

API Reference

12/13/2014

Content

1	DIO library	2
1.1.	DI_Read	2
1.2.	DO_Read	3
1.3.	DO_Write	3
1.4.	DO_WriteBit	3
2	Temperature library	4
2.1.	Sensor_GetID	4
2.2.	Sensor_ReadByID	4
2.3.	Sensor_ReadAll	4
3	Network library	5
2.4.	BSD socket library	5
2.4.1.	Reference link: http://en.wikipedia.org/wiki/Berkeley_sockets ...	5
2.5.	Network connection functions	5
2.5.1.	netconn_new	5
2.5.2.	netconn_bind	5
2.5.3.	netconn_listen	6
2.5.4.	netconn_accept	6
2.5.5.	netconn_recv	6
2.5.6.	netconn_write	6
2.5.7.	netconn_close	7
2.5.8.	netconn_delete	7
2.6.	lwip socket library	7
2.6.1.	lwip_socket	7
2.6.2.	lwip_bind	7
2.6.3.	lwip_listen	8
2.6.4.	lwip_accept	8
2.6.5.	lwip_recv	8
2.6.6.	lwip_send	8
2.6.7.	lwip_close	8
4	UART library	9
3.1.	prg_uart_Open	9
3.2.	prg_uart_Close	9
3.4.	prg_uart_FlowControl	9

3.5.	pgr_uart_RxTimeoutThreshold.....	9
3.6.	pgr_uart_puts	9
3.7.	pgr_uart_putc	9
3.9.	pgr_uart_gets.....	9
3.10.	pgr_uart_getc.....	9
3.11.	pgr_uart_Available.....	9
3.12.	pgr_uart_SendBreak	9
3.13.	pgr_uart_GetCTS.....	9
3.14.	pgr_uart_GetDCD.....	9
3.15.	pgr_uart_SetRTS	9
3.16.	pgr_uart_GetRTS.....	9
3.17.	pgr_uart_SetDTR.....	9
3.18.	pgr_uart_GetDTR	9
3.19.	pgr_uart_SetDSR.....	9
3.20.	pgr_uart_GetDSR	10
5	EEPROM library	10
4.1.	pgr_eeprom_Open.....	10
4.2.	pgr_eeprom_Close.....	10
4.3.	pgr_eeprom_Erase.....	10
4.4.	pgr_eeprom_WriteBlock.....	10
4.5.	pgr_eeprom_ReadBlock.....	10
4.6.	get_network_settings	10
4.7.	get_uart_settings	10
4.8.	set_network_settings	10
4.9.	set_uart_settings	10
6	Web library.....	10
5.1.	user_web_init	10

1 DIO library

1.1.DI_Read

1.1 int DI_Read(unsigned char *output)

1.1.1 output[out]: Digital Input value in byte:

1.1.1.1 output[0]: MSB: DI#8, LSB: DI#1

1.1.1.2 output[1]: MSB: DI#16, LSB: DI#9

1.1.2 return:

1.1.2.1 number of DI channel: 16

1.2.DO_Read

1.2 int DO_Read(unsigned char *output)

1.2.1 output[out]: DO (Relay) value in byte:

1.2.1.1 output[0]: MSB: DO#8, LSB: DO#1

1.2.2 return:

1.2.2.1 number of DO channel: 8

1.3.DO_Write

1.3 int DO_Write(unsigned char do_value)

1.3.1 do_value[in]: DO(Relay) value in byte:

1.3.1.1 MSB: DO#8, LSB: DO#1

1.3.2 return:

1.3.2.1 number of DO channel: 8

1.4.DO_WriteBit

1.4 int DO_WriteBit(int no, unsigned char do_value)

1.4.1 no[in]: DO channel number

1.4.2 do_value[in]: 0 or 1

1.4.3 return:

1.4.3.1 0:OK

1.4.3.2 -1:channel number error

1.4.3.3 -2: error in value

2 Temperature library

2.1. Sensor_GetID

- 2.1 int Sensor_GetID(unsigned char *output)
 - 2.1.1 output[out]: sensor ID (8 bytes long per sensor):
 - 2.1.1.1 output[0]~output[7]: Sensor#1
 - 2.1.1.2 output[8]~output[15]: Sensor#2
 - 2.1.1.3 output[16]~output[23]: Sensor#3
 - 2.1.2 return:
 - 2.1.2.1 sensor number:

2.2.Sensor_ReadByID

- 2.2 int Sensor_ReadByID(unsigned char *id, int unit, float *output)
 - 2.2.1 id[in]: Sensor ID, 8 bytes: id[0] ~id[7]
 - 2.2.2 unit[in]: 0: degree C, 1: degree F
 - 2.2.3 output[out]: temperature value
 - 2.2.4 return:
 - 2.2.4.1 0: OK
 - 2.2.4.2 -1: ID error

