

# SliceKit GPIO Example Applications

---

REV A

Publication Date: 2012/10/15  
XMOS © 2012, All Rights Reserved.



## Table of Contents

<b>1 Overview</b>	<b>3</b>
<b>2 Evaluation Platforms</b>	<b>4</b>
2.1 Recommended Hardware	4
2.2 Example Applications	4
2.2.1 app_slicekit_simple_demo	4
2.2.2 app_slicekit_com_demo	4
<b>3 Programming Guide</b>	<b>6</b>
3.1 Simple Demo	6
3.1.1 Structure	6
3.1.2 API	6
3.1.3 Usage and Implementation	7
3.2 COM Port Demo	9
3.2.1 Structure	9
3.2.2 API	9
3.2.3 Usage and Implementation	10

# 1 Overview

---

This document covers two example applications that demonstrate various features of the XA-SK-GPIO Slice Card including the ADC, LEDs, UART connector and buttons, the I2C master xSOFTip component, as well as various basic features of the xCORE processor.

Low level details of how the application is implemented are covered here, with information about how to modify and experiment with them.

For getting started instructions for both applications, please refer to their respective Quick Start guides.

## 2 Evaluation Platforms

---

### IN THIS CHAPTER

- ▶ Recommended Hardware
  - ▶ Example Applications
- 

### 2.1 Recommended Hardware

This application may be evaluated using the SliceKit Modular Development Platform, available from digikey. Required board SKUs are:

- ▶ XP-SKC-L2 (SliceKit L2 Core Board) plus XA-SK-GPIO plus XA-SK-XTAG2 (SliceKit XTAG adaptor) plus XTAG2 (debug adaptor),

### 2.2 Example Applications

#### 2.2.1 app\_sliceKit\_simple\_demo

This application has the following features:

- ▶ module\_i2c\_master from the xSOFT ip library is used to access the external ADC, which is equipped with an external linearised thermistor circuit for temperature sensing.
- ▶ simple code to print the recorded temperature to the XDE debug console on the press of one of the Slice Card buttons
- ▶ simple code to cycle through the 4 LEDs each time the other button is pressed.
- ▶ demonstrates use of XC select statements for handling multiple concurrent inputs
- ▶ demonstrates basic usage of XCore ports

#### 2.2.2 app\_sliceKit\_com\_demo

This application extends the simple demo to provides the following functionality (all options are dynamically reconfigurable via the APIs for app\_sliceKit\_com\_demo application):

- ▶
  - ▶ Baud Rate: 150 to 115200 bps
  - ▶ Parity: None, Mark, Space, Odd, Even
  - ▶ Stop Bits: 1,2
  - ▶ Data Length: 1 to 30 bits (Max 30 bits assumes 1 stop bit and no parity)

- ▶ Cycles Through LEDs on button Press
- ▶ Displays temperature value and button press events on the terminal console of a host PC via the UART

## 3 Programming Guide

---

### IN THIS CHAPTER

- ▶ Simple Demo
  - ▶ COM Port Demo
- 

## 3.1 Simple Demo

### 3.1.1 Structure

All of the files required for operation are located in the `app_sk_gpio_simple_demo/src` directory. The files that are need to be included for use of this component in an application are:

---

File	Description
<code>common.h</code>	<b>Header file for API interfaces and Look up tables for thermistor.</b>
<code>main.xc</code>	Main file which implements the demo functionality

---

### 3.1.2 API

```
void app_manager(chanend c_uartTX,  
                chanend c_chanRX,  
                chanend c_process,  
                chanend c_end)
```

Polling uart RX and push button switches and send received commands to `process_data` thread.

This function has the following parameters:

<code>c_uartTX</code>	Channel to Uart TX Thread
<code>c_chanRX</code>	Channel to Uart RX Thread
<code>c_process</code>	Channel to process data Thread
<code>c_end</code>	Channel to read data from process thread

```
int linear_interpolation(int adc_value)
```

Calculates temperatue based on linear interpolation.

This function has the following parameters:

<code>adc_value</code>	int value read from ADC
------------------------	-------------------------

This function returns:

Returns linear interpolated Temperature value

```
int read_adc_value()
```

Read ADC value using I2C.

