype:

```
void UART_PutChar(CHAR cData)
```

Assembler:

```
mov  A,0x33             ; Load ASCII character "3" in A
lcall UART_PutChar       ; Call function to send single character to
                        ; UART TX port.
```

Parameters:

CHAR cData: The character to be sent to the UART TX port. Data is passed in the Accumulator.

Return Value:

None

Side Effects:

Program flow stays in this function until the data can be written to the UART TX buffer. The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_cReadChar

Description:

Reads UART RX port immediately if data is not available or an error condition exists, or zero is returned. Otherwise, the character is read and returned.

C Prototype:

```
CHAR UART_cReadChar(void)
```

Assembler:

```
lcall UART_cReadChar           ; Call function to read a character
cmp  A,0x00                   ; Check for error
jz   ProcessError             ; If error, Process the error condition
mov  [CharBuffer],A           ; Store retrieved character in buffer
```

Parameters:

None

Return Value:

CHAR bData: Character read from UART RX port. ASCII characters from 1 to 255 are valid. A returned zero signifies an error condition or no data available.

Side Effects:

Function only accepts characters from 1 to 255 as valid. A 0x00 (null) character is detected as an error condition. The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.