# Design document for communication with CY8C9560A through RT5350A



*Fig 1. High level block diagram of interfaces*

*Drivers application refers to the application provided by Atif Shabir*

*User’s application refers to the application developed by Sergey*

Fig 1 shows a high level design view. It has RT5350 board as the main board. Peripheral board is the one which used CY8C9560A as port expander functionality. The description of the applications is as follows.

## Drivers application

This application will be completed by Atif Shabir. It takes care of everything related to the I2C driver and its APIs such as ioctl() and read()/write(). It will use the built in driver for I2C communication with the slave addresses of CY8C9560A chip. The application has two interfaces; On one side it communicates with CY8C9560A over I2C, and on the other side it communicates with the user’s application over TCP with loopback IP address 127.0.0.1. The port and ip address can be configure in the configuration file as describe in this document.

## User’s application

User’s application will be completed by Sergey’s team. This application will communicate with CY8C9560A through the driver’s application. All the messages, which this application wants to send to CY8C9560A chip, will be sent to the driver’s application with a specific format which will then be delivered to the CY8C9560A chip by the driver’s application of I2C. The message format is described in section. The user’s application does not need to know about the internals of the I2C driver details as they will be handled by the driver’s application.

## Compilation of the code

The code is developed in C using eclipse IDE. The project is created with the name “DemonAppCY8C9560A”. Cross compiler tools should be set in the compiler and linker section of the project setting accordingly.

## Running of the code

The final executable of the successfully compiled code can be run from the terminal anywhere. The steps needed to run the demon are

1. Insert the kernel module for i2c. This will make the i2c dev available for the demon. The command is;

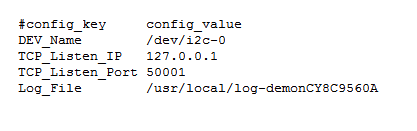
*modprobe /lib/modules/$(uname –r)/i2c-dev.ko*

1. Now run the demon. The command for running the demon in the background is

.*/DemonAppCY8C9560A > null &*

By default the demon will user configure file placed at */usr/local/config-demonCY8C9560A.* This can be changed using command line arguments of the Demon. Run ./*DemonAppCY8C9560A –h* for help which will show the option.

The above two steeps should be added to the initialization script that is to be autorun just after bootup.



*Fig 3. View of sample configuration file.*

The configuration file contains parameters such as I2C device name, Tcp listen ip for the demon, tcp listen socket for the demon and log output file.

## Message formats between demon application and user application

The user’s application will act as TCP client and will connect to the demon application on its listened port and IP address. After the TCP connection has been established, messages will be sent to the demon by user’s application and the demon application will send the response for each packet. The length of the response message will always be equal to the length of command message. So if user’s application wants to read long bytes of data then it must send as many bytes to the demon by padding garbage. For example if the user’s application wants to read 20 bytes from EEPROM then it must append garbage of 20 bytes in the fields designated for the data so that the response length is equal to the command length. Otherwise the response data will be truncated.

|  |  |  |  |
| --- | --- | --- | --- |
| **Byte 00** | **Bytes 01-04** | **Bytes 05** | **Bytes 06** |
| Msg type | Sequence No | Demon Response | Message body |

*Table 1. TCP Message format*

**Msg Type:**

First byte of every message will contain the message type.

The values of message type are;

|  |  |
| --- | --- |
| **Msg type** | **Description** |
| 0x01 | Open Device |
| 0x02 | Close Device |
| 0x03 | Command |
| 0x04 | Reserved |
| 0x05 | Reserved |

*Table 2. Response field from demon*

**Sequence No:**

Four bytes sequence number. This will be used to tally the response received from the demon.

**Demon response:**

This is the demon response byte. This byte is filled by the demon when sending a response. The values of this field are explained in table

These values are explained in table 1.

|  |  |
| --- | --- |
| **Response** | **Description** |
| 0x00 | Succes |
| 0x01 | Invalid Msg Type |
| 0x02 | Invalid Cmd |
| 0x03 | Invalid Params |
| 0x04 | Invalid PortNo |
| 0x05 | Invalid Port Config Code |
| 0x06 | Invalid Port Operation |
| 0x07 | Invalid Pin Operation |
| 0x08 | Invalid Pin Config Code |
| 0x09 | Invalid PWM Operation |
| 0x0A | Invalid PWM No |
| 0x0B | Invalid PWM Params |
| 0x0C | Invalid IntStatus Operation |
| 0x0D | Invalid POR Operation |
| 0x0E | Write To Dev Failed |
| 0x0F | Read From Dev Failed |
| 0x10 | Opening I2C Dev Failed |
| 0x11 | I2C Dev Already Opened |
| 0x12 | I2C Dev Is Not Opened |
| 0x13 | Invalid EEPROM Operation |
| 0x14 | Dev Addr Failed |
| 0x15 | Inalid MultiPort Reg Operation |