This function returns:

Returns ADC value

### 3.1.3 Usage and Implementation

The port declaration for the LEDs, Buttons and I2C are declared as below. LEDs and Buttons use 4 bit ports and I2C uses 1 bit port for SCL(I2c Clock) and SDA (I2C data).

```
on stdcore[CORE_NUM]: out port p_led=XS1_PORT_4A;
on stdcore[CORE_NUM]: port p_PORT_BUT_1=XS1_PORT_4C;
struct r_i2c i2cOne = {
    XS1_PORT_1F,
    XS1_PORT_1B,
    1000
};
```

The app\_manager API writes the configuration settings information to the ADC as shows below.

```
i2c_master_write_reg(0x28, 0x00, data, 1, i2cOne); //Write configuration information to
↳ ADC
```

The select statement in the app\_manager API selects one of the two cases in it, checks if there is IO event or timer event. This statement monitors both the events and executes which ever event is occurred first. The select statement in the application is listed below. The statement checks if there is button press or not. If there is button press then it looks if the button state is same even after 200msec. If the buton state is same then it recognises as an valid push.

```
select
{
    case button => p_PORT_BUT_1 when pinsneq(button_press_1):> button_press_1: //
↳ checks if any button is pressed
        button=0;
        break;

    case !button => t when timerafter(time+2000000):>time: //waits for 200ms and
↳ checks if the same button is pressed or not
        p_PORT_BUT_1:> button_press_2;
        if(button_press_1==button_press_2)
            if(button_press_1 == BUTTON_PRESS_VALUE) //Button 1 is pressed
            {
                printstrln("Button 1 Pressed");
                p_led<:(led_value);
                led_value=led_value<<1;
                led_value|=0x01;
                led_value=led_value & 0x0F;
```

```

        if(led_value == 15)
        {
            led_value=0x0E;
        }
    }
    if(button_press_1 == BUTTON_PRESS_VALUE-1) //Button 2 is pressed
    {
        data1[0]=0;data1[1]=0;
        i2c_master_rx(0x28, data1, 2, i2cOne); //Read ADC value using I2C
        ↪ read
        printf("Reading Temperature value....");
        data1[0]=data1[0]&0x0F;
        adc_value=(data1[0]<<6)|(data1[1]>>2);
        printf("Temperature is :");
        printf("%d",linear_interpolation(adc_value));
    }

    button=1;
    break;
}

```

After recognising the valid push then it checks if Button 1 is pressed or Button 2 is pressed. If Button 1 is pressed then, the application reads the status of LEDs and shift the position of the LEDs to left by 1. If Button 2 is pressed, then the application reads the contents of ADC register using I2C read instruction and input the ADC value to linear interpolation function as shown below.

```

int linear_interpolation(int adc_value)
{
    int i=0,x1,y1,x2,y2,temper;
    while(adc_value<TEMPERATURE_LUT[i][1])
    {
        i++;
    }
    //::Formula start
    //Calculating Linear interpolation using the formula  $y=y_1+(x-x_1)*(y_2-y_1)/(x_2-x_1)$ 
    //::Formula
    x1=TEMPERATURE_LUT[i-1][1];
    y1=TEMPERATURE_LUT[i-1][0];
    x2=TEMPERATURE_LUT[i][1];
    y2=TEMPERATURE_LUT[i][0];
    temper=y1+(((adc_value-x1)*(y2-y1))/(x2-x1));
    return temper;//Return Temperature value
}

```

The linear interpolation function calculates the linear interpolation value using the following formula and returns the temperature value from temperature look up table.

```
//Calculating Linear interpolation using the formula  $y=y_1+(x-x_1)*(y_2-y_1)/(x_2-x_1)$ 
```

```

int TEMPERATURE_LUT[][2]= //Temperature Look up table
{
    {-10,845},{-5,808},{0,765},{5,718},{10,668},{15,614},{20,559},{25,504},{30,450},{35,395},
    ↪
    {40,352},{45,308},{50,269},{55,233},{60,202}
};

```



## 3.2 COM Port Demo

### 3.2.1 Structure

All of the files required for operation are located in the `app_sliceKit_simple_demo/src` directory. The files that are need to be included for use of this component in an application are:

File	Description
<code>common.h</code>	<b>Header file for API interfaces and Look up tables for thermistor. FIXME - what about the uart</b>
<code>main.xc</code>	Main file which implements the demo functionality

### 3.2.2 API

```
void app_manager(chanend c_uartTX,
                 chanend c_chanRX,
                 chanend c_process,
                 chanend c_end)
```

Polling uart RX and push button switches and send received commands to `process_data` thread.

