# 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 UDP with loopback IP address 127.0.0.1. The ports are given as command line arguments of the applications.

Dear Atif, the driver should be separated from user space application (in this way other applications will be able to use it in the same time), so the structure should be as follow:

Driver (character device/dev/cyg8c9560a) to/from User space demon to/from User space application.

Driver will communicate to/from User space demon via ioctl(), read(),write().

User space demon will communicate to/from User space application via TCP with loopback IP address 127.0.0.1 and IP address will have given as command line argument of the demon (it may be both internal and external address).

Second command line parameter (if exist) will tell the demon to write log of all commands and actions to /tmp/CY8C9560A\_log.

The driver should be registered on system while we will start User space demon.

When we will stop User space demon, it will close all its open sockets.

## 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.

User Application, which describe above it is User space demon, which should be written as part of your job, because it is part of CY8C9560A memory configuration.

User Application, which should be written by my team, so API to User space demon should be for example as follow:

port\_out, port\_in, port\_set\_interrupt\_mask, set\_pwm\_config, set\_port\_config, pin\_out, pin\_in, pin\_set\_interrupt\_mask, set\_pin\_pwm\_config, set\_pin\_config, save\_boot\_configand separate read and write toEEPROM commands, port\_pwn\_enable (“1” for enable “0” for disable), end so on, it should cover all possibilities of CY8C9560A and it should be able to configure one or more separate pins on one or many ports.

Also should be commands which contain multiply commands in one packet for many ports, for example: 0x55 0x03 0x66 0x01 0x02 0x03 0x01 0xff 0x02

0x55 – multiply command for many ports, 0x03 – for three ports, 0x01 – port 1, 0x02 – port 2,0x03 – port 3, 0x56 – port\_out, 0x01 – this value will appear port 1 as output, 0xff – this value will appear on port 2 as output, 0x02 – this value will appear port 3 as output.

Also should be commands which contain multiply commands in one packet for many pins of one port, for example: 0x75 0x3 0x76 0x1 0x2 0x3 0x1 0x1 0x0

0x75 – multiply command for many pins, 0x3 – for three pins, 0x1 – pin 1, 0x2 – pin 2,0x3 – pin 3, 0x76 – pin\_out, 0x1 – this value will appear on pin 1 as output, 0x1 – this value will appear on pin 2 as output, 0x0 – this value will appear port 3 as output.

For all commands, User space demon will answer ERROR/OK.

So you should define complete API to/from User Space Application.

Finally, I should get:

1. All sources files written in C.
2. All instructions how to use the driver.
3. All instructions how to use API (should be detailed explained).
4. All instructions how and where to put source files in order to recompile them.

## Message formats between demon application and user application

The demon application will listen for TCP connections on loop back IP address and port specified by command line arguments for that application so that user’s may be able to use their own specified ports. The user’s application will act as TCP client and will connect to the demon application on its listened port. After the TCP connection has been established messages will be sent to the demon by user’s application and the demon application will ACK or NACK each message. First byte of every message will contain the message type. Every message will be ACKed when its post processing has been completed by the demon. NACK will be issued in case of a command unable to be processed.

The values of message type are;

0x01: Open device

0x02: Close device

0x03: Command

0x04: ACK

0x05: NACK

These values are explained in table 1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Byte 00 (Message type) | Byte 01-04  (Seq Num) | Byte 05  (Err code) | Byte 02 onwards (message body) | Description |
| 0x01 |  |  |  | Open the I2C Dev |
| 0x02 |  |  | N/A | Close any opened device |
| 0x03 |  |  | Command | Issue the command (read or write etc) |
| 0x04 |  |  | N/A | ACK of the previous command |
| 0x05 |  |  | N/A | NACK of the previous command |

*Table 1. Message types from user’s application to demon.*

Messages with command are further explained in table 2 and 3.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Byte06** | **Byte07** | **Byte08** | **Byte09** | **Byte10** | **Byte11** |  | **Description** |
| 0x01 | *GPort* | 0x01 |  |  |  | Port Operations | Read 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 |  |  |  |  | Misc | Get device ID |
| 0x03 | 0x02 | value |  |  |  | Set watch dog timer with *value* |
| 0x04 | *GPort* | *PinMask* | 0x01 |  |  | Pin Operations | Read 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 |
| 0x06 | *GPort* | 0x01 |  |  |  | Int status | Read from the int status register of *GPort* |
| 0x06 | *GPort* | 0x02 |  |  |  | Write to the int status register of *GPort* |

*Table 2. 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 3. 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 | Clk Src | Clk Div | Period | Pl. Wid | Int Edge |

*Table 3. Command format for pwm config.*

Table 2 shows the list of commands to be used with the port operations. Table 3 shows the list of commands used for EEPROM operations.

Examples of port and EEPROM operations are;

Example 1:

User’s want to open the device present at /dev/CY8C9560A, The TCP message sent by the user’s application will be

0x01,0x0000000000, ”/dev/CY8C9560A”.

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

Byte 06-16 = ”/dev/CY8C9560A” (Device path)

Then demon will open the device and send ACK message to the user’s application. The ack message will be

Byte 00= 0x04 (Message type = ACK)

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 10 = 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 10 = 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 10 = 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 0x1000000A. The TCP message sent by the user’s application will be

0x81, 0x0000000000,0x04, 0x01, 0x1000000A,0x0008,0x01,0x02,0x03,0x04,0x05,0x06,0x06,0x07,0x08

Byte 00= 0x81 (Message type = device command)

Byte 10 = 0x04 (command type = EEPROM operation)

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

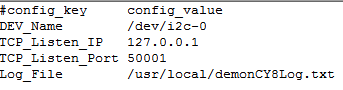
Byte 08-07 = 0x1000000A (Address = 0x10000000A)

Byte 08-09 = 0x0008 (Size = 8 bytes)

Byte 10-17 = 0x010203040506060708 (Data = 0x01 upto 0x08)

# Demon configuration File

By default configurations for the demon are located in the file /usr/local/config-demonCY8C9560A. However this file path can be modified with –c option of the demon. The file contains following parameters;



*Fig 3. View of configuration file.*

# Demon log file

All the actions along-with tcp messages are logged in the file configured in config-file.

# TCP msg error codes:

Upon each message received by the demon from user’s app it sends back the response with specific error codes. List of these error codes is is given in table 5

|  |  |
| --- | --- |
| 0x00 | errCodeSucces, |
| 0x01 | errCodeInvalidMsgType, |
| 0x02 | errCodeInvalidCmd, |
| 0x03 | errCodeInvalidParams, |
| 0x04 | errCodeInvalidPortNo, |
| 0x05 | errCodeInvalidPortConfigCode, |
| 0x06 | errCodeInvalidPortOperation, |
| 0x07 | errCodeInvalidPinOperation, |
| 0x08 | errCodeInvalidPinConfigCode, |
| 0x09 | errCodeInvalidPWMOperation, |
| 0x10 | errCodeInvalidPWMNo, |
| 0x11 | errCodeInvalidPWMParams, |
| 0x12 | errCodeInvalidIntStatusOperation, |
| 0x13 | errCodeInvalidPOROperation, |
| 0x14 | errCodeWriteToDevFailed, |
| 0x15 | errCodeReadFromDevFailed, |
| 0x16 | errCodeOpeningI2CDevFailed, |
| 0x17 | errCodeI2CDevAlreadyOpened, |
| 0x18 | errCodeI2CDevNotOpened, |
| 0x19 | errCodeInvalidEEPROMOperation, |
| 0x20 | errCodeDevAddrFailed |

*Table 5. Error codes from demon to user app*