*Table 3. Response field from demon*

Messages with command are further explained in table 1 and 2.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Byte06** | **Byte07** | **Byte08** | **Byte09** | **Byte10** | **Byte11** |  | **Description** |
| 0x01 | *GPort* | 0x01 | *Value* |  |  | Port Operations | Read *Value* from *GPort* |
| 0x01 | *GPort* | 0x02 | *Value* |  |  | Write *value* to *GPort* |
| 0x01 | *GPort* | 0x03 | 0x01 | *Value* |  | Interrupt mask for *GPort* |
| 0x01 | *GPort* | 0x03 | 0x02 | *Value* |  | Set output to PWM for *GPort* |
| 0x01 | *GPort* | 0x03 | 0x03 | *Value* |  | Invert *GPort.PinNo* |
| 0x01 | *GPort* | 0x03 | 0x04 | *Value* |  | Set pin direction of *GPort* |
| 0x01 | *GPort* | 0x03 | 0x05 | *Value* |  | Set drive mode for *GPort* |
| 0x02 | *PWMNo* | PWM operation explained in table 4 | | | | | |
| 0x03 | 0x01 | *value* |  |  |  | Misc |  |
| 0x03 | 0x02 | *value* |  |  |  |  |
| 0x04 | *GPort* | *PinMask* | 0x01 | *Value* |  | Pin Operations | Read *Value* from *GPort.PinMask* |
| 0x04 | *GPort* | *PinMask* | 0x02 | *Value* |  | Write *value* to *GPort. PinMask* |
| 0x04 | *GPort* | *PinMask* | 0x03 | 0x01 | *Value* | Interrupt mask for *GPort. PinMask* |
| 0x04 | *GPort* | *PinMask* | 0x03 | 0x02 | *Value* | Set output to PWM for *GPort. PinMask* |
| 0x04 | *GPort* | *PinMask* | 0x03 | 0x03 | *Value* | Invert *GPort. PinMask* |
| 0x04 | *GPort* | *PinMask* | 0x03 | 0x04 | *Value* | Set pin direction of *GPort. PinMask* |
| 0x04 | *GPort* | *PinMask* | 0x03 | 0x05 | *Value* | Set drive mode for *GPort. PinMask* |
| 0x05 | 0x01 |  |  |  |  | POR | Store current config to EEPROM defaults |
| 0x05 | 0x02 |  |  |  |  | Restore factory defaults |
| 0x05 | 0x03 |  |  |  |  |  | Restore POR from EEPROM |
| 0x06 | *GPort* | 0x01 |  |  |  | Int status | Read from the int status register of *GPort* |
| 0x06 | *GPort* | 0x02 |  |  |  | Write to the int status register of *GPort* |
| 0x07 | 0x01 | *Addr* | *Value* |  |  | Multiport reg | Write *Value* to multiport address *Addr* |
| 0x07 | 0x02 | *Addr* | *Value* |  |  | Read *Value* from multiport address *Addr* |

*Table 4. List of commands for register operations like port and pwm.*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Byte 06** | **Byte 07** | **Byte 08-09** | **Byte 12** | **Byte 13** | **Byte 14 onwards** |
| 0x81 | 0x01 | *Address* | *Len* | | *Data of size Len* |
| 0x81 | 0x02 | *Address* | *Len* | |  |

*Table 5. List of commands for EEPROM operations.*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Byte 06** | **Byte 07** | **Byte 08** | **Byte 09** | **Byte 10** | **Byte 11** | **Byte 12** | **Byte 13** |
| 0x02 | PWMNo | OpCode  1 = Set  2 = Get | Clk Src | Clk Div | Period | Pl. Wid | Int Edge |

*Table 6. Command format for pwm config.*

Table 4 shows the list of commands to be used with the port operations. Table 5 shows the list of commands used for EEPROM operations. Table 6 and 7 show pwm operation parameters.

|  |  |
| --- | --- |
| 0x00 | 32 kHz (fixed) |
| 0x01 | 24MHz (fixed) |
| 0x02 | 1.5MHz (fixed) |
| 0x03 | 93.75kHz (fixed) |
| 0x04 | Programable by divider |

*Table 7. Clk sources for pwm.*

Examples of port and EEPROM operations are;

Example 1:

User’s app asks demon to open the i2c device. The command will be

0x01,0x0000000000

Byte 00= 0x01 (Message type = Open device)

Byte 01-04 = 0x00000000 (sequence number)

Byte 05 = 0x00 (success. Will be set by the demon in response)

The demon will open the device and send back the response. If failed the device could not be op (ened then the response message will be

0x01,0x00000000,0x10.