This function has the following parameters:

`c_uartTX`      Channel to Uart TX Thread  
`c_chanRX`      Channel to Uart RX Thread  
`c_process`      Channel to process data Thread  
`c_end`          Channel to read data from process thread

```
void process_data(chanend c_process, chanend c_end)
```

process received data to see if received data is valid command or not Polling switches to see for button press

This function has the following parameters:

`c_process`      Channel to receive data from app manager Thread  
`c_end`          Channel to communicate to app manager thread

```
void uart_tx_string(chanend c_uartTX, unsigned char message[100])
```

Transmits byte by byte to the UART TX thread for an input string.

This function has the following parameters:

`c_uartTX`      Channel to receive data from app Uart TX Thread  
`message`        Buffer to store array of characters

```
int linear_interpolation(int adc_value)
```

Calculates temperatue based on linear interpolation.

This function has the following parameters:

adc\_value     int value read from ADC

This function returns:

Returns linear interpolated Temperature value

```
int read_adc_value()
```

Read ADC value using I2C.

This function returns:

Returns ADC value

### 3.2.3 Usage and Implementation

The port declaration for the LEDs, Buttons, I2C and UART are declared as below. LEDs and Buttons uses 4 bit ports, UART uses 1 bit ports both for Transmit and Receive and I2C uses 1 bit port for SCL(I2c Clock) and SDA (I2C data).

```
#define CORE_NUM 1
#define BUTTON_PRESS_VALUE 2
on stdcore[CORE_NUM] : buffered in port:1 p_rx = XS1_PORT_1G;
on stdcore[CORE_NUM] : out port p_tx = XS1_PORT_1C;
on stdcore[CORE_NUM]: port p_led=XS1_PORT_4A;
on stdcore[CORE_NUM]: port p_button1=XS1_PORT_4C;
struct r_i2c i2c0ne = {
    XS1_PORT_1F,
    XS1_PORT_1B,
    1000
};
```

The app\_manager API writes the configuration settings information to the ADC as shows below.

```
i2c_master_write_reg(0x28, 0x00, i2c_register, 1, i2c0ne); //Configure ADC by writing the
↳ settings to register
```

The select statement in the app\_manager API selects one of the three cases in it, checks if there is IO event or timer event or any event on the Uart Receive pin. This statement monitors all the events and executes which ever event is occurred first. The select statement in the applciation is listed below. The statement checks if there is button press or availability of data on the Uart Receive pin. If there is button press then it looks if the button state is same as even after 200msec. If the buton state is same then it recognises as a valid push. If there is data on the Uart Receive pin the it echoes the data back to the uart Transmit pin until > character is received in the input data.

```
select
```

```

{
    case c_end:>data:
        c_end:>data;
        if(data == BUTTON_1) //Cycle LEDs on button 1 press
        {
            printstrln("Button 1 Pressed");
            p_led<:(led_value);
            led_value=led_value<<1;
            if(led_value == 16) //If LED value is 16 then
                ↳ assigns LED value to 1 which cycles LEDs by
                ↳ button press
            {
                led_value=0x01;
            }
        }
        if(data == BUTTON_2) //Displays Temperature on console if
            ↳ Button 2 is pressed
        {
            adc_value=read_adc_value();
            data_arr[0]=(linear_interpolation(adc_value));
            printstr("Temperature is :");
            printint(linear_interpolation(adc_value));
            printstrln(" C");
        }
        break;
    case uart_rx_get_byte_byref(c_uartRX, rxState, buffer):
//:::Command start
        if(buffer == '>') //IUF received data is '>' character
            ↳ then expects cmd to endter into command mode
        {
            j=0;
            uart_rx_get_byte_byref(c_uartRX, rxState, buffer)
                ↳ ;
            cmd_rcvbuffer[j]=buffer;
            if((cmd_rcvbuffer[j] == 'C' )|| (cmd_rcvbuffer[j]
                ↳ =='c')) //Checks if received data is 'C' or
                ↳ 'c'
            {
                j++;
                uart_rx_get_byte_byref(c_uartRX, rxState,
                    ↳ buffer);
                cmd_rcvbuffer[j]=buffer;

                if((cmd_rcvbuffer[j] == 'm' )|| (
                    ↳ cmd_rcvbuffer[j] =='M')) //Checks if
                    ↳ received data is 'M' or 'm'
                {
                    j++;
                    uart_rx_get_byte_byref(c_uartRX,
                        ↳ rxState, buffer);
                    cmd_rcvbuffer[j]=buffer;
                    if((cmd_rcvbuffer[j] == 'D' )|| (
                        ↳ cmd_rcvbuffer[j] =='d'))//
                        ↳ Checks if received data is '
                        ↳ D' or 'd'
                    {
                        uart_tx_send_byte(
                            ↳ c_uartTX, '\r');
                        uart_tx_send_byte(
                            ↳ c_uartTX, '\n');
                    }
                }
            }
        }
    }
}