2.3.Sensor_ReadAll

- 2.3 int Sensor_ReadAll(int unit, Sensor_struct *resp)
 - 2.3.1 unit[in]: 0: degree C, 1: degree F
 - 2.3.2 resp[out]: Sensor_struct: unsigned ID[8], float value
 - 2.3.3 return:
 - 2.3.3.1 number of sensor: 0~3

3 Network library

2.4. BSD socket library

2.4.1. Reference link: http://en.wikipedia.org/wiki/Berkeley_sockets

We support:

- 2.4.1.1. `int socket(int domain, int type, int protocol)`
- 2.4.1.2. `int bind(int s, const struct sockaddr *name, socklen_t namelen)`
- 2.4.1.3. `int listen(int s, int backlog)`
- 2.4.1.4. `int accept(int s, struct sockaddr *addr, socklen_t *addrlen)`
- 2.4.1.5. `int recv(int s, void *mem, size_t len, int flags)`
- 2.4.1.6. `int send(int s, void *dataptr, size_t size, int flags)`
- 2.4.1.7. `int closesocket(int s)`

2.5. Network connection functions

2.5.1. `netconn_new`

- 2.5.1.1. `struct netconn * netconn_new(enum netconn_type type)`
- 2.5.1.2. Creates a new connection abstraction structure. The argument can be one of `NETCONN_TCP` or `NETCONN_UDP`, yielding either a TCP or a UDP connection. No connection is established by the call to this function and no data is sent over the network.

2.5.2. `netconn_bind`

- 2.5.2.1. `err_t netconn_bind(struct netconn *conn, ip_addr_t *addr, u16_t port)`
- 2.5.2.2. Binds the connection `conn` to the local IP address `addr` and TCP or UDP port `port`. If `addr` is `NULL` the local IP address is determined by the networking system.

2.5.3. netconn_listen

2.5.3.1. `err_t netconn_listen(struct netconn *conn)`

2.5.3.2. Puts the TCP connection `conn` into the TCP LISTEN state.

2.5.4. netconn_accept

2.5.4.1. `err_t netconn_accept(struct netconn *conn, struct netconn **new_conn)`

2.5.4.2. Blocks the process until a connection request from a remote host arrives on the TCP connection `conn`. The connection must be in the LISTEN state so `netconn_listen()` must be called prior to `netconn_accept()`. When a connection is established with the remote host, a new connection structure is returned.

2.5.5. netconn_recv

2.5.5.1. `err_t netconn_recv(struct netconn *conn, struct netbuf **new_buf)`

2.5.5.2. Blocks the process while waiting for data to arrive on the connection `conn`. If the connection has been closed by the remote host, NULL is returned, otherwise a `netbuf` containing the received data is returned.

2.5.6. netconn_write

2.5.6.1. `err_t netconn_write(struct netconn *conn, const void *dataptr, size_t size, u8_t apiflags)`

2.5.6.2. This function is only used for TCP connections. It puts the data pointed to by `dataptr` on the output queue for the TCP connection `conn`. The length of the data is given by `size`. There is no restriction on the length of the data. This function does not require the application to explicitly allocate buffers, as this is taken care of by the stack. The `apiflags` parameter has two possible states, as shown below.

2.5.6.2.1. `#define NETCONN_NOCOPY 0x00`

`#define NETCONN_COPY 0x01`

When passed the flag `NETCONN_COPY` the data is copied into internal buffer which is allocated for the data. This allows the data to be modified directly after the call, but is inefficient both in terms of execution time and memory usage. If the flag `NETCONN_NOCOPY` is used, the data is not copied but rather referenced.

The data must not be modified after the call, since the data can be put on the retransmission queue for the connection, and stay there for an indeterminate amount of time. This is useful when sending data that is located in ROM and therefore is immutable. If greater control over the modifiability of the data is needed, a combination of copied and non-copied data can be used, as seen in the example below.

2.5.7. netconn_close

2.5.7.1. `err_t netconn_close(struct netconn *conn)`

2.5.7.2. Closes the connection `conn`.

2.5.8. netconn_delete

2.5.8.1. `err_t netconn_delete(struct netconn *conn)`

2.5.8.2. Deallocates the netconn `conn`. If the connection is open, it is closed as a result of this call.

2.6. lwip socket library

2.6.1. lwip_socket

2.6.1.1. `int lwip_socket(int domain, int type, int protocol)`

2.6.1.2. The `lwip_socket()` call allocates a socket. The parameters to `lwip_socket()` are used to specify what type of socket that is requested. Since this socket API implementation is concerned only with network sockets, these are the only socket type that is supported. Also, only UDP (SOCK_DGRAM) or TCP (SOCK_STREAM) sockets can be used.

2.6.2. lwip_bind

2.6.2.1. `int lwip_bind(int s, const struct sockaddr *name, socklen_t namelen)`

2.6.2.2. The `lwip_bind()` call binds the socket to a local address. In the call to `lwip_bind()` the local IP address and port number are specified.