Byte 05 = 0x10 (Dev could not be opened) (see the table x)

Example 2:

If user’s application wants to write 0x0A to GPort1. The TCP message sent by the user’s application will be

0x03, 0x0000000000,0x01,0x01, 0x00,0x02, 0x01, 0x0A

Byte 00= 0x03 (Message type = Device command)

Byte 01-04 = 0x00000000 (sequence number)

Byte 05 = 0x00 (response byte)

Byte 06 = 0x01 (Command type = Port operation)

Byte 07 = 0x01 (Target Port = GPort1)

Byte 08 = 0x02 (Port operation = Write to port)

Byte 09 = 0x0A (Write value = 10)

Example 3:

If user’s application wants to write 1 to Pin0 and Pin6 of GPort3. The TCP message sent by the user’s application will be

0x03, 0x0000000000,0x01,0x01, 0x00,0x02, 0x01, 0x01

Byte 00= 0x03 (Message type = Device command)

Byte 01-04 = 0x00000000 (sequence number)

Byte 05 = 0x00 (response byte)

Byte 06 = 0x04 (Command type = Pin operation)

Byte 07 = 0x03 (Target Port = GPort3)

Byte04 = 0x41 (Pin mask = binary 01000001, 0th and 6th pin only )

Byte 08 = 0x02 (Pin operation = Write to pin)

Byte 09 = 0x01 (Write value = 1)

Example 4:

If user’s application wants to configure GPort2 Pin No 4 as PWM output. The TCP message sent by the user’s application will be

0x03, 0x0000000000,0x01,0x01, 0x00,0x02, 0x01, 0x0A

Byte 00= 0x03 (Message type = Device command)

Byte 01-04 = 0x00000000 (sequence number)

Byte 05 = 0x00 (response byte)

Byte 06 = 0x04 (Command type = Pin operation)

Byte 07 = 0x02 (Target Port = GPort2)

Byte04 = 0x10 (Pin mask = binary 00010000, 4th pin only )

Byte 08 = 0x03 (Pin operation = configure)

Byte 09 = 0x02 (Configuration = Set output to PWM)

Byte 10 = 0x01 (value = 1 which means set to PWM out)

Example 5:

User’s application wants to write 8 bytes of data to EEPROM starting address 0xC005. The TCP message sent by the user’s application will be

0x03, 0x0000000000,0x81, 0x01, 0xC005,0x0008,0x01,0x02,0x03,0x04,0x05,0x06,0x06,0x07,0x08

Byte 00= 0x03 (Message type = device command)

Byte 01-04 = 0x00000000 (sequence number)

Byte 05 = 0x00 (response byte)

Byte 06 = 0x81 (command type = EEPROM operation)

Byte 07 = 0x01 (EEPROM operation = write to EEPROM)

Byte 08-09 = 0xC005 (Address = 0xC005)

Byte 10-11 = 0x0008 (Size = 8 bytes)

Byte 12-19 = 0x010203040506060708 (Data = 0x01 upto 0x08)

Example 6:

Users application wants to set a pwm output on Pin 0 of Port 0.

In order to act as pwm output for a pin, the pin has to be configured as output, pwm\_out, logic 1.

First of all Pin 0 of Port 0 will be configured as output, logic1 out, and pwm out via pin configuration command messages. Then following command message will be sent

0x03, 0x0000000000,0x02,0x07,0x01,0x04,0x0A,0x05,0x03,0x00

Byte 00= 0x03 (Message type = Device command)

Byte 01-04 = 0x00000000 (sequence number)

Byte 05 = 0x00 (response byte)

Byte 06 = 0x02 (Command type = PWM operation)

Byte 07 = 0x07 (PWM Number = 7. Port0-Pin0 is works with pwm 7)

Byte04 = 0x01 (PWM Settings = set(write))

Byte 08 = 0x04 (PWM clock source 0x04 (93.75 kHz)(see data sheet))

Byte 09 = 0x0A (PWM divider = 10)

Byte 10 = 0x05 (PWM period = 5)

Byte 11 = 0x03 (PWM pulse width = 3)

Byte 12 = 0x00 (PWM initerrupt edge = falling)

Output results will be as follows

PWM Freqency = clk source / (divider \* period) = 93750 / (10 \* 5) = 1875 Hz

PWM Duty cycle = (PWM pulse width / PWM Perio) = 3/5 = 60% .

Note: Divider value will always be taken as 1 when clk source 1,2,3 are selected.