```

```

        uart_tx_string(c_uartTX,
            ↳ CONSOLE_MESSAGES[0])
            ↳ ;
        COMMAND_MODE=1; //
            ↳ activates command
            ↳ mode as received
            ↳ data is '>cmd'
        uart_tx_send_byte(
            ↳ c_uartTX, '\r');
        uart_tx_send_byte(
            ↳ c_uartTX, '\n');
        uart_tx_send_byte(
            ↳ c_uartTX, '>'); //
            ↳ displays '>' if
            ↳ command mode is
            ↳ activated
    }
    else
    {
        uart_tx_send_byte(
            ↳ c_uartTX, '>');
        for(int i=0;i<3;i++)
            uart_tx_send_byte
                ↳ (c_uartTX,
                ↳ cmd_rcvbuffer
                ↳ [i]); // if
                ↳ received dta
                ↳ is not 'c'
                ↳ displays
                ↳ back the
                ↳ received
                ↳ data
    }
    }
    else
    {
        uart_tx_send_byte(c_uartTX, '>');
            ↳ //if received data is not '
            ↳ m' displays the received
            ↳ data
        for(int i=0;i<2;i++)
            uart_tx_send_byte(
                ↳ c_uartTX,
                ↳ cmd_rcvbuffer[i]);
    }
    }
    else
    {
        uart_tx_send_byte(c_uartTX, '>');
        uart_tx_send_byte(c_uartTX, cmd_rcvbuffer
            ↳ [j]);
        j=0;
    }
    }
    else
    {
        uart_tx_send_byte(c_uartTX,buffer); //Echoes back
            ↳ the input characters if not in command mode
    }
}
//:::Command
while(COMMAND_MODE) //Command mode activated

```



```

cmd_rcvbuffer
↳ [inc
↳ ]='0';
↳
        j=0;
    }
    break;
//::Send
case c_end:>data:
    if(data!=EXIT && data!=
↳ INVALID )
    {
        uart_tx_string(
↳ c_uartTX,
↳ CONSOLE_MESSAGES
↳ [3]); //
↳ Displays
↳ CCommand
↳ Executed
↳ Message on
↳ Uart
    }
switch(data)
{
    case EXIT: //Exit
↳ from
↳ command mode
        COMMAND_MODE
↳ =0;
        skip=0;
        uart_tx_string
↳ (
↳ c_uartTX
↳ ,
↳ CONSOLE_MESSAGES
↳ [1])
↳ ;
        uart_tx_string
↳ (
↳ c_uartTX
↳ ,
↳ CONSOLE_MESSAGES
↳ [13])
↳ ;
        break;
    case SET_LED_1:
↳ //Read port
↳ Value and
↳ Set LED 1 ON
        p_led:>
↳ data
↳ ;
        data=data
↳ | 0
↳ xE;
        p_led<:
↳ data
↳ ;
        break;