2.6.3. lwip_listen

2.6.3.1. int lwip_listen(int s, int backlog)

2.6.3.2. Puts the TCP connection conn into the TCP LISTEN state. The backlog parameter is ignored in this version.

2.6.4. lwip_accept

2.6.4.1. int lwip_accept(int s, struct sockaddr *addr, socklen_t *addrlen)

2.6.4.2. If no pending connections are present on the queue, and the socket is not marked as nonblocking, lwip_accept() blocks the caller until a connection is present. If the socket is marked nonblocking and no pending connections are present on the queue, lwip_accept() fails with the error **EAGAIN** or **EWOULDBLOCK**.

2.6.5. lwip_recv

2.6.5.1. int lwip_recv(int s, void *mem, size_t len, int flags)

2.6.5.2. The lwip_recv() call is used on a connected socket to receive data. They can be used for both TCP and UDP connections.

2.6.6. lwip_send

2.6.6.1. int lwip_send(int s, void *dataptr, size_t size, int flags)

2.6.6.2. The lwip_send() call is used in both UDP and TCP connection for sending data. Before a call to lwip_send() the receiver of the data must have been set up using lwip_connect(). For UDP sessions, the lwip_send() call resembles the netconn_send() function from the lwIP API, but since the lwIP API require the application to explicitly allocate buffers, a buffer must be allocated and deallocated within the lwip_send() call. Therefore, a buffer is allocated and the data is copied into the allocated buffer.

2.6.7. lwip_close

2.6.7.1. int lwip_close(int s)

2.6.7.2. Closes the connection conn

4 UART library

3.1. prg_uart_Open

3.1.1. int prg_uart_Open(int port_no, int type);

3.2. prg_uart_Close

3.2.1. int prg_uart_Close(int port_no);

3.3. prg_uart_Config

3.3.1. int prg_uart_Config(int port_no, unsigned long baud, char *serial_mode);

3.4. prg_uart_FlowControl

3.4.1. int prg_uart_FlowControl(int port_no , char *flowmode);

3.5. pgr_uart_RxTimeoutThreshold

3.5.1. int pgr_uart_RxTimeoutThreshold(int port_no, int vtime, int vmin);

3.6. prg_uart_puts

3.6.1. int prg_uart_puts(int port_no , void *buf , size_t size);

3.7. prg_uart_putc

3.8. int prg_uart_putc(int port_no , int c);

3.9. prg_uart_gets

3.9.1. int prg_uart_gets(int port_no, void *buf , size_t size);

3.10. prg_uart_getc

3.10.1. int prg_uart_getc(int port_no);

3.11. prg_uart_Available

3.11.1. int prg_uart_Available(int port_no);

3.12. prg_uart_SendBreak

3.12.1. int prg_uart_SendBreak(int port_no,int duration);

3.13. prg_uart_GetCTS

3.13.1. int prg_uart_GetCTS(int port_no);

3.14. prg_uart_GetDCD

3.14.1. int prg_uart_GetDCD(int port_no);

3.15. prg_uart_SetRTS

3.15.1. int prg_uart_SetRTS(int port_no,int state);

3.16. prg_uart_GetRTS

3.16.1. int prg_uart_GetRTS(int port_no);

3.17. prg_uart_SetDTR

3.17.1. int prg_uart_SetDTR(int port_no,int state);

3.18. prg_uart_GetDTR

3.18.1. int prg_uart_GetDTR(int port_no);

3.19. prg_uart_SetDSR

3.19.1. int prg_uart_SetDSR(int port_no,int state);

3.20. prg_uart_GetDSR

3.20.1. int prg_uart_GetDSR(int port_no);

5 EEPROM library

4.1. prg_eeprom_Open

4.1.1. int prg_eeprom_Open(void)

4.2. prg_eeprom_Close

4.2.1. int prg_eeprom_Close(void)

4.3. prg_eeprom_Erase

4.3.1. int prg_eeprom_Erase(void)

4.4. prg_eeprom_WriteBlock

4.4.1. size_t prg_eeprom_WriteBlock(unsigned long addr, unsigned char *data_buf,
unsigned short tot_len)

4.5. prg_eeprom_ReadBlock

4.5.1. size_t prg_eeprom_ReadBlock(unsigned int addr, char *buffer, size_t size)

4.6. get_network_settings

4.6.1. int get_network_settings(struct net_settings *ptr_net_cfg)

4.6.2. get network settings

4.7. get_uart_settings

4.7.1. int get_uart_settings(int port_no, struct serial_settings *ptr_serial_cfg)

4.7.2. get serial UART settings

4.8. set_network_settings

4.8.1. int set_network_settings(struct net_settings *ptr_net_cfg)

4.8.2. set network setting for next boot

4.9. set_uart_settings

4.9.1. int set_uart_settings(int port_no, struct serial_settings *ptr_serial_cfg)

4.9.2. set UART setting for next boot

6 Web library

5.1. user_web_init

5.1.1. void user_web_init(unsigned short (* user_web))(char *name, char
*ajax_buf_ptr, char method, char *post_data, unsigned short post_len))

5.1.2. callback function when file open

5.1.3. web_ajax.c : please refer to example program in E05_web&settings