```

```
case CLEAR_LED_1
↳ //Read port
↳ Value and
↳ Set LED 1
↳ OFF
    p_led:>
↳ data
↳ ;
    p_led<:
↳ data
↳ &0x1
↳ ;
    break;

case SET_LED_2:
↳ //Read port
↳ Value and
↳ Set LED 2 ON
    p_led:>
↳ data
↳ ;
    p_led<:
↳ data
↳ | 0
↳ xD;
    break;

case CLEAR_LED_2:
↳ //Read port
↳ Value and
↳ Set LED 2
↳ OFF
    p_led:>
↳ data
↳ ;
    p_led<:
↳ data
↳ &0x2
↳ ;
    break;

case SET_LED_3:
↳ //Read port
↳ Value and
↳ Set LED 3 ON
    p_led:>
↳ data
↳ ;
    p_led<:
↳ data
↳ | 0
↳ xB;
    break;

case CLEAR_LED_3:
↳ //Read port
↳ Value and
↳ Set LED 3
↳ OFF
```

```
        p_led:>
            ↳ data
            ↳ ;
        p_led<:
            ↳ data
            ↳ &0x4
            ↳ ;
        break;

case SET_LED_4:
    ↳ //Read port
    ↳ Value and
    ↳ Set LED 4 ON
        p_led:>
            ↳ data
            ↳ ;
        p_led<:
            ↳ data
            ↳ | 0
            ↳ x7;
        break;

case CLEAR_LED_4:
    ↳ //Read port
    ↳ Value and
    ↳ Set LED 4
    ↳ OFF
        p_led:>
            ↳ data
            ↳ ;
        p_led<:
            ↳ data
            ↳ &0x8
            ↳ ;

        break;

case CLEAR_ALL:
    ↳ //sets all
    ↳ four LEDs
    ↳ OFF
        p_led<:0
            ↳ xF;
        break;

case SET_ALL: //
    ↳ sets all
    ↳ four LEDs ON
        p_led<:0
            ↳ x0;
        break;

case
    ↳ BUTTON_PRESSED
    ↳ : //Checks
    ↳ if button is
    ↳ pressed
        c_end:>
            ↳ button
            ↳ ;
```



```

        if(button
            ↳ ==
            ↳ BUTTON_1
            ↳ ) //
            ↳ Prints
            ↳
            ↳ Button
            ↳ 1
            ↳ is
            ↳ pressed
            ↳ on
            ↳ the
            ↳ Uart
        {
            console_message(
                ↳ [4][9]='1'
            )
            uart_tx_string(
                ↳ (
                ↳ c_uartTX
                ↳ ,
                ↳ CONSOLE_MESSAG
                ↳ [4])
                ↳ ;
            )
            button1_press
            ↳ =1;
        }
        if(button
            ↳ ==
            ↳ BUTTON_2
            ↳ ) //
            ↳ Prints
            ↳
            ↳ Button
            ↳ 2
            ↳ is
            ↳ pressed
            ↳ on
            ↳ Uart
        {
            console_message(
                ↳ [4][9]='2'
            )
            uart_tx_string(
                ↳ (
                ↳ c_uartTX
                ↳ ,
                ↳ CONSOLE_MESSAG
                ↳ [4])
                ↳ ;
            )
            button2_press
            ↳ =1;
        }
        break;
    case HELP: //
        ↳ Displays
        ↳ help

```

```
↳ messages on
↳ Uart
    uart_tx_string
    ↳ (
    ↳ c_uartTX
    ↳ ,
    ↳ CONSOLE_MESSAGES
    ↳ [14])
    ↳ ;
    uart_tx_string
    ↳ (
    ↳ c_uartTX
    ↳ ,
    ↳ CONSOLE_MESSAGES
    ↳ [7])
    ↳ ;
    uart_tx_string
    ↳ (
    ↳ c_uartTX
    ↳ ,
    ↳ CONSOLE_MESSAGES
    ↳ [8])
    ↳ ;
    uart_tx_string
    ↳ (
    ↳ c_uartTX
    ↳ ,
    ↳ CONSOLE_MESSAGES
    ↳ [9])
    ↳ ;
    uart_tx_string
    ↳ (
    ↳ c_uartTX
    ↳ ,
    ↳ CONSOLE_MESSAGES
    ↳ [10])
    ↳ ;
    uart_tx_string
    ↳ (
    ↳ c_uartTX
    ↳ ,
    ↳ CONSOLE_MESSAGES
    ↳ [15])
    ↳ ;
    uart_tx_string
    ↳ (
    ↳ c_uartTX
    ↳ ,
    ↳ CONSOLE_MESSAGES
    ↳ [11])
    ↳ ;
    uart_tx_send_byte
    ↳ (
    ↳ c_uartTX
    ↳ , '\
    ↳ r');
    uart_tx_send_byte
    ↳ (
    ↳ c_uartTX
    ↳ , '\
    ↳ n');
```

```
        break;
    case READ_ADC: //
        ↳ Displays
        ↳ temperature
        ↳ value on the
        ↳ Uart
            adc_value
                ↳ =
                ↳ read_adc_value
                ↳ ();
            data_arr
                ↳ [0]=(
                ↳ linear_interpolat
                ↳ (
                ↳ adc_value
                ↳ ));
            uart_tx_string
                ↳ (
                ↳ c_uartTX
                ↳ ,
                ↳ CONSOLE_MESSAGES
                ↳ [12])
                ↳ ;
            uart_tx_send_byte
                ↳ (
                ↳ c_uartTX
                ↳ , (
                ↳ data_arr
                ↳ [0]/10)
                ↳ +'0')
                ↳ ;
            uart_tx_send_byte
                ↳ (
                ↳ c_uartTX
                ↳ , (
                ↳ data_arr
                ↳ [0]%10)
                ↳ +'0')
                ↳ ;
            uart_tx_send_byte
                ↳ (
                ↳ c_uartTX
                ↳ ,
                ↳ 32);
            uart_tx_send_byte
                ↳ (
                ↳ c_uartTX
                ↳ , 'C
                ↳ ');
        break;
    case INVALID: //
        ↳ Displays
        ↳ command
        ↳ input is
        ↳ invalid
        ↳ command on
        ↳ the Uart
            uart_tx_string
                ↳ (
                ↳ c_uartTX
                ↳ ,
```

```

        ↪ console_messages
        ↪ [2])
        ↪ ;
        break;
    case CHK_BUTTONS:
        ↪ //Checks if
        ↪ button are
        ↪ pressed and
        ↪ displays on
        ↪ the Uart
        if(
            ↪ button1_press
            ↪ )
        {
            console_messages
            ↪ [4][9]='1'
            ↪
            uart_tx_string
            ↪ (
            ↪ c_uartTX
            ↪ ,
            ↪ console_m
            ↪ [4])
            ↪ ;
            ↪ //
            ↪ Displays
            ↪
            ↪ Button
            ↪
            ↪ 1
            ↪
            ↪ is
            ↪
            ↪ pressed
            ↪
        }
        if(
            ↪ button2_press
            ↪ )
        {
            console_messages
            ↪ [4][9]='2'
            ↪
            uart_tx_string
            ↪ (
            ↪ c_uartTX
            ↪ ,
            ↪ console_m
            ↪ [4])
            ↪ ;
            ↪ //
            ↪ Displays
            ↪
            ↪ Button
            ↪
            ↪ 2
            ↪
            ↪ is
            ↪
        }
    }
}

```

```

    }
    if( !
        ↳ button1_press
        ↳ &&
        ↳ !
        ↳ button2_press
        ↳ )
    {
        uart_tx_string
        ↳ (
        ↳ c_uartTX
        ↳ ,
        ↳ CONSOLE_M
        ↳ [5])
        ↳ ;
        ↳ //
        ↳ Displays
        ↳ No
        ↳ Buttons
        ↳ are
        ↳ pressed
    }
    button1_press
        ↳ =0;
    button2_press
        ↳ =0;
    break;
}
if(data != EXIT) //Exits
↳ from command mode
{
    uart_tx_send_byte
        ↳ (c_uartTX,
        ↳ '\r');
    uart_tx_send_byte
        ↳ (c_uartTX,
        ↳ '\n');
    uart_tx_send_byte
        ↳ (c_uartTX,
        ↳ '>');
}
break;
} //select

//:::State
                                } //skip
                                j=0;
                                } //command mode
                                break;
} //main select

```

If the received data is > character the it waits to see if the next received successive bytes are c, m and d. If the successive received data is >cmd then the application activates command mode otherwise the data is echoed back to the Uart Transmit pin. The part of code which explains about the command mode is as blow.

```

if(buffer == '>') //IUF received data is '>' character then expects cmd to endter into
↳ command mode
{
    j=0;
    uart_rx_get_byte_byref(c_uartRX, rxState, buffer);
    cmd_rcvbuffer[j]=buffer;
    if((cmd_rcvbuffer[j] == 'C' )|| (cmd_rcvbuffer[j] == 'c')) //Checks if received
↳ data is 'C' or 'c'
    {
        j++;
        uart_rx_get_byte_byref(c_uartRX, rxState, buffer);
        cmd_rcvbuffer[j]=buffer;

        if((cmd_rcvbuffer[j] == 'm' )|| (cmd_rcvbuffer[j] == 'M')) //Checks if
↳ received data is 'M' or 'm'
        {
            j++;
            uart_rx_get_byte_byref(c_uartRX, rxState, buffer);
            cmd_rcvbuffer[j]=buffer;
            if((cmd_rcvbuffer[j] == 'D' )|| (cmd_rcvbuffer[j] == 'd'))//Checks
↳ if received data is 'D' or 'd'
            {
                uart_tx_send_byte(c_uartTX, '\r');
                uart_tx_send_byte(c_uartTX, '\n');
                uart_tx_string(c_uartTX,CONSOLE_MESSAGES[0]);
                COMMAND_MODE=1; //activates command mode as received data
↳ is '>cmd'
                uart_tx_send_byte(c_uartTX, '\r');
                uart_tx_send_byte(c_uartTX, '\n');
                uart_tx_send_byte(c_uartTX, '>'); //displays '>' if
↳ command mode is activated
            }
            else
            {
                uart_tx_send_byte(c_uartTX, '>');
                for(int i=0;i<3;i++)
                    uart_tx_send_byte(c_uartTX, cmd_rcvbuffer[i]); //
↳ if received dta is not 'c' displays back
↳ the received data
            }
        }
        else
        {
            uart_tx_send_byte(c_uartTX, '>'); //if received data is not 'm'
↳ displays the received data
            for(int i=0;i<2;i++)
                uart_tx_send_byte(c_uartTX, cmd_rcvbuffer[i]);
        }
    }
    else
    {
        uart_tx_send_byte(c_uartTX, '>');
        uart_tx_send_byte(c_uartTX, cmd_rcvbuffer[j]);
        j=0;
    }
}

```

```

}
else
{
    uart_tx_send_byte(c_uartTX,buffer); //Echoes back the input characters if not in
    ↪ command mode
}

```

After the command mode is active the application receives all the input commands and send to the process\_data API using a channel. The part of the code is shown below.

```

select
{
    case uart_rx_get_byte_byref(c_uartRX, rxState, buffer):
        cmd_rcvbuffer[j]=buffer;
        if(cmd_rcvbuffer[j++] == '\r')
        {
            skip=0;
            j=0;
            while(cmd_rcvbuffer[j] != '\r')
            {
                c_process<:cmd_rcvbuffer[j]; //received valid command and
                ↪ send the command to the process_data thread
                uart_tx_send_byte(c_uartTX, cmd_rcvbuffer[j]);
                j++;
            }
            cmd_rcvbuffer[j]='\0';
            c_process<:cmd_rcvbuffer[j];
            for(int inc=0;inc<20;inc++) //Clears the command buffer
                cmd_rcvbuffer[inc]='0';
            j=0;
        }
        break;
}

```

The process\_data thread checks if any button is pressed or checks if there is any command from app\_manager thread. If there is button press then the thread sends instructions to the app\_manager thread about the button or if command is received, then it send instructions about the command received. The details in the process\_data thread is as shown below.

Process\_data thread send instructions to the app\_manager thread about the command received. The app\_manager thread then implementys the state machine according to the instructions received from the process\_data thread. The state machine of app\_manager thread is as below.



Copyright © 2012, All Rights Reserved.

---

Xmos Ltd. is the owner or licensee of this design, code, or Information (collectively, the "Information") and is providing it to you "AS IS" with no warranty of any kind, express or implied and shall have no liability in relation to its use. Xmos Ltd. makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.

XMOS and the XMOS logo are registered trademarks of Xmos Ltd. in the United Kingdom and other countries, and may not be used without written permission. All other trademarks are property of their respective owners. Where those designations appear in this book, and XMOS